Algorithms for programmers
ideas and source code

This document is work in progress: read the "important remarks" near the beginning

Jörg Arndt
arndt@jjj.de

Draft version[1] of 2005-November-20

[1]The latest version and the accompanying software is online at http://www.jjj.de/fxt/.

Contents

Important remarks about this document **xiii**

I Low level and combinatorial algorithms 1

1 Bit wizardry **3**
1.1 Trivia . . . 3
1.2 Copying and swapping bits . . . 7
1.3 Avoiding branches . . . 8
1.4 Operations on low bits or blocks of a word . . . 11
1.5 Computing the index of a single set bit . . . 13
1.6 Operations on high bits or blocks of a word . . . 15
1.7 Functions related to the base-2 logarithm . . . 17
1.8 Counting the bits of a word . . . 18
1.9 Swapping bits or blocks of a word . . . 20
1.10 Isolating blocks of bits and single bits . . . 21
1.11 Bit set lookup . . . 23
1.12 Bitwise rotation of a word . . . 24
1.13 Functions related to bitwise rotation . . . 25
1.14 Reversing the bits of a word . . . 28
1.15 Bitwise zip . . . 32
1.16 Gray code and parity . . . 33
1.16.1 The parity of a binary word . . . 34
1.16.2 Byte-wise Gray code and parity . . . 35
1.16.3 Incrementing (counting) in Gray code . . . 35
1.16.4 The Golay-Rudin-Shapiro sequence * . . . 37
1.16.5 The green code . . . 37
1.17 Bit sequency . . . 38
1.18 Powers of the Gray code . . . 39
1.19 Invertible transforms on words . . . 41
1.20 Moves of the Hilbert curve . . . 47
1.21 The Z-order . . . 48
1.22 Scanning for zero bytes . . . 49
1.23 Manipulation of colors . . . 50
1.24 2-adic inverse and square root . . . 52
1.25 Radix −2 representations . . . 53
1.26 A sparse signed binary representation . . . 55
1.27 Generating bit combinations . . . 56
1.28 Generating bit subsets . . . 59
1.29 Binary words in lexicographic order . . . 59
1.30 Minimal-change bit combinations . . . 63

1.31 Minimal-change Fibonacci words * . . . 65
1.32 Binary words and parentheses * . . . 67
1.33 Error detection by hashing: the CRC . . . 69
1.33.1 Optimization via lookup tables . . . 72
1.33.2 Parallel CRCs . . . 73
1.34 Permutations via primitives . . . 74
1.35 CPU instructions often missed . . . 77

2 Permutations **79**
2.1 The revbin permutation . . . 79
2.2 The radix permutation . . . 83
2.3 In-place matrix transposition . . . 84
2.4 Revbin permutation and matrix transposition * . . . 85
2.5 The zip permutation . . . 87
2.6 The reversed zip permutation . . . 89
2.7 The XOR permutation . . . 90
2.8 The Gray code permutation . . . 91
2.8.1 Cycles of the permutation . . . 92
2.8.2 In-place routines . . . 93
2.8.3 Performance of the routines . . . 93
2.9 The reversed Gray code permutation . . . 94
2.10 The green code permutation . . . 96
2.11 The reversed green code permutation . . . 96
2.12 Decomposing permutations . . . 97
2.13 General permutations . . . 99
2.13.1 Basic definitions and operations . . . 99
2.13.2 Compositions of permutations . . . 101
2.13.3 Representation as disjoint cycles . . . 101
2.13.4 Inverse and square, in-place . . . 103
2.13.5 Powers of a permutation . . . 104
2.13.6 Applying permutations to data, in-place . . . 105
2.13.7 Random permutations . . . 107

3 Sorting and searching **109**
3.1 Sorting . . . 109
3.2 Binary search . . . 111
3.3 Index sorting . . . 112
3.4 Pointer sorting . . . 114
3.5 Sorting by a supplied comparison function . . . 114
3.5.1 Sorting complex numbers . . . 115
3.5.2 Index and pointer sorting . . . 116
3.6 Determination of unique elements . . . 117
3.7 Unique elements with inexact types . . . 118
3.8 Determination of equivalence classes . . . 121
3.8.1 Examples . . . 122
3.8.2 The number of equivalence relations for a set of n elements . . . 125
3.9 Determination of monotonicity and convexity * . . . 125
3.10 Heap-sort . . . 127
3.11 Counting sort and radix sort . . . 128
3.12 Searching in unsorted arrays . . . 130
3.12.1 Minimal and maximal elements in unsorted arrays . . . 130
3.12.2 Searching for values . . . 132
3.12.3 Counting values . . . 132
3.12.4 Searching matches . . . 133

3.12.5 Selecting matches: grep . . . 133

4 Data structures 135
4.1 Stack (LIFO) . . . 135
4.2 Ring buffer . . . 137
4.3 Queue (FIFO) . . . 138
4.4 Deque (double-ended queue) . . . 140
4.5 Heap and priority queue . . . 142
4.6 Bit-array . . . 145
4.7 Finite-state machines . . . 147
4.8 Emulation of coroutines . . . 149

5 Combinatorial algorithms 153
5.1 Combinations in lexicographic order . . . 153
5.2 Combinations in co-lexicographic order . . . 155
5.3 Combinations in minimal-change order . . . 156
5.4 A recursive routine for combinations . . . 157
5.5 Parenthesis strings and restricted growth strings . . . 159
5.6 Integer partitions . . . 164
5.7 Set partitions . . . 168
5.8 Compositions in co-lexicographic order . . . 172
5.9 Compositions in minimal-change order . . . 175
5.10 Compositions and combinations . . . 176
5.11 Subsets in lexicographic order . . . 177
5.12 Subsets in minimal-change order . . . 179
5.13 Subsets ordered by number of elements . . . 181
5.14 Subsets ordered with De Bruijn sequences . . . 182
5.15 k-subsets where k lies in a given range . . . 183
5.16 Mixed radix numbers in lexicographic order * . . . 187
5.17 Permutations . . . 189
5.17.1 Lexicographic order . . . 190
5.17.2 An order from mixed-radix counting . . . 193
5.17.3 Minimal-change order . . . 195
5.17.4 Derangement order . . . 197
5.17.5 Star-transposition order . . . 199
5.17.6 Permutations with restricted prefixes . . . 200
5.18 Gray codes for Fibonacci and Pell words . . . 202
5.18.1 Gray code for Fibonacci words . . . 202
5.18.2 Gray code for Pell words . . . 204
5.19 A Gray code for sparse signed binary words . . . 204
5.20 A Gray code from Hilbert's space-filling curve * . . . 206
5.21 A string substitution engine . . . 206
5.22 Necklaces and Lyndon words . . . 210
5.22.1 Generating all necklaces . . . 211
5.22.2 The number of necklaces . . . 214
5.22.3 The number of necklaces with fixed content . . . 216

6 Shift registers 217
6.1 Linear feedback shift registers (LFSR) . . . 217
6.2 Galois and Fibonacci setup . . . 220
6.3 The number of m-sequences and De Bruijn sequences . . . 221
6.4 Auto correlation of m-sequences . . . 222
6.5 Feedback carry shift register (FCSR) . . . 224
6.6 Linear hybrid cellular automata (LHCA) . . . 225

6.6.1 Conversion to binary polynomials . . . 226
6.6.2 Conversion of irreducible binary polynomials to LHCAs . . . 229
6.7 Additive linear hybrid cellular automata . . . 230

7 Searching paths in directed graphs 233
7.1 Representation of digraphs . . . 234
7.2 Searching full paths . . . 235
7.2.1 Paths in the complete graph: permutations . . . 236
7.2.2 Paths in the De Bruijn graph: De Bruijn sequences . . . 237
7.2.3 A modified De Bruijn graph: complement-shift sequences . . . 239
7.3 Conditional search . . . 239
7.3.1 Modular adjacent changes (MAC) Gray codes . . . 240
7.3.2 Adjacent changes (AC) Gray codes . . . 241
7.4 Sorting the edges: lucky paths . . . 244
7.5 Gray codes for Lyndon words . . . 245
7.5.1 Graph search with edge sorting . . . 246
7.5.2 An optimized algorithm . . . 248

II Fast orthogonal transforms 251

8 The Fourier transform 253
8.1 The discrete Fourier transform . . . 253
8.2 Summary of definitions of Fourier transforms * . . . 254
8.3 Radix-2 FFT algorithms . . . 256
8.3.1 Decimation in time (DIT) FFT . . . 256
8.3.2 Decimation in frequency (DIF) FFT . . . 259
8.4 Saving trigonometric computations . . . 261
8.4.1 Using lookup tables . . . 261
8.4.2 Recursive generation . . . 262
8.5 Higher radix FFT algorithms . . . 263
8.5.1 Decimation in time . . . 263
8.5.2 Decimation in frequency . . . 264
8.5.3 Implementation of radix-r FFTs . . . 264
8.5.4 Radix-4 DIT FFT codes . . . 265
8.5.5 Radix-4 DIF FFT codes . . . 268
8.6 Split-radix Fourier transforms (SRFT) . . . 270
8.7 Symmetries of the Fourier transform . . . 273
8.8 Inverse FFT for free . . . 274
8.9 Real valued Fourier transforms . . . 275
8.9.1 Real valued FT via wrapper routines . . . 277
8.9.2 Real valued split-radix Fourier transforms . . . 278
8.10 Multidimensional Fourier transforms . . . 282
8.11 The matrix Fourier algorithm (MFA) . . . 283

9 Algorithms for fast convolution 285
9.1 Definition and computation via FFT . . . 285
9.2 Correlation . . . 289
9.3 Mass storage convolution using the MFA . . . 291
9.4 Weighted Fourier transforms and convolutions . . . 293
9.5 Half cyclic convolution for half the price? . . . 295
9.6 Convolution using the MFA . . . 296
9.6.1 The case $R = 2$ . . . 296
9.6.2 The case $R = 3$ . . . 297

9.7 Convolution of real valued data using the MFA . . . 297
9.8 The z-transform (ZT) . . . 298
9.9 Prime length FFTs . . . 299

10 The Walsh transform and its relatives 303
10.1 The Walsh transform: Walsh-Kronecker basis . . . 303
10.2 The Kronecker product . . . 305
10.3 A variant of the Walsh transform * . . . 308
10.4 Higher radix Walsh transforms . . . 309
10.5 Localized Walsh transforms . . . 312
10.6 Dyadic convolution . . . 317
10.7 The Walsh transform: Walsh-Paley basis . . . 319
10.8 Sequency ordered Walsh transforms . . . 320
10.9 The slant transform . . . 326
10.10 The Reed-Muller transform . . . 328
10.11 The arithmetic transform . . . 331
10.12 Hadamard and conference matrices * . . . 334
10.12.1 Hadamard matrices via LFSR . . . 334
10.12.2 Hadamard matrices via conference matrices . . . 336
10.12.3 Conference matrices via finite fields . . . 338
10.13 Eigenvectors of the Walsh transform * . . . 341

11 The Haar transform 343
11.1 The 'standard' Haar transform . . . 343
11.2 In-place Haar transform . . . 344
11.3 Non-normalized Haar transforms . . . 347
11.4 Transposed Haar transforms . . . 348
11.5 The reversed Haar transform . . . 350
11.6 Relations between Walsh- and Haar- transforms . . . 353

12 The Hartley transform 357
12.1 Definition and symmetries . . . 357
12.2 Radix-2 FHT algorithms . . . 358
12.2.1 Decimation in time (DIT) FHT . . . 358
12.2.2 Decimation in frequency (DIF) FHT . . . 361
12.3 Complex FT by HT . . . 363
12.4 Complex FT by complex HT and vice versa . . . 364
12.5 Real FT by HT and vice versa . . . 365
12.6 Higher radix FHT algorithms . . . 366
12.7 Convolution via FHT . . . 367
12.8 Negacyclic convolution via FHT . . . 370
12.9 Localized FHT algorithms . . . 371
12.10 Two-dimensional FHTs . . . 372
12.11 Discrete cosine transform (DCT) by HT . . . 373
12.12 Discrete sine transform (DST) by DCT . . . 375
12.13 Automatic generation of transform codes . . . 375
12.14 Eigenvectors of the Fourier and Hartley transform * . . . 378

13 Number theoretic transforms 379
13.1 Basic concepts from number theory . . . 379
13.2 Implementation of NTTs . . . 380
13.2.1 Radix-2 DIT NTT . . . 381
13.2.2 Radix-2 DIF NTT . . . 382
13.2.3 Radix-4 NTTs . . . 383

13.3 Convolution with NTTs . . . 385

III Arithmetical algorithms 387

14 Fast multiplication and exponentiation 389
14.1 Asymptotics of algorithms . . . 389
14.2 The Karatsuba algorithm . . . 390
14.3 Fast multiplication via FFT . . . 391
14.4 Radix/precision considerations with FFT multiplication . . . 393
14.5 The sum-of-digits test . . . 394
14.6 Binary exponentiation . . . 395
14.6.1 Cost of binary exponentiation of full-precision numbers . . . 397

15 Root extraction 399
15.1 Division, square root and cube root . . . 399
15.1.1 Inverse and division . . . 399
15.1.2 Inverse square root . . . 400
15.1.3 Cube root extraction . . . 401
15.1.4 Improved iteration for the square root . . . 401
15.1.5 A different view on the iterations . . . 401
15.2 Square root extraction for rationals . . . 402
15.3 Iterations for the inverse a-th root . . . 404
15.4 Initial approximations for iterations . . . 407
15.4.1 Inverse roots . . . 407
15.4.2 Exponential function . . . 408
15.5 Some applications of the matrix square root . . . 408
15.5.1 Re-orthogonalization . . . 408
15.5.2 Polar decomposition . . . 410
15.5.3 Sign decomposition . . . 411
15.5.4 Pseudo-inverse . . . 411
15.6 Goldschmidt's algorithm . . . 412
15.6.1 An algorithm for the a-th root . . . 414
15.6.2 An algorithm for the inverse a-th root . . . 414
15.6.3 Higher order algorithms for the inverse a-th root . . . 415
15.7 Products for the a-th root . . . 415
15.7.1 Second order products . . . 416
15.7.2 Products of arbitrary order . . . 417

16 Iterations for the inversion of a function 419
16.1 Iterations and their rate of convergence . . . 419
16.2 Schröder's formula . . . 420
16.3 Householder's formula . . . 422
16.4 Dealing with multiple roots . . . 423
16.5 More iterations . . . 425
16.6 Improvements by the delta squared process . . . 427

17 Transcendental functions and the AGM 429
17.1 The AGM . . . 429
17.2 Computation of the logarithm . . . 431
17.3 Computation of the exponential function . . . 434
17.4 Logarithms of power series * . . . 437
17.5 The elliptic functions K and E . . . 439
17.5.1 Elliptic K . . . 439

17.5.2 Elliptic E . . . 440
17.6 Computation of $\pi/\log(q)$ . . . 442
17.7 Computation of $q = \exp(-\pi K'/K)$ . . . 443
17.8 Computation of π . . . 444

18 Computing the elementary functions with limited resources 451
18.1 Shift-and-add algorithms for $\log_b(x)$ and b^x . . . 451
18.1.1 Computing the base-b logarithm . . . 451
18.1.2 Computing b^x . . . 453
18.1.3 An alternative algorithm for the logarithm . . . 454
18.2 CORDIC algorithms . . . 456
18.2.1 The circular case: sine and cosine . . . 456
18.2.2 The linear case: multiplication and division . . . 458
18.2.3 The hyperbolic case: *sinh* and *cosh* . . . 458

19 Recurrences and Chebyshev polynomials 461
19.1 Recurrences . . . 461
19.1.1 Fast computation using matrix powers . . . 462
19.1.2 Faster computation using polynomial arithmetic . . . 463
19.1.3 Inhomogeneous recurrences . . . 465
19.1.4 Recurrence relations for subsequences . . . 467
19.1.5 Binet forms for recurrences . . . 468
19.1.6 Generating functions for recurrences . . . 470
19.1.7 Logarithms of generating functions * . . . 471
19.2 Chebyshev polynomials . . . 471
19.2.1 Recurrence relation, generating functions and the composition law . . . 474
19.2.2 Index-doubling and relations between T and U . . . 474
19.2.3 Fast computation of the Chebyshev polynomials . . . 476
19.2.4 Relations to approximations of the square root * . . . 478

20 Miscellaneous arithmetical algorithms 481
20.1 The binary splitting algorithm for rational series . . . 481
20.2 The magic sumalt algorithm . . . 485
20.3 Continued fractions . . . 488
20.3.1 Computation of the convergents . . . 490
20.3.2 Implementation in pari/gp . . . 491
20.3.3 Subsequences of convergents . . . 493
20.3.4 Fast computation as infinite product . . . 493
20.3.5 Miscellaneous relations for continued fractions . . . 494
20.4 Hypergeometric functions . . . 496
20.4.1 Transformations . . . 499
20.4.2 Examples: elementary functions . . . 504
20.4.3 The function x^x . . . 507
20.4.4 Elliptic K and E . . . 508
20.5 AGM based algorithms for hypergeometric functions . . . 512
20.6 Multiplication of hypercomplex numbers * . . . 514
20.6.1 The Cayley-Dickson construction . . . 515
20.6.2 Fast multiplication of quaternions . . . 517

21 Arithmetic modulo m and some number theory 519
21.1 Implementation of the arithmetic operations . . . 519
21.2 Modular reduction with structured primes . . . 524
21.3 The sieve of Eratosthenes . . . 525
21.4 The Chinese Remainder Theorem (CRT) . . . 527

21.5 The order of an element . . . 529
21.6 Prime modulus: the field $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p = GF(p)$ . . . 530
21.7 Composite modulus: the ring $\mathbb{Z}/m\mathbb{Z}$ . . . 531
21.7.1 Cyclic and noncyclic rings . . . 532
21.7.2 Generators in cyclic rings . . . 534
21.7.3 Generators in noncyclic rings . . . 534
21.8 Quadratic residues . . . 535
21.9 The Rabin-Miller test for compositeness . . . 537
21.9.1 Pseudoprimes and strong pseudoprimes . . . 538
21.9.2 The Rabin-Miller test . . . 539
21.10 Proving primality . . . 542
21.10.1 Pratt's certificate of primality . . . 542
21.10.2 Tests for $n = k\,2^t + 1$ . . . 544
21.10.3 Tests for $n = k\,2^t - 1$ . . . 547
21.10.4 Lucas-Lehmer test with floats * . . . 550
21.10.5 Combined primality tests * . . . 550
21.11 Complex moduli . . . 553
21.11.1 The construction of complex numbers . . . 553
21.11.2 Complex finite fields . . . 553
21.11.3 Efficient reduction modulo $x^2 + x + 1$, $x^2 - x - 1$ and $x^2 + 1$ . . . 554
21.11.4 An algorithm for primitive 2^j-th roots . . . 555
21.11.5 Primitive 2^j-th roots with Mersenne primes . . . 556
21.11.6 Cosine and sine in $GF(p^2)$ . . . 557
21.11.7 Cosine and sine in $GF(p)$ . . . 557
21.11.8 Decomposing $p = 4\,k + 1$ as sum of two squares . . . 558
21.12 Solving $x^2 - d\,y^2 = \pm 1$ . . . 560
21.13 Multigrades * . . . 562
21.14 Properties of the convergents of $\sqrt{2}$ * . . . 563

22 Binary polynomials **569**
22.1 The basic arithmetical operations . . . 569
22.2 Multiplication for polynomials of high degree . . . 574
22.3 Arithmetic modulo a polynomial . . . 579
22.4 Irreducible and primitive polynomials . . . 581
22.4.1 Roots of primitive polynomials have maximal order . . . 582
22.4.2 Testing for irreducibility . . . 582
22.4.3 Irreducible polynomials of special forms * . . . 586
22.4.4 Testing for primitivity . . . 589
22.4.5 Primitive polynomials of special forms * . . . 591
22.5 The number of irreducible and primitive polynomials . . . 593
22.6 Generating irreducible polynomials from necklaces . . . 594
22.7 Irreducible and cyclotomic polynomials * . . . 596
22.8 Factorization of binary polynomials . . . 598
22.8.1 Factorization of squarefree polynomials . . . 598
22.8.2 Extracting the squarefree part of a polynomial . . . 601
22.8.3 Factorization of arbitrary polynomials . . . 602

23 Binary finite fields: $GF(2^n)$ **605**
23.1 Arithmetic and basic properties . . . 605
23.2 Implementation . . . 608
23.3 Minimal polynomials . . . 610
23.4 Solving quadratic equations . . . 612
23.5 Representation by matrices * . . . 613
23.6 Representation by normal bases . . . 616

23.7 Conversion between normal and polynomial representation . . . 619
23.8 Optimal normal bases (ONB) . . . 619
23.8.1 Type-1 optimal normal bases . . . 620
23.8.2 Type-2 optimal normal bases . . . 620
23.9 Gaussian normal bases . . . 621
23.9.1 Computation of the multiplication matrix . . . 622
23.9.2 Determination of the field polynomial . . . 623
23.10 The number of binary normal polynomials . . . 626
23.11 A construction for normal polynomials of degree 2^n * . . . 630

24 Synthetic Iterations * 633
24.1 A variation of the iteration for the inverse . . . 633
24.2 An iteration related to the Thue constant . . . 636
24.3 An iteration related to the Golay-Rudin-Shapiro sequence . . . 637
24.4 Iterations related to the ruler function . . . 639
24.5 An iteration related to the period-doubling sequence . . . 641
24.5.1 Generalizations . . . 643
24.6 An iteration from substitution rules with sign . . . 645
24.7 Iterations related to the sum of digits . . . 646
24.7.1 Weighted sum of digits . . . 647
24.8 Iterations related to the binary Gray code . . . 648
24.8.1 Series where coefficients are the Gray code of exponents . . . 648
24.8.2 Differences of the Gray code . . . 649
24.8.3 Sum of Gray code digits . . . 650
24.8.4 Differences of the sum of Gray code digits . . . 651
24.8.5 Weighted sum of Gray code digits . . . 652
24.9 A function that encodes the Hilbert curve . . . 653
24.9.1 A simplified algorithm . . . 654
24.9.2 The turns of the Hilbert curve . . . 656
24.10 Sparse variants of the inverse . . . 656
24.10.1 A fourth order iteration . . . 656
24.10.2 A different fourth order iteration . . . 657
24.10.3 A sixth order iteration . . . 659
24.11 An iteration related to the Fibonacci numbers . . . 659
24.11.1 Modified initializations . . . 660
24.11.2 Fibonacci representation . . . 661
24.11.3 Fibonacci representation as binary numbers . . . 661
24.11.4 Digit extract algorithms for the rabbit constant . . . 663
24.12 An iteration related to the Lucas numbers . . . 665
24.13 Iterations related to the Pell numbers . . . 666
24.13.1 Pell palindromes . . . 667
24.13.2 Pell representation . . . 668
24.13.3 Pell Gray code . . . 669

A Machine used for benchmarking 673

B The pseudo language Sprache 675

Bibliography 677

Index 687

Important remarks about this document

This is a draft of a text about selected algorithms. The audience in mind are programmers who are interested in the treated algorithms and actually want to create and understand working and reasonably optimized code.

The style varies somewhat which I do not consider bad per se: While some topics (as fast Fourier transforms) need a clear and explicit introduction others (like the bit wizardry chapter) seem to be best presented by basically showing the code with just a few comments.

Some sections are rather sketchy as I include new sections as soon as they contain anything useful.

The pseudo language Sprache is used when I see a clear advantage to do so, mainly when the corresponding C++ does not appear to be self explanatory. Larger pieces of code are presented in C++.

C programmers do not need to be shocked by the '++' as only a rather minimal set of the C++ features is used. Some of the code, especially in part 3 (Arithmetical algorithms), is given in the pari/gp language as the use of other languages would likely bury the idea in technicalities.

A printable version of this text will always stay online for free download. The referenced sources are online as part of FXT (fast transforms and low level routines [FXT]) or hfloat (high precision floating point algorithms [hfloat]).

The reader is welcome to criticize and make suggestions. Thanks go to those[1] who helped to improve this document so far!

In case you want to cite this document, take a look at entry [109] in the bibliography. Please avoid referencing individual chapters or sections as their numbers (and title) might change.

Enjoy reading!

Legal matters

This book is copyright © by its author, Jörg Arndt.
Redistributing or selling this book in printed or in electronic form is prohibited.
This book must not be mirrored on the internet.
Using this book as promotional material is prohibited.
CiteSeer (`http://citeseer.ist.psu.edu/cs/`, and its mirrors) is allowed to keep a copy of this book in its database.

[1] in particular Nathan Bullock, Dominique Delande, Andreas Grünbacher, Jeff Hurchalla, Dirk Lattermann, Gál László, Avery Lee, Brent Lehman, John Lien, Mirko Liss, Doug Moore, David Nalepa, Scott Paine, Yves Paradis, Edith Parzefall, André Piotrowski, David García Quintas, Thomas Schraitle, Ralf Stephan, Michal Staruch, Tony Reix, Vinnie Winkler and Rui Zhang

"Why make things difficult, when it is possible to make them cryptic and totally illogic, with just a little bit more effort?"

– Aksel Peter Jørgensen

Part I

Low level and combinatorial algorithms

Chapter 1

Bit wizardry

In this chapter low-level functions are presented that operate on the bits of a binary word. It is often not obvious what these are good for and I do not attempt much to motivate why particular functions are presented. However, *if* you happen to have an application for a given routine you will love that it is there: The program using it may run significantly faster.

The type `unsigned long` is abbreviated as `ulong` as defined in [FXT: file fxttypes.h].

It is assumed that `BITS_PER_LONG` reflects the size of the C-type `unsigned long`. It is defined in [FXT: file bits/bitsperlong.h] and usually equals the machine word size. That is, it equals 32 on 32-bit architectures and 64 on 64-bit machines. Further, the quantity `BYTES_PER_LONG` shall reflect the number of bytes in a machine word, that is, it equals `BITS_PER_LONG` divided by eight.

The examples of assembler code are for the x86 and the AMD64 architecture. They should be simple enough to be understood by readers knowing the assembler-mnemonics of any CPU.

1.1 Trivia

Little endian versus big endian

The order in which the bytes of an integer are stored in memory can start with the least significant byte (*little endian* machine) or with the most significant byte (*big endian* machine). The hexadecimal number `0x0D0C0B0A` will be stored in the following manner when memory addresses grow from left to right:

```
adr:   z   z+1   z+2  z+3
mem:   0D   0C   0B   0A   // big endian
mem:   0A   0B   0C   0D   // little endian
```

The difference is only visible when you cast pointers. Let `V` be the 32-bit integer with the value above. Then the result of `char c = &(char *)(&V);` will be `0x0A` (value modulo 256) on a little endian machine but `0x0D` (value divided by 2^{24}) on a big endian machine. Portable code that uses casts may need two versions, one for each endianness. Though friends of the big endian way sometimes refer to little endian as 'wrong endian', the wanted result of the shown pointer cast is much more often the modulo operation.

Size of pointer is size of long

On sane architectures a pointer fits into a type `long` integer. When programming for a 32-bit architecture (where the size of `int` and `long` coincide) casting pointers to integers (and back) will work. The same code *will* fail on 64-bit machines. If you have to cast pointers to an integer type, cast them to `long`.

Shifts and division

With two's complement arithmetic (that is: on likely every computer you'll ever touch) division and multiplication by powers of two is right and left shift, respectively. This is true for unsigned types and for multiplication (left shift) with signed types. Division with signed types rounds toward zero, as one would expect, but right shift is a division (by a power of two) that rounds to minus infinity:

```
int a = -1;
int c = a >> 1;   // c == -1
int d = a / 2;    // d ==  0
```

The compiler still uses a shift instruction for the division, but a 'fix' for negative values:

```
  9:test.cc   @ int foo(int a)
 10:test.cc   @ {
285 0003 8B442410              movl 16(%esp),%eax  // move argument to %eax
 11:test.cc   @     int s = a >> 1;
289 0007 89C1                  movl %eax,%ecx
290 0009 D1F9                  sarl $1,%ecx
 12:test.cc   @     int d = a / 2;
293 000b 89C2                  movl %eax,%edx
294 000d C1EA1F                shrl $31,%edx  // fix: %edx=(%edx<0?1:0)
295 0010 01D0                  addl %edx,%eax // fix: add one if a<0
296 0012 D1F8                  sarl $1,%eax
```

For unsigned types the shift would suffice. One more reason to use unsigned types whenever possible.

The assembler listing was generated from C code via the following commands:

```
# create assembler code:
c++ -S -fverbose-asm -g -O2  test.cc -o test.s
# create asm interlaced with source lines:
as -alhnd test.s  > test.lst
```

There are two types of *right* shifts: a so-called logical and an arithmetical shift. The logical version (`shrl` in the above fragment) always fills the higher bits with zeros, corresponding to division[1] of unsigned types. The arithmetical shift (`sarl` in the above fragment) fills in ones or zeros, according to the most significant bit of the original word.

Computing remainders modulo a power of two with unsigned types is equivalent to a bit-and using a mask:

```
ulong a = b % 32;  // == b & (32-1)
```

All of the above is done by the compiler's optimization wherever possible.

Division by constants can be replaced by multiplications and shift. The magic machinery inside the compiler does it for you. A division by the constant 10 is compiled to:

```
  5:test.cc   @ ulong foo(ulong a)
  6:test.cc   @ {
  7:test.cc   @     ulong b = a / 10;
290 0000 8B442404              movl 4(%esp),%eax
291 0004 F7250000              mull .LC33   // value == 0xcccccccd
292 000a 89D0                  movl %edx,%eax
293 000c C1E803                shrl $3,%eax
```

Thereby it is sometimes reasonable to have separate code branches with explicit special values. Similarly, for modulo computations with a constant modulus (using modulus 10,000):

```
  8:test.cc   @ ulong foo(ulong a)
  9:test.cc   @ {
 53 0000 8B4C2404              movl 4(%esp),%ecx
 10:test.cc   @     ulong b = a % 10000;
 57 0004 89C8                  movl %ecx,%eax
 58 0006 F7250000              mull .LC0   // value == 0xd1b71759
 59 000c 89D0                  movl %edx,%eax
 60 000e C1E80D                shrl $13,%eax
 61 0011 69C01027              imull $10000,%eax,%eax
 62 0017 29C1                  subl %eax,%ecx
 63 0019 89C8                  movl %ecx,%eax
```

Algorithms to replace divisions by a constant by multiplications and shifts are given in [52].

[1] So you can think of it as 'unsigned arithmetical' shift.

A pitfall

```
 c=................  -c=................   c=     0  -c=     0  <--=
 c=...............1  -c=1111111111111111   c=     1  -c=    -1
 c=..............1.  -c=111111111111111.   c=     2  -c=    -2
 c=..............11  -c=111111111111111.1  c=     3  -c=    -3
 c=.............1..  -c=11111111111111..   c=     4  -c=    -4
 c=.............1.1  -c=11111111111111.1   c=     5  -c=    -5
 c=.............11.  -c=11111111111111.1.  c=     6  -c=    -6
[--snip--]
 c=.111111111111111  -c=1..............11  c= 32765  -c=-32765
 c=.111111111111111.  -c=1..............1.  c= 32766  -c=-32766
 c=.111111111111111.  -c=1..............1  c= 32767  -c=-32767
 c=1...............  -c=1...............   c=-32768  -c=-32768  <--=
 c=1...............1  -c=.111111111111111  c=-32767  -c= 32767
 c=1..............1.  -c=.111111111111111.  c=-32766  -c= 32766
 c=1..............11  -c=.111111111111111.1  c=-32765  -c= 32765
 c=1.............1..  -c=.11111111111111..  c=-32764  -c= 32764
 c=1.............1.1  -c=.11111111111111.1  c=-32763  -c= 32763
 c=1.............11.  -c=.11111111111111.1.  c=-32762  -c= 32762
[--snip--]
 c=111111111111111.1  -c=..............111  c=    -7  -c=     7
 c=111111111111111.1.  -c=..............11.  c=    -6  -c=     6
 c=111111111111111.11  -c=..............11.1  c=    -5  -c=     5
 c=111111111111111..1  -c=..............1..1  c=    -4  -c=     4
 c=111111111111111..1.  -c=..............1..  c=    -3  -c=     3
 c=111111111111111.1.  -c=..............1.1  c=    -2  -c=     2
 c=111111111111111.11  -c=..............1.11  c=    -1  -c=     1
```

Figure 1.1: With two's complement there is one nonzero value that is its own negative.

In two's complement zero is not the only number that is equal to its negative value. With a data type of n bits the value with just the highest bit set (the most negative value) also has this property. Figure 1.1 (the output of [FXT: demo bits/gotcha-demo.cc]) shows the situation for words of sixteen bits. This is the reason why innocent looking code like

```
if ( x<0 )  x = -x;
// assume x positive here (WRONG!)
```

can simply fail.

Shortcuts

To test whether at least one of `a` and `b` equals zero use `if ( (a+b)==a )`. This works for signed and unsigned integers. Checking whether both are zero can be done using `if ( (a|b)==0 )`. This obviously generalizes for several variables as `if ( (a|b|c|..|z)==0 )`). Test whether exactly one of two variables is zero using `if ( (!a) ^ (!b) )` .

Toggling between values

In order to toggle an integer `x` between two values `a` and `b` use:

```
precalculate:  t  = a ^ b;
toggle:        x ^= t;   // a <--> b
```

the equivalent trick for floats is

```
precalculate:  t = a + b;
toggle:        x = t - x;
```

Integer versus float multiplication

The floating point multiplier gives the highest bits of the product. Integer multiplication gives the result modulo 2^b where b is the number of bits of the integer type used. As an example we square the number 1010101 using a 32-bit integer type and floating point types with 24-bit and 53-bit mantissa:

```
   a = 111111111
a*a = 12345678987654321 // true result

a*a = 1653732529            // result with 32-bit integer multiplication
(a*a)%(2**32) = 1653732529  // ... which is modulo (2**bits_per_int)
```

```
  a*a = 1.2345679481405440e+16 // result with float multiplication (24 bit mantissa)
  a*a = 1.2345678987654320e+16 // result with float multiplication (53 bit mantissa)
```

Manipulation of individual bits

The following functions should be self explaining. Following the spirit of the C language there is no check whether the indices used are out of bounds. That is, if any index is greater or equal `BITS_PER_LONG`, the result is undefined. Find these in [FXT: file bits/bittest.h].

```
inline ulong test_bit(ulong a, ulong i)
// Return zero if bit[i] is set,
//  else return one-bit word with bit[i] set.
{
    return  (a & (1UL << i));
}

static inline bool test_bit01(ulong a, ulong i)
// Return whether bit[i] is set.
{
    return test_bit(a, i);
}

inline ulong set_bit(ulong a, ulong i)
// Return a with bit[i] set.
{
    return  (a | (1UL << i));
}

inline ulong delete_bit(ulong a, ulong i)
// Return a with bit[i] cleared.
{
    return  (a & ~(1UL << i));
}

inline ulong change_bit(ulong a, ulong i)
// Return a with bit[i] changed.
{
    return  (a ^ (1UL << i));
}
```

Optimization considerations

Never ever think that some code is the 'fastest possible', there always another trick that can still improve performance. Many factors can have an influence on performance like number of CPU registers or cost of branches. Code that performs well on one machine might perform badly on another. The old trick to swap variables without using a temporary

```
            //   a=0, b=0   a=0, b=1   a=1, b=0   a=1, b=1
    a ^= b; //     0    0     1    1     1    0     0    1
    b ^= a; //     0    0     1    0     1    1     0    1
    a ^= b; //     0    0     1    0     0    1     1    1

    equivalent to:
    tmp = a;  a = b;  b = tmp;
```

is pretty much out of fashion today. However in some specific context (like extreme register pressure) it may be the way to go.

The only way to find out which version of a function is faster is to actually do profiling (timing). The performance does depend on the stream of instructions before the machine code (we assume that all of these low-level functions get inlined). Studying the generated CPU instructions does help to understand what is going on but can never replace profiling.

Never just replace the unoptimized version of some code fragment when introducing a streamlined one. Keep the original in the source. In case something nasty happens (think of low level software failures when porting to a different platform) you'll be very thankful for the chance to temporarily use the slow but correct version.

Proper documentation is an absolute must for optimized code, just assume that nobody will be able to read and understand it from the supplied source alone. The experience of not being able to understand some piece of code *oneself* has written time ago helps a lot in this matter.

1.2 Copying and swapping bits

Copying a bit

The trick is to generate a one exactly if the bits at the two positions differ. Then an XOR changes the target bit if needed [FXT: file bits/bitcopy.h]:

```
inline ulong copy_bit(ulong a, ulong isrc, ulong idst)
// Copy bit at [isrc] to position [idst]
{
    ulong x = ((a>>isrc) ^ (a>>idst)) & 1;  // one if bits differ
    a ^= (x<<idst); // change if bits differ
}
```

The situation gest slightly tricky if the bit positions are given as (one bit) masks:

```
inline ulong mask_copy_bit(ulong a, ulong msrc, ulong mdst)
// Copy bit according at src-mask (msrc)
//  to the bit according to the dest-mask (mdst).
// Both msrc and mdst must have exactly one bit set.
{
    ulong x = mdst;
    if ( msrc & a )  x = 0;  // zero if source bit set
    x ^= mdst;  // ==mdst if source bit set, else zero
    a &= ~mdst;  // clear dest bit
    a |= x;
    return a;
}
```

The compiler generates branch-free code as the conditional assignment is compiled to a `cmov` (conditional move) assembler instruction. If one or both masks have several bits set the routine will set all bits of `mdst` if any of the bits in `msrc` is one else clear all bits of `mdst`.

Swapping bits

A function to swap the two bits of a word [FXT: file bits/bitswap.h]:

```
static inline ulong bit_swap(ulong a, ulong k1, ulong k2)
// Return a with bits at positions [k1] and [k2] swapped.
// k1==k2 is allowed (a is unchanged then)
{
    ulong x = ((a>>k1) ^ (a>>k2)) & 1;  // one if bits differ
    a ^= (x<<k2); // change if bits differ
    a ^= (x<<k1); // change if bits differ
    return  a;
}
```

When it is known that the bits do have different values the following routine can be used:

```
static inline ulong bit_swap_01(ulong a, ulong k1, ulong k2)
// Return a with bits at positions [k1] and [k2] swapped.
// Bits must have different values (!)
// (i.e. one is zero, the other one)
// k1==k2 is allowed (a is unchanged then)
{
    return  a ^ ( (1UL<<k1) ^ (1UL<<k2) );
}
```

1.3 Avoiding branches

Branches are expensive operations with many CPUs, especially if the CPU pipeline is very long. The function in this section avoid branches, they are given in [FXT: file bits/branchless.h].

The following function returns $\max(0, x)$. That is, zero is returned for negative input, else the unmodified input:

```
static inline long max0(long x)
{
    return  x & ~(x >> (BITS_PER_LONG-1));
}
```

There is no restriction on input range. The trick used is that with negative x the arithmetic shift will give a word of all ones which is then negated and the AND-operation deletes all bits. Similarly:

```
static inline long min0(long x)
// Return min(0, x), i.e. return zero for positive input
{
    return  x & (x >> (BITS_PER_LONG-1));
}
```

Computation of the average of two arguments x and y. The function gives the correct value even if $(x + y)/2$ does not fit into a machine word:

```
static inline ulong average(ulong x, ulong y)
// Use the fact that x+y == ((x&y)<<1) + (x^y)
//    that is:       sum ==  carries   + sum_without_carries
{
    return  (x & y) + ((x ^ y) >> 1);
}
```

If it is known that $x \geq y$ then one can alternatively use the statement `return y+(x-y)/2`.

The following `upos_*()` functions only work for a limited range. The highest bit must not be set in order to have the highest bit emulate the carry flag. Branchless computation of the absolute difference $|(|\, a - b)$:

```
static inline ulong upos_abs_diff(ulong a, ulong b)
{
    long d1 = b - a;
    long d2 = (d1 & (d1>>(BITS_PER_LONG-1)))<<1;
    return  d1 - d2; // == (b - d) - (a + d);
}
```

Sorting of the arguments:

```
static inline void upos_sort2(ulong &a, ulong &b)
// Set {a, b} := {minimum(a, b), maximum(a,b)}
// Both a and b must not have the most significant bit set
{
    long d = b - a;
    d &= (d>>(BITS_PER_LONG-1));
    a += d;
    b -= d;
}
```

The following two functions adjust a given values when it lies outside a given range.

```
static inline long clip_range0(long x, long m)
// Code equivalent (for m>0) to:
//   if ( x<0 )  x = 0;
//   else if ( x>m )  x = m;
//   return  x;
{
    if ( (ulong)x > (ulong)m )  x = m & ~(x >> (BITS_PER_LONG-1));
    return  x;
}

static inline long clip_range(long x, long mi, long ma)
// Code equivalent to (for mi<=ma):
//   if ( x<mi )  x = mi;
//   else if ( x>ma )  x = ma;
{
    x -= mi;
    x = clip_range0(x, ma-mi);
```

```
    x += mi;
    return  x;
}
```

Conditional swap

The following statement is compiled with a branch:

```
     if ( a<b )   { ulong t=a; a=b; b=t; } // swap if a < b
  // Here:   a in  %rcx,    b in  %rdx
  62 000e 4889C8                movq    %rcx, %rax      # X, X
  68 0011 4839D1                cmpq    %rdx, %rcx      # X, X
  69 0014 7306                  jae     .L3     #,   // the branch
  71 0016 4889D1                movq    %rdx, %rcx      # X, X
  72 0019 4889C2                movq    %rax, %rdx      # X, X
  73                    .L3:
```

As conditional assignments can be done branchless, an equivalent branchless version is:

```
     { ulong x=a^b;  if (a>=b) { x=0; }  a^=x;  b^=x; } // swap if a < b
 255 00af 4889EA                movq    %rbp, %rdx      # a, x
 257 00b2 B9000000              movl    $0, %ecx        #, tmp83
 257      00
 260 00b7 4831C2                xorq    %rax, %rdx      # b, x
 261 00ba 4839C5                cmpq    %rax, %rbp      # b, a
 262 00bd 480F43D1              cmovae  %rcx, %rdx      # x,, tmp83, x
 263 00c1 4831D5                xorq    %rdx, %rbp      # x, a
 264 00c4 4831D0                xorq    %rdx, %rax      # x, b
```

We'd like to have fewer instructions. If one tries

```
    { ulong ta=a;  if (a<b) {a=b; b=ta;} } // swap if a < b
```

the generated code is identical to the first version. Let's try the chain saw [FXT: file bits/cswap.h]:

```
static inline void cswap_lt(ulong &a, ulong &b)
// Branchless equivalent to:
// if ( a<b )   { ulong t=a; a=b; b=t; } // swap if a < b
{
    asm volatile("movq %0, %%r15 \n"   // t=a
                 "cmpq %0, %1 \n"      // cmp a, b
                 "cmovae %1, %0 \n"    // cond a=b
                 "cmovae %%r15, %1 \n" // cond b=t
                 ""
                 : "=r" (a), "=r" (b) // output
                 : "0" (a), "1" (b)   // input
                 : "r15" // clobber
                 );
}
```

Looks good:

```
  // Here:   a in  %rax,    b in  %rdx
 111 0027 4989C7        movq %rax, %r15         # tmp71
 112 002a 4839C2        cmpq %rax, %rdx         # tmp71, tmp72
 113 002d 480F43C2      cmovae %rdx, %rax       # tmp72, tmp71
 114 0031 490F43D7      cmovae %r15, %rdx       # tmp72
```

Clearly, the relative speed of the three versions depends on the machine used. But it also turns out to be dependent of the surrounding code. We use bubble sort for benchmarking:

```
void bubble_sort(ulong *f, ulong n)
{
    while ( n-- > 1 )
        for (ulong k=0; k<n; ++k)  cswap_NN(f[k], f[k+1]);
}
```

Were we use the three versions of conditional swap for `cswap_NN()`. We sort an array of length 2^{15} twice with each version, once starting with an already sorted array and once with an array that is sorted in descending order. The 'plain' version wins:

```
    cswap_gt_plain(f[k], f[k+1]); //  3.58s
    cswap_gt_xor(f[k], f[k+1]);   //  6.34s
    cswap_gt(f[k], f[k+1]);       //  5.10s
```

This is due to the fact that the compiler bypasses the store when no swap happens:

```
103 0020 488B4808  movq 8(%rax), %rcx #, tmp71
104 0024 488B10    movq (%rax), %rdx #* f, tmp72
105 0027 4839D1    cmpq %rdx, %rcx # tmp72, tmp71
106 002a 7307      jae .L7 #,
108 002c 48895008  movq %rdx, 8(%rax) # tmp72,
109 0030 488908    movq %rcx, (%rax) # tmp71,* f
111               .L7:
```

If we change the inner loop to

```
for (ulong k=0; k<n; ++k)
{
    ulong a = f[k],  b = f[k+1];
    cswap_NN(a, b);
    f[k] = a;   f[k+1] = b;
}
```

then we obtain:

```
cswap_gt_plain(a, b);  //  5.78s
cswap_gt_xor(a, b);    //  6.22s
cswap_gt(a, b);        //  4.68s
```

Our innocent looking change in the code prevented the compiler from doing its nice trick. The XOR version is (within timing precision) as slow as before. The assembler version wins because the data is already in registers. We learn that profiling is an absolute must.

Your compiler may be smarter than you thought

The machine code generated for

```
x =  x & ~(x >> (BITS_PER_LONG-1));  // max0()
```

is

```
 35:   48 99                    cqto
 37:   48 83 c4 08              add    $0x8,%rsp
 3b:   48 f7 d2                 not    %rdx
 3e:   48 21 d0                 and    %rdx,%rax
```

The variable `x` resides in the register `rAX` both at start and end of the function. The compiler uses a special (AMD64) instruction `cqto`. Quoting [AMD24594]:

> Copies the sign bit in the rAX register to all bits of the rDX register. The effect of this instruction is to convert a signed word, doubleword, or quadword in the rAX register into a signed doubleword, quadword, or double-quadword in the rDX:rAX registers. This action helps avoid overflow problems in signed number arithmetic.

Now the equivalent

```
x = ( x<0 ? 0 : x );  // max0()  "simple minded"
```

is compiled to only three instructions:

```
 35:   ba 00 00 00 00           mov    $0x0,%edx
 3a:   48 85 c0                 test   %rax,%rax
 3d:   48 0f 48 c2              cmovs  %rdx,%rax // note %edx is %rdx
```

A conditional move (`cmovs`) instruction is used here. That is, our optimized version is (on my machine) actually worse than the straightforward equivalent.

A second example is the function `clip_range()` above. It is compiled to

```
  0:   48 89 f8                 mov    %rdi,%rax
  3:   48 29 f2                 sub    %rsi,%rdx
  6:   31 c9                    xor    %ecx,%ecx
  8:   48 29 f0                 sub    %rsi,%rax
  b:   78 0a                    js     17 <_Z2CLlll+0x17>  // the branch
  d:   48 39 d0                 cmp    %rdx,%rax
 10:   48 89 d1                 mov    %rdx,%rcx
```

```
  13:   48 0f 4e c8             cmovle %rax,%rcx
  17:   48 8d 04 0e             lea    (%rsi,%rcx,1),%rax
```

Now we replace the code by

```
inline long clip_range(long x, long mi, long ma)
{
    x -= mi;
    if ( x<0 )  x = 0;
//    else  // commented out to make (compiled) function really branchless
    {
        ma -= mi;
        if ( x>ma )  x = ma;
    }
    x += mi;
}
```

Then the compiler generates branchless code:

```
   0:   48 89 f8                mov    %rdi,%rax
   3:   b9 00 00 00 00          mov    $0x0,%ecx
   8:   48 29 f0                sub    %rsi,%rax
   b:   48 0f 48 c1             cmovs  %rcx,%rax
   f:   48 29 f2                sub    %rsi,%rdx
  12:   48 39 d0                cmp    %rdx,%rax
  15:   48 0f 4f c2             cmovg  %rdx,%rax
  19:   48 01 f0                add    %rsi,%rax
```

Still, with CPUs that do not have a conditional move instruction (or some branchless equivalent of it) the techniques shown in this section can be useful.

1.4 Operations on low bits or blocks of a word

The underlying idea of functions that operate on the lowest set bit is that addition and subtraction of 1 always changes a burst of bits at the lower end of the word. The following functions are given in [FXT: file bits/bitlow.h].

Isolation of the lowest set bit is achieved via

```
static inline ulong lowest_bit(ulong x)
// Return word where only the lowest set bit in x is set.
// Return 0 if no bit is set.
{
    return  x & -x;  // use: -x == ~x + 1
}
```

The lowest zero (unset bit) of some word `x` is then trivially isolated using `lowest_bit( ~x )`. The sequence of returned values for $x = 0, 1, \ldots$ is the binary ruler function, the highest power of two that divides $x + 1$:

```
   x:  ==   x        lowest_zero(x)
   0:  == ........    .......1
   1:  == .......1    ......1.
   2:  == ......11    .....1..
   3:  == .....111    ....1...
   4:  == ....1...    .......1
   5:  == ....1.1.    ......1.
   6:  == ....11..    .....1..
   7:  == ....111.    ....1...
   8:  == ....1...    .......1
   9:  == ....1..1    ......1.
  10:  == ....1.1.    .....1..
```

Clearing the lowest set bit in a word can be achieved via

```
static inline ulong delete_lowest_bit(ulong x)
// Return word were the lowest bit set in x is cleared.
// Return 0 for input == 0.
{
    return  x & (x-1);
}
```

while setting the lowest unset bit is done by

```
static inline ulong set_lowest_zero(ulong x)
// Return word were the lowest unset bit in x is set.
// Return ~0 for input == ~0.
{
    return  x | (x+1);
}
```

Isolate the burst of low bits/zeros as follows:

```
static inline ulong low_bits(ulong x)
// Return word where all the (low end) ones are set.
// Example:  01011011 --> 00000011
// Return 0 if lowest bit is zero:
//       10110110 --> 0
{
    if ( ~0UL==x )  return ~0UL;
    return (((x+1)^x) >> 1);
}
```

and

```
static inline ulong low_zeros(ulong x)
// Return word where all the (low end) zeros are set.
// Example:  01011000 --> 00000111
// Return 0 if all bits are set.
{
    if ( 0==x )  return ~0UL;
    return (((x-1)^x) >> 1);
}
```

Isolation of the lowest block of ones (which may have zeros to the right of it) can be achieved via:

```
static inline ulong lowest_block(ulong x)
// Isolate lowest block of ones.
// e.g.:
// x   = *****011100
// l   = 00000000100
// y   = *****100000
// x^y = 00000111100
// ret = 00000011100
{
    ulong l = x & -x;  // lowest bit
    ulong y = x + l;
    x ^= y;
    return  x & (x>>1);
}
```

Extracting the *index* (position) of the lowest bit is easy when the corresponding assembler instruction is used [FXT: file bits/bitasm-i386.h]:

```
static inline ulong asm_bsf(ulong x)
// Bit Scan Forward
{
    asm ("bsfl %0, %0" : "=r" (x) : "0" (x));
    return x;
}
```

Without the assembler instruction an algorithm that uses proportional $\log_2(\texttt{BITS\_PER\_LONG})$ can be used, so the resulting function may look like[2]

```
static inline ulong lowest_bit_idx(ulong x)
// Return index of lowest bit set.
// Examples:
//    ***1 --> 0
//    **10 --> 1
//    *100 --> 2
// Return 0 (also) if no bit is set.
{
    ulong r = 0;
    x &= -x;  // isolate lowest bit
#if  BITS_PER_LONG >= 64
    if ( x & 0xffffffff00000000 )  r += 32;
    if ( x & 0xffff0000ffff0000 )  r += 16;
    if ( x & 0xff00ff00ff00ff00 )  r += 8;
    if ( x & 0xf0f0f0f0f0f0f0f0 )  r += 4;
```

[2] thanks go to Nathan Bullock for emailing this improved version.

```
    if ( x & 0xccccccccccccccc )  r += 2;
    if ( x & 0xaaaaaaaaaaaaaaaa )  r += 1;
#else // BITS_PER_LONG >= 64
    if ( x & 0xffff0000 )  r += 16;
    if ( x & 0xff00ff00 )  r += 8;
    if ( x & 0xf0f0f0f0 )  r += 4;
    if ( x & 0xcccccccc )  r += 2;
    if ( x & 0xaaaaaaaa )  r += 1;
#endif // BITS_PER_LONG >= 64

#endif // BITS_USE_ASM
    return r;
}
```

The function returns zero for two inputs, one and zero. If one needs a special value for the input zero, add a statement like

```
    if ( 1>=x )  return  x-1; // 0 if 1, ~0 if 0
```

as first line of the function.

Occasionally one wants to set a rising or falling edge at the position of the lowest bit:

```
static inline ulong lowest_bit_01edge(ulong x)
// Return word where a all bits from  (including) the
//   lowest set bit to bit 0 are set.
// Return 0 if no bit is set.
{
    if ( 0==x )  return 0;
    return  x^(x-1);
}

static inline ulong lowest_bit_10edge(ulong x)
// Return word where a all bits from (including) the
//   lowest set bit to most significant bit are set.
// Return 0 if no bit is set.
{
    if ( 0==x )  return 0;
    x ^= (x-1);
    // here  x == lowest_bit_01edge(x);
    return  ~(x>>1);
}
```

1.5 Computing the index of a single set bit

In the function `lowest_bit_idx()` we first isolated the lowest bit of a word `x` by first setting `x&=-x`. At this point, `x` contains just one set bit (or `x==0`). The following lines in the routine implement an algorithm that computes the index of the single bit set. This section gives some alternative techniques to compute the index of a single-bit word.

Cohen's trick

A nice trick is presented in [Cohen]: For N-bit words find a number m so that all powers of two are different modulo m. That is, the order of two modulo m must be greater or equal to N. One uses a table `mt[]` of size m that contains the powers of two: `mt[2**j mod m] = j` for $j > 0$ and a special value for $j = 0$. To look up the index of a one-bit-word `x` it is reduced modulo m and `mt[x]` is returned.

We demonstrate the method for $N = 8$ where $m = 11$ is the smallest number with the required property. The setup routine for the table is

```
const ulong m = 11; // the modulus
ulong mt[m+1];
static void mt_setup()
{
    mt[0] = 0;  // special value for the zero word
    ulong t = 1;
    for (ulong i=1; i<m; ++i)
    {
```

```
        mt[t] = i-1;
        t *= 2;
        if ( t>=m )  t -= m;  // modular reduction
    }
}
```

The entry in `mt[0]` will be accessed when the input is the zero word. One can use a special value that the algorithm will return for input zero. Here we simply used zero in order to always have the same return value as with `lowest_bit_idx()`.

The computation of the index can then be achieved by

```
inline ulong m_lowest_bit_idx(ulong x)
{
    x &= -x;  // isolate lowest bit
    x %= m;   // power of two modulo m
    return  mt[x];  // lookup
}
```

The code is given in the demo [FXT: demo bits/modular-lookup-demo.cc], the output with $N = 8$ (edited for size) is

```
  modulus m=11
     k =  0  1  2  3  4  5  6  7
  mt[k]=  0  0  1  8  2  4  9  7
  Lowest bit == 0:       x= .......1 =   1    x % m= 1 ==> lookup = 0
  Lowest bit == 1:       x= ......1. =   2    x % m= 2 ==> lookup = 1
  Lowest bit == 2:       x= .....1.. =   4    x % m= 4 ==> lookup = 2
  Lowest bit == 3:       x= ....1... =   8    x % m= 8 ==> lookup = 3
  Lowest bit == 4:       x= ...1.... =  16    x % m= 5 ==> lookup = 4
  Lowest bit == 5:       x= ..1..... =  32    x % m= 10 ==> lookup = 5
  Lowest bit == 6:       x= .1...... =  64    x % m= 9 ==> lookup = 6
  Lowest bit == 7:       x= 1....... = 128    x % m= 7 ==> lookup = 7
```

The following moduli m can be used for N bit words:

```
    N:  4   8  16   32   64   128   256
    m:  5  11  19   37   67   131   269
```

The list was obtained via pari/gp:

```
    for(n=2,8,N=2^n;forprime(z=N,N+9999,if(znorder(Mod(2,z))>=N,print(N,": ",z);break())))
```

Using De Bruijn sequences

The following method (given in [74]) is even more elegant, it uses binary De Bruijn sequences of size N. A binary De Bruijn sequence of length 2^N contains all binary words of length N (see section 6.1 on page 217). These are the sequences for 32 and 64 bit, as binary words:

```
#if BITS_PER_LONG == 32
const ulong db = 0x4653ADFUL;
// == 00000100011001010011101011011111
const ulong s = 32-5;
#else
const ulong db = 0x218A392CD3D5DBFUL;
// == 0000001000011000101000111001001011001101001111010101110110111111
const ulong s = 64-6;
#endif
```

Let w_i be the i-th sub-word from the left (high end). We create a table so that the entry with index w_i points to i:

```
ulong dbt[BITS_PER_LONG];
static void dbt_setup()
{
    for (ulong i=0; i<BITS_PER_LONG; ++i)  dbt[ (db<<i)>>s ] = i;
}
```

The computation of the index involves a multiplication and a table lookup:

```
inline ulong db_lowest_bit_idx(ulong x)
{
    x &= -x;  // isolate lowest bit
    x *= db;  // multiplication by a power of two is a shift
```

```
    x >>= s;  // use log_2(BITS_PER_LONG) highest bits
    return  dbt[x];  // lookup
}
```

The used sequences must start with at least $\log_2(N) - 1$ zeros because in the line `x *= db` the word `x` is shifted (not rotated). The code is given in the demo [FXT: demo bits/debruijn-lookup-demo.cc], the output with $N = 8$ (edited for size, dots denote zeros) is

```
db=...1.111
    k  =  0  1  2  3  4  5  6  7
dbt[k] =  0  1  2  4  7  3  6  5

Lowest bit == 0:    x = .......1   db * x = ...1.111   shifted = ........ == 0 ==> lookup = 0
Lowest bit == 1:    x = ......1.   db * x = ..1.111.   shifted = .......1 == 1 ==> lookup = 1
Lowest bit == 2:    x = .....1..   db * x = .1.111..   shifted = ......1. == 2 ==> lookup = 2
Lowest bit == 3:    x = ....1...   db * x = 1.111...   shifted = .....1.1 == 5 ==> lookup = 3
Lowest bit == 4:    x = ...1....   db * x = .111....   shifted = ......11 == 3 ==> lookup = 4
Lowest bit == 5:    x = ..1.....   db * x = 111.....   shifted = .....111 == 7 ==> lookup = 5
Lowest bit == 6:    x = .1......   db * x = 11......   shifted = .....11. == 6 ==> lookup = 6
Lowest bit == 7:    x = 1.......   db * x = 1.......   shifted = .....1.. == 4 ==> lookup = 7
```

Using floating point numbers

Floating point numbers are normalized so that the highest bit in the mantissa is one. Therefore if one converts an integer into a float then the position of the *highest* set bit can be read off the exponent. By isolating the lowest bit before that operation its index can be found by the same trick. However, the conversion between integers and floats is usually slow. Further, the technique is highly machine dependent.

1.6 Operations on high bits or blocks of a word

For the functions operating on the highest bit there is not a way as trivial as with the equivalent task with the lower end of the word. With a bit-reverse CPU-instruction available life would be significantly easier. However, almost no CPU seems to have it. The following functions are given in [FXT: file bits/bithigh.h].

Isolation of the highest set bit is achieved via the bit-scan instruction when it is available [FXT: file bits/bitasm-i386.h]:

```
static inline ulong asm_bsr(ulong x)
// Bit Scan Reverse
{
    asm ("bsrl %0, %0" : "=r" (x) : "0" (x));
    return x;
}
```

else one may use

```
static inline ulong highest_bit_01edge(ulong x)
// Return word where a all bits from (including) the
//   highest set bit to bit 0 are set.
// Return 0 if no bit is set.
{
    x |= x>>1;
    x |= x>>2;
    x |= x>>4;
    x |= x>>8;
    x |= x>>16;
#if  BITS_PER_LONG >= 64
    x |= x>>32;
#endif
    return  x;
}
```

so the resulting code may look like

```
static inline ulong highest_bit(ulong x)
// Return word where only the highest bit in x is set.
// Return 0 if no bit is set.
```

```
{
#if defined  BITS_USE_ASM
    if ( 0==x )  return 0;
    x = asm_bsr(x);
    return  1UL<<x;
#else
    x = highest_bit_01edge(x);
    return  x ^ (x>>1);
#endif // BITS_USE_ASM
}
```

Trivially, the highest zero can be isolated using `highest_bit(~x)`. Thereby

```
static inline ulong set_highest_zero(ulong x)
// Return word were the highest unset bit in x is set.
// Return ~0 for input == ~0.
{
    return  x | highest_bit( ~x );
}
```

Finding the index of the highest set bit uses the equivalent algorithm as with the lowest set bit:

```
static inline ulong highest_bit_idx(ulong x)
// Return index of highest bit set.
// Return 0 if no bit is set.
{
#if defined  BITS_USE_ASM
    return  asm_bsr(x);
#else // BITS_USE_ASM

    if ( 0==x )  return  0;

    ulong r = 0;
#if  BITS_PER_LONG >= 64
    if ( x & (~0UL<<32) )  { x >>= 32;  r += 32; }
#endif
    if ( x & 0xffff0000 )  { x >>= 16;  r += 16; }
    if ( x & 0x0000ff00 )  { x >>=  8;  r +=  8; }
    if ( x & 0x000000f0 )  { x >>=  4;  r +=  4; }
    if ( x & 0x0000000c )  { x >>=  2;  r +=  2; }
    if ( x & 0x00000002 )  {            r +=  1; }
    return r;
#endif // BITS_USE_ASM
}
```

Isolation of the high zeros goes like

```
static inline ulong high_zeros(ulong x)
// Return word where all the (high end) zeros are set.
// e.g.:  00011001 --> 11100000
// Returns 0 if highest bit is set:
//        11011001 --> 00000000
{
    x |= x>>1;
    x |= x>>2;
    x |= x>>4;
    x |= x>>8;
    x |= x>>16;
#if  BITS_PER_LONG >= 64
    x |= x>>32;
#endif
    return  ~x;
}
```

The high bits can be isolated using arithmetical right shift

```
static inline ulong high_bits(ulong x)
// Return word where all the (high end) ones are set.
// e.g.  11001011 --> 11000000
// Returns 0 if highest bit is zero:
//       01110110 --> 00000000
{
    long y = (long)x;
    y &= y>>1;
    y &= y>>2;
    y &= y>>4;
    y &= y>>8;
    y &= y>>16;
#if  BITS_PER_LONG >= 64
```

```
    y &= y>>32;
#endif
    return  (ulong)y;
}
```

In case arithmetical shifts are more expensive than unsigned shifts, instead use

```
static inline ulong high_bits(ulong x)
{
    return  high_zeros( ~x );
}
```

```
        ------------------------------------------------------------
        ................1111....1111.111 = 0xf0f7  == word
        ................1............... = highest_bit
        ................1111111111111111 = highest_bit_01edge
        1111111111111111............... = highest_bit_10edge
                                      15 = highest_bit_idx
        ............................... = low_zeros
        ............................111 = low_bits
        ................................1 = lowest_bit
        ............................1 = lowest_bit_01edge
        11111111111111111111111111111111 = lowest_bit_10edge
                                       0 = lowest_bit_idx
        ............................111 = lowest_block
        ................1111....1111.11. = delete_lowest_bit
        ............................1... = lowest_zero
        ................1111....11111111 = set_lowest_zero
        ............................... = high_bits
        1111111111111111............... = high_zeros
        1............................... = highest_zero
        1...............1111....1111.111 = set_highest_zero
        ------------------------------------------------------------
        1111111111111111....1111....1... = 0xffff0f08  == word
        1............................... = highest_bit
        11111111111111111111111111111111 = highest_bit_01edge
        1............................... = highest_bit_10edge
                                     31 = highest_bit_idx
        ............................111 = low_zeros
        ............................... = low_bits
        ............................1... = lowest_bit
        ............................1111 = lowest_bit_01edge
        11111111111111111111111111111... = lowest_bit_10edge
                                      3 = lowest_bit_idx
        ............................1... = lowest_block
        1111111111111111....1111........ = delete_lowest_bit
        ................................1 = lowest_zero
        1111111111111111....1111....1..1 = set_lowest_zero
        1111111111111111............... = high_bits
        ............................... = high_zeros
        ................1............... = highest_zero
        1111111111111111...1111....1... = set_highest_zero
        ------------------------------------------------------------
```

Figure 1.2: Operations on the highest and lowest bits (and blocks) of a binary word for two different 32-bit input words. Dots denote zeros.

A demonstration of selected functions operating on the highest or lowest bit (or block) of binary words is given in [FXT: demo bits/bithilo-demo.cc]. A part of the output is shown in figure 1.2.

1.7 Functions related to the base-2 logarithm

The following functions are given in [FXT: file bits/bit2pow.h].

The function `ld()` that shall return $\lfloor \log_2(x) \rfloor$ can be implemented using the obvious algorithm:

```
static inline ulong ld(ulong x)
// Return k so that 2^k <= x < 2^(k+1)
// If x==0 then 0 is returned (!)
{
    ulong k = 0;
    while ( x>>=1 )  { ++k; }
    return k;
```

```
}
```

And then, `ld()` is the same as `highest_bit_idx()`, so one can use

```
static inline ulong ld(ulong x)
{
    return  highest_bit_idx(x);
}
```

The bit-wise algorithm can be faster if the average result is known to be small.

The function `one_bit_q()` can be used to determine whether its argument is a power of two:

```
static inline bool one_bit_q(ulong x)
// Return 1 iff x \in {1,2,4,8,16,...}
{
    ulong m = x-1;
    return  (((x^m)>>1) == m);
}
```

The following function does the same except that it returns `true` also for the zero argument:

```
static inline bool is_pow_of_2(ulong x)
// Return 1 if x == 0(!) or x == 2**k
{
    return  !(x & (x-1));
}
```

Occasionally useful in FFT based computations (where the length of the available FFTs is often restricted to powers of two) are

```
static inline ulong next_pow_of_2(ulong x)
// Return x if x=2**k
// else return 2**ceil(log_2(x))
{
    ulong n = 1UL<<ld(x);  // n<=x
    if ( n==x )  return  x;
    else         return  n<<1;
}
```

and

```
static inline ulong next_exp_of_2(ulong x)
// Return k if x=2**k else return k+1.
{
    ulong ldx = ld(x);
    ulong n = 1UL<<ldx;  // n<=x
    if ( n==x )  return  ldx;
    else         return  ldx+1;
}
```

1.8 Counting the bits of a word

If your CPU does not have a bit count instruction (sometimes called 'population count') then you might use an algorithm given in [FXT: file bits/bitcount.h]. The following functions need proportional to $\log_2$(`BITS_PER_LONG`) operations:

```
static inline ulong bit_count(ulong x)
// Return number of bits set
{
    x = (0x55555555 & x) + (0x55555555 & (x>> 1));  // 0-2 in 2 bits
    x = (0x33333333 & x) + (0x33333333 & (x>> 2));  // 0-4 in 4 bits
    x = (0x0f0f0f0f & x) + (0x0f0f0f0f & (x>> 4));  // 0-8 in 8 bits
    x = (0x00ff00ff & x) + (0x00ff00ff & (x>> 8));  // 0-16 in 16 bits
    x = (0x0000ffff & x) + (0x0000ffff & (x>>16));  // 0-31 in 32 bits
    return x;
}
```

The underlying idea is to do a search via bit masks. The code can be improved to either

```
    x  = ((x>>1) & 0x55555555) + (x & 0x55555555);  // 0-2 in 2 bits
    x  = ((x>>2) & 0x33333333) + (x & 0x33333333);  // 0-4 in 4 bits
    x  = ((x>>4) + x) & 0x0f0f0f0f;                 // 0-8 in 4 bits
```

```
    x +=  x>> 8;                                    // 0-16 in 8 bits
    x +=  x>>16;                                    // 0-32 in 8 bits
    return  x & 0xff;
```

or (taken from [AMD22007])

```
    x -=  (x>>1) & 0x55555555;
    x  = ((x>>2) & 0x33333333) + (x & 0x33333333);
    x  = ((x>>4) + x) & 0x0f0f0f0f;
    x *= 0x01010101;
    return  x>>24;
```

Which of the latter two versions is faster mainly depends on the speed of integer multiplication.

For 64-bit words the masks have to be adapted and one more step must be added (example corresponding to the second variant above):

```
    x = ((x>>1) & 0x5555555555555555) + (x & 0x5555555555555555);  // 0-2 in 2 bits
    x = ((x>>2) & 0x3333333333333333) + (x & 0x3333333333333333);  // 0-4 in 4 bits
    x = ((x>>4) + x) & 0x0f0f0f0f0f0f0f0f;                         // 0-8 in 4 bits
    x +=  x>> 8;                                                   // 0-16 in 8 bits
    x +=  x>>16;                                                   // 0-32 in 8 bits
    x +=  x>>32;                                                   // 0-64 in 8 bits
    return  x & 0xff;
```

The slightly weird algorithm

```
static inline ulong bit_count_01(ulong x)
// Return number of bits in a word
// for words of the special form 00...0001...11
{
    ulong ct = 0;
    ulong a;
#if  BITS_PER_LONG == 64
    a = (x & (1UL<<32)) >> (32-5);  // test bit 32
    x >>= a;  ct += a;
#endif
    a = (x & (1<<16)) >> (16-4);  // test bit 16
    x >>= a;  ct += a;

    a = (x & (1<<8)) >> (8-3);  // test bit 8
    x >>= a;  ct += a;

    a = (x & (1<<4)) >> (4-2);  // test bit 4
    x >>= a;  ct += a;

    a = (x & (1<<2)) >> (2-1);  // test bit 2
    x >>= a;  ct += a;

    a = (x & (1<<1)) >> (1-0);  // test bit 1
    x >>= a;  ct += a;

    ct += x & 1; // test bit 0

    return ct;
}
```

avoids all branches and may be useful on a planet with pink air.

Sparse counting

When the (average input) word is known to have only a few bits set the following sparse count variant can be advantageous:

```
static inline ulong bit_count_sparse(ulong x)
// Return number of bits set.
// The loop will execute once for each bit of x set.
{
    if ( 0==x )  return 0;
    ulong n = 0;
    do  { ++n; }  while ( x &= (x-1) );
    return  n;
}
```

Counting blocks

The number of bit-blocks in a binary word can computed by the following function:

```
static inline ulong bit_block_count(ulong x)
// Return number of bit blocks.
// E.g.:
// ..1..11111...111.  -> 3
// ...1..11111...111  -> 3
// ......1.....1.1..  -> 3
// .........111.1111  -> 2
{
    return  bit_count( (x^(x>>1)) ) / 2 + (x & 1);
}
```

Similarly, the number of blocks with two or more bits can be counted via:

```
static inline ulong bit_block_ge2_count(ulong x)
// Return number of bit blocks with at least 2 bits.
// E.g.:
// ..1..11111...111.  -> 2
// ...1..11111...111  -> 2
// ......1.....1.1..  -> 0
// .........111.1111  -> 2
{
    return  bit_block_count( x & ( (x<<1) & (x>>1) ) );
}
```

1.9 Swapping bits or blocks of a word

The functions in this section are given in [FXT: file bits/bitswap.h]

Pairs of adjacent bits can be swapped via

```
static inline ulong bit_swap_1(ulong x)
// Return x with neighbour bits swapped.
{
#if  BITS_PER_LONG == 32
    ulong m = 0x55555555UL;
#else
#if  BITS_PER_LONG == 64
    ulong m = 0x5555555555555555UL;
#endif
#endif
    return  ((x & m) << 1) | ((x & (~m)) >> 1);
}
```

The 64-bit branch is omitted in the following examples.

Groups of 2 bits are swapped by

```
static inline ulong bit_swap_2(ulong x)
// Return x with groups of 2 bits swapped.
{
    ulong m = 0x33333333UL;
    return  ((x & m) << 2) | ((x & (~m)) >> 2);
}
```

Equivalently,

```
static inline ulong bit_swap_4(ulong x)
// Return x with groups of 4 bits swapped.
{
    ulong m = 0x0f0f0f0fUL;
    return  ((x & m) << 4) | ((x & (~m)) >> 4);
}
```

and

```
static inline ulong bit_swap_8(ulong x)
// Return x with groups of 8 bits swapped.
{
    ulong m = 0x00ff00ffUL;
    return  ((x & m) << 8) | ((x & (~m)) >> 8);
```

```
}
```

When swapping half-words (here for 32-bit architectures)

```
static inline ulong bit_swap_16(ulong x)
// Return x with groups of 16 bits swapped.
{
    ulong m = 0x0000ffffUL;
    return  ((x & m) << 16) | ((x & (m<<16)) >> 16);
}
```

GCC is clever enough to recognize that the whole operation is equivalent to a (left or right) word rotation and indeed emits just a single rotate instruction.

Functions for swapping two selected bits are shown in section 1.1 on page 3.

1.10 Isolating blocks of bits and single bits

The following functions are concerned with creation or extraction of bit-blocks, single bits and related tasks.

Creating bit-blocks

The following functions are given in [FXT: file bits/bitblock.h].

```
static inline ulong bit_block(ulong p, ulong n)
// Return word with length-n bit block starting at bit p set.
// Both p and n are effectively taken modulo BITS_PER_LONG.
{
    ulong x = (1UL<<n) - 1;
    return  x << p;
}
```

A version with indices wrapping around is

```
static inline ulong cyclic_bit_block(ulong p, ulong n)
// Return word with length-n bit block starting at bit p set.
// The result is possibly wrapped around the word boundary.
// Both p and n are effectively taken modulo BITS_PER_LONG.
{
    ulong x = (1UL<<n) - 1;
    return  (x<<p) | (x>>(BITS_PER_LONG-p));
}
```

Isolating single bits or zeros

The following functions are given in [FXT: file bits/bitmisc.h].

```
static inline ulong single_bits(ulong x)
// Return word were only the single bits from x are set.
{
    return  x & ~( (x<<1) | (x>>1) );
}

static inline ulong single_zeros(ulong x)
// Return word were only the single zeros from x are set.
{
    return  single_bits( ~x );
}

static inline ulong single_values(ulong x)
// Return word were only the single bits and the
//   single zeros from x are set.
{
    return  (x ^ (x<<1)) & (x ^ (x>>1));
}
```

Isolating single bits or zeros at the word boundary

```
static inline ulong border_bits(ulong x)
// Return word were only those bits from x are set
//   that lie next to a zero.
{
    return  x & ~( (x<<1) & (x>>1) );
}
```

```
static inline ulong border_values(ulong x)
// Return word were those bits/zeros from x are set
//   that lie next to a zero/bit.
{
    ulong g = x ^ (x>>1);
    g |= (g<<1);
    return   g | (x & 1);
}
```

Isolating bits at zero-one transitions

```
static inline ulong high_border_bits(ulong x)
// Return word were only those bits from x are set
//   that lie right to (i.e. in the next lower bin of) a zero.
{
    return  x & ( x ^ (x>>1) );
}
```

```
static inline ulong low_border_bits(ulong x)
// Return word were only those bits from x are set
//   that lie left to (i.e. in the next higher bin of) a zero.
{
    return  x & ( x ^ (x<<1) );
}
```

Isolating bits or zeros at block boundaries

```
static inline ulong block_border_bits(ulong x)
// Return word were only those bits from x are set
//   that are at the border of a block of at least 2 bits.
{
    return  x & ( (x<<1) ^ (x>>1) );
}
```

```
static inline ulong low_block_border_bits(ulong x)
// Return word were only those bits from x are set
//   that are at left of a border of a block of at least 2 bits.
{
    ulong t = x & ( (x<<1) ^ (x>>1) );  // block_border_bits()
    return  t & (x>>1);
}
```

```
static inline ulong high_block_border_bits(ulong x)
// Return word were only those bits from x are set
//   that are at right of a border of a block of at least 2 bits.
{
    ulong t = x & ( (x<<1) ^ (x>>1) );  // block_border_bits()
    return  t & (x<<1);
}
```

```
static inline ulong block_bits(ulong x)
// Return word were only those bits from x are set
//   that are part of a block of at least 2 bits.
{
    return  x & ( (x<<1) | (x>>1) );
}
```

Isolating the interior of bit blocks

```
static inline ulong block_values(ulong x)
```

```
// Return word were only those bits/values are set
//   that do not lie next to an opposite value.
{
    return  ~single_values(x);
}

static inline ulong interior_bits(ulong x)
// Return word were only those bits from x are set
//   that do not have a zero to their left or right.
{
    return  x & ( (x<<1) & (x>>1) );
}

static inline ulong interior_values(ulong x)
// Return word were only those bits/zeros from x are set
//   that do have a zero/bit to their left or right.
{
    return  ~border_values(x);
}
```

1.11 Bit set lookup

There is a nice trick to determine whether a given number is contained in the set $\{0, 1, 2, \ldots, \texttt{BITS\_PER\_LONG}-1\}$. As an example, in order to determine whether `x` is a prime less than 32, one can use the function

```
ulong m = (1UL<<2) | (1UL<<3) | (1UL<<5) | ... | (1UL<<31);  // precomputed
static inline ulong is_tiny_prime(ulong x)
{
    return  m & (1UL << x);
}
```

The same idea applied to lookup tiny factors [FXT: file bits/tinyfactors.h]:

```
static inline bool is_tiny_factor(ulong x, ulong d)
// For x,d < BITS_PER_LONG (!)
// return whether d divides x  (1 and x included as divisors)
// no need to check whether d==0
//
{
    return  ( 0 != ( (tiny_factors_tab[x]>>d) & 1 ) );
}
```

The function uses the precomputed array

```
extern const ulong tiny_factors_tab[] =
{
                   0x0,  // x = 0:           ( bits: ........)
                   0x2,  // x = 1:  1        ( bits: ......1.)
                   0x6,  // x = 2:  1 2      ( bits: .....11.)
                   0xa,  // x = 3:  1 3      ( bits: ....1.1.)
                  0x16,  // x = 4:  1 2 4    ( bits: ...1.11.)
                  0x22,  // x = 5:  1 5      ( bits: ..1...1.)
                  0x4e,  // x = 6:  1 2 3 6  ( bits: .1..111.)
                  0x82,  // x = 7:  1 7      ( bits: 1.....1.)
                 0x116,  // x = 8:  1 2 4 8
                 0x20a,  // x = 9:  1 3 9
[--snip--]
            0x20000002,  // x = 29:  1 29
            0x4000846e,  // x = 30:  1 2 3 5 6 10 15 30
            0x80000002,  // x = 31:  1 31
#if  ( BITS_PER_LONG > 32 )
          0x100010116,  // x = 32:  1 2 4 8 16 32
          0x20000080a,  // x = 33:  1 3 11 33
[--snip--]
    0x2000000000000002,  // x = 61:  1 61
    0x4000000080000006,  // x = 62:  1 2 31 62
    0x800000000020028a   // x = 63:  1 3 7 9 21 63
#endif // ( BITS_PER_LONG > 32 )
};
```

which is defined in [FXT: file bits/tinyfactors.cc].

1.12 Bitwise rotation of a word

Neither C nor C++ have a statement for bitwise rotation[3]. The operations can be 'emulated' [FXT: file bits/bitrotate.h]:

```
static inline ulong bit_rotate_left(ulong x, ulong r)
// Return word rotated r bits to the left
// (i.e. toward the most significant bit)
{
    return  (x<<r) | (x>>(BITS_PER_LONG-r));
}
```

As already mentioned, GCC emits exactly the CPU instruction that is *meant* here, even with non-constant argument `r`. Well done, GCC folks!

Of course the explicit use of the corresponding assembler instruction should not do any harm:

```
static inline ulong bit_rotate_right(ulong x, ulong r)
// Return word rotated r bits to the right
// (i.e. toward the least significant bit)
{
#if defined  BITS_USE_ASM    // use x86 asm code
    return asm_ror(x, r);
#else
    return  (x>>r) | (x<<(BITS_PER_LONG-r));
#endif
}
```

where we used [FXT: file bits/bitasm-i386.h]:

```
static inline ulong asm_ror(ulong x, ulong r)
{
    asm ("rorl   %%cl, %0" : "=r" (x) : "0" (x), "c" (r));
    return x;
}
```

The equivalent for the AMD64 CPU is obtained by changing `rorl` to `rorq` [FXT: file bits/bitasm-amd64.h].

Rotations using only a part of the word length are achieved by

```
static inline ulong bit_rotate_left(ulong x, ulong r, ulong ldn)
// Return ldn-bit word rotated r bits to the left
// (i.e. toward the most significant bit)
// Must have  0 <= r <= ldn
{
    ulong m = ~0UL >> ( BITS_PER_LONG - ldn );
    x &= m;
    x = (x<<r) | (x>>(ldn-r));
    x &= m;
    return  x;
}
```

and

```
static inline ulong bit_rotate_right(ulong x, ulong r, ulong ldn)
// Return ldn-bit word rotated r bits to the right
// (i.e. toward the least significant bit)
// Must have  0 <= r <= ldn
{
    ulong m = ~0UL >> ( BITS_PER_LONG - ldn );
    x &= m;
    x = (x>>r) | (x<<(ldn-r));
    x &= m;
    return  x;
}
```

Finally, the functions

```
static inline ulong bit_rotate_sgn(ulong x, long r, ulong ldn)
// Positive r --> shift away from element zero
{
    if ( r > 0 )  return  bit_rotate_left(x, (ulong)r, ldn);
    else          return  bit_rotate_right(x, (ulong)-r, ldn);
```

[3] which I consider a missing feature.

```
}
```

and (full-word version)

```
static inline ulong bit_rotate_sgn(ulong x, long r)
// Positive r --> shift away from element zero
{
    if ( r > 0 )  return  bit_rotate_left(x, (ulong)r);
    else          return  bit_rotate_right(x, (ulong)-r);
}
```

are sometimes convenient.

1.13 Functions related to bitwise rotation

We give several functions related to cyclic rotations of binary words. The functions shown work on the full length of the words, equivalents for length-`ldn` words are given in the the respective files.

The following function determines whether there is a cyclic right shift of its second argument so that it matches the first argument. It is given in [FXT: file bits/bitcyclic-match.h]:

```
static inline ulong bit_cyclic_match(ulong x, ulong y)
// Return  r if x==rotate_right(y, r) else return ~0UL.
// In other words: return
//   how often the right arg must be rotated right (to match the left)
// or, equivalently:
//   how often the left arg must be rotated left (to match the right)
{
    ulong r = 0;
    do
    {
        if ( x==y )  return r;
        y = bit_rotate_right(y, 1);
    }
    while ( ++r < BITS_PER_LONG );
    return ~0UL;
}
```

The minimum among all cyclic shifts of a word is can be computed via the following function given in [FXT: file bits/bitcyclic-minmax.h]:

```
static inline ulong bit_cyclic_min(ulong x)
// Return minimum of all rotations of x
{
    ulong r = 1;
    ulong m = x;
    do
    {
        x = bit_rotate_right(x, 1);
        if ( x<m )  m = x;
    }
    while ( ++r < BITS_PER_LONG );
    return  m;
}
```

Selecting from all n-bit words those that are equal to their cyclic minimum gives the sequence of the binary length-n necklaces, see section 5.22 on page 210. For example, with 6-bit words:

```
 word   period
......  1
.....1  6
....11  6
...1.1  6
...111  6
..1..1  3
..1.11  6
..11.1  6
..1111  6
.1.1.1  2
.1.111  6
.11.11  3
.11111  6
111111  1
```

The values in the right column can be computed using [FXT: `bit_cyclic_period()` in bits/bitcyclic-period.h]:

```
static inline ulong bit_cyclic_period(ulong x, ulong ldn)
// Return minimal positive bit-rotation that transforms x into itself.
//  (using ldn-bit words)
// The returned value is a divisor of ldn.
{
    ulong y = bit_rotate_right(x, 1, ldn);
    return  bit_cyclic_match(x, y, ldn) + 1;
}
```

A demo for generating necklaces via the obvious (brute force) search is [FXT: demo bits/necklace-demo.cc]. It is possible to completely avoid the rotation of partial words: let d be a divisor of the word length n. Then the rightmost $(n-1)\,d$ bits of the word obtained as `x^(x>>d)` are zero exactly if the word has period d. Thereby we can use the following function body:

```
    ulong sl = BITS_PER_LONG-ldn;
    for (ulong s=1; s<ldn; ++s)
    {
        ++sl;
        if ( 0==( (x^(x>>s)) << sl ) )  return s;
    }
    return ldn;
```

Testing for periods that are not divisor of the word length can be avoided as follows:

```
    ulong f = tiny_factors_tab[ldn];
    ulong sl = BITS_PER_LONG-ldn;
    for (ulong s=1; s<ldn; ++s)
    {
        ++sl;
        f >>= 1;
        if ( 0==(f&1) )  continue;
        if ( 0==( (x^(x>>s)) << sl ) )  return s;
    }
    return ldn;
```

The table of tiny factors used is shown in section 1.11 on page 23.

The version for `ldn==BITS_PER_LONG` can be optimized similarly:

```
static inline ulong bit_cyclic_period(ulong x)
// Return minimal positive bit-rotation that transforms x into itself.
// (same as bit_cyclic_period(x, BITS_PER_LONG) )
//
// The returned value is a divisor of the word length,
//   i.e. 1,2,4,8,...,BITS_PER_LONG.
{
    ulong r = 1;
    do
    {
        ulong y = bit_rotate_right(x, r);
        if ( x==y )  return r;
        r <<= 1;
    }
    while ( r < BITS_PER_LONG );

    return  r;  // == BITS_PER_LONG
}
```

A related function computes the cyclic distance between two words [FXT: file bits/bitcyclic-dist.h]:

```
inline ulong bit_cyclic_dist(ulong a, ulong b)
// Return minimal bitcount of (t ^ b)
// where t runs through the cyclic rotations.
{
    ulong d = ~0UL;
    ulong t = a;
    do
    {
        ulong z = t ^ b;
        ulong e = bit_count( z );
        if ( e < d )  d = e;
        t = bit_rotate_right(t, 1);
    }
    while ( t!=a );
```

```
    return  d;  // not reached
}
```

The functions [FXT: file bits/bitcyclic-xor.h]

```
static inline ulong bit_cyclic_rxor(ulong x)
{
    return x ^ bit_rotate_right(x, 1);
}
```

and

```
static inline ulong bit_cyclic_lxor(ulong x)
{
    return x ^ bit_rotate_left(x, 1);
}
```

return a word where the number of bits is even. In order two produce a random value with an even number of set bits one can use either variant. If the bit count shall be odd, XOR the value with one[4] afterwards.

Iterated application always ends in a cycle, two examples using 6-bit words are:

```
    .11111
    11.1..
    1.1...  <-- cycle start
    1111..
    1...11
    11..11
    ..1.1.
    ..1111  <-- cycle end
    1.1...

    .111..
    .1..1.
    .11.11  <-- cycle start
    11.11.
    1.11.1  <-- cycle end
    .11.11
```

Zero is a fixed point (a period with cycle length one), a typical sequence using 8-bit words is:

```
    .1..11.1
    1...1.11
    .1..111.
    .11.1.1.
    11.111.1
    ..11..11
    1.1.1.1.
    11111111
    ........  <-- cycle start == cycle end
```

Cyclic shifts of a word produce cyclic shifts of the same cycle of words. A word and its complement produce the same result.

The inverse functions need no rotation at all, the inverse of `bit_cyclic_rxor()` is the inverse Gray code (see section 1.16 on page 33):

```
static inline ulong bit_cyclic_inv_rxor(ulong x)
// Return v so that bit_cyclic_rxor(v) == x.
{
    return  inverse_gray_code(x);
}
```

The argument `x` must have an even number of bits. If this is the case then the lowest bit of the result is zero. The complement of the returned value is also an inverse of `bit_cyclic_rxor()`.

The inverse of `bit_cyclic_lxor()` is the inverse green code (see section 1.16.5 on page 37):

```
static inline ulong bit_cyclic_inv_lxor(ulong x)
// Return v so that bit_cyclic_lxor(v) == x.
{
    return  inverse_green_code(x);
}
```

[4] Actually any value with an odd number of set bits will do for the XOR.

We do not need to mask out the lowest bit because for valid arguments (that have an even number of bits) the high bits of the result are zero. This function can be used to solve the quadratic equation $v^2 + v = x$ in the finite field $GF(2^n)$ when normal bases are used, see page 617.

1.14 Reversing the bits of a word

Reversing the bits of a word can be achieved with successive swaps of bit-blocks as described in section 1.9 on page 20. The functions shown are taken from [FXT: file bits/revbin.h].

The following is a 64-bit version of `revbin`

```
static inline ulong revbin(ulong x)
// Return x with bitsequence reversed
{
    x = bit_swap_1(x);
    x = bit_swap_2(x);
    x = bit_swap_4(x);
    x = bit_swap_8(x);
    x = bit_swap_16(x);
    x = bit_swap_32(x);
    return x;
}
```

For 32-bit machines the `bit_swap_32()` line would have to be deleted.

The steps after `bit_swap_4()` correspond to a byte-reverse operation. This operation is just one assembler instruction for many CPUs ('`bswap`'). The inline assembler with GCC for AMD64 CPUs is given in [FXT: file bits/bitasm-amd64.h]:

```
static inline ulong asm_bswap(ulong x)
{
    asm ("bswap %0" : "=r" (x) : "0" (x));
    return x;
}
```

We use it for byte reversion when available:

```
static inline ulong bswap(ulong x)
// Return word with reversed byte order.
{
#ifdef BITS_USE_ASM
    x = asm_bswap(x);
#else
    x = bit_swap_8(x);
    x = bit_swap_16(x);
#if  BITS_PER_LONG >= 64
    x = bit_swap_32(x);
#endif
#endif // def BITS_USE_ASM
    return x;
}
```

The function actually used for bit reversion is good for both 32 and 64 bit words:

```
static inline ulong revbin(ulong x)
{
    x = bit_swap_1(x);
    x = bit_swap_2(x);
    x = bit_swap_4(x);
    x = bswap(x);
    return x;
}
```

One can generate the masks used in the process:

```
static inline ulong revbin(ulong x)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    while ( s )
    {
```

```
        x = ( (x & m) << s ) ^ ( (x & (~m)) >> s );
        s >>= 1;
        m ^= (m<<s);
    }
    return  x;
}
```

Note that the above function is pretty expensive and it is not even clear whether it beats the obvious, bit-wise algorithm:

```
static inline ulong revbin(ulong x)
{
    ulong r = 0,  ldn = BITS_PER_LONG;
    while ( ldn-- != 0 )
    {
        r <<= 1;
        r += (x&1);
        x >>= 1;
    }
    return  r;
}
```

Therefore the function

```
static inline ulong revbin(ulong x, ulong ldn)
// Return word with the ldn least significant bits
//   (i.e. bit_0 ... bit_{ldn-1})  of x reversed,
//   the other bits are set to zero.
{
    return  revbin(x) >> (BITS_PER_LONG-ldn);
}
```

should only be used when `ldn` is not too small, else replaced by the trivial algorithm.

One can also use table lookups methods so that, for example, eight bits are reversed at a time using a 256-byte table. We give the routine for full words:

```
unsigned char revbin_tab[256]; // reversed 8-bit words
ulong revbin_t(ulong x)
{
    ulong r = 0;
    for (ulong k=0; k<BYTES_PER_LONG; ++k)
    {
        r <<= 8;
        r |= revbin_tab[ x & 255 ];
        x >>= 8;
    }
    return r;
}
```

The routine can be optimized by unrolling in order to avoid all branches:

```
static inline ulong revbin_t(ulong x)
{
    ulong r = 0;
    r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
    r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
    r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
    r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
#if BYTES_PER_LONG > 4
    r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
    r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
    r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
    r <<= 8; r |= revbin_tab[ x & 255 ]; x >>= 8;
#endif
    return r;
}
```

However, reversing the first 2^{30} binary words with this routine takes (on a 64-bit machine) longer than with the routine using the `bit_swap_NN()` calls, see [FXT: demo bits/revbin-tab-demo.cc].

Bit-hackers life would be easier if there was a CPU instruction for the reversion of a binary word.

Generating the bit-reversed words in order

If the bit-reversed words have to be generated in the (reversed) counting order and there is a significantly cheaper way to do the update [FXT: file bits/revbin-upd.h]:

```
static inline ulong revbin_upd(ulong r, ulong h)
// Let n=2**ldn and h=n/2.
// Then, with r == revbin(x, ldn) at entry, return revbin(x+1, ldn)
// Note: routine will hang if called with r the all-ones word
{
    while ( !((r^=h)&h) )  h >>= 1;
    return  r;
}
```

Now assume we want to generate the bit-reversed words of all $N = 2^n$ words smaller than 2^n. The total number of branches with the `while`-loop can be estimated by observing that for half of the updates just one bit changes, for a quarter two bits change, three bits change for one eighth of all updates, and so on. Thereby we have $(1/2 + 2/4 + 3/8 + \ldots + n/2^n)\, N < 2\, N$ branches. Observing that the updates that involve a single bit change occur at every second step allows to avoid half of all branches.

For large vales of N the following method can be significantly faster if a fast routine is available for the computation of the least significant bit in a word. The underlying observation is that for a fixed word size n there are just n different patterns of bit-changes with incrementing. We generate a lookup table of the bit-reversed patterns, `utab[]`, an array of `BITS_PER_LONG` elements:

```
inline void make_revbin_upd_tab(ulong ldn)
// Initialize lookup table used by revbin_tupd()
{
    utab[0] = 1UL<<(ldn-1);
    for (ulong k=1; k<ldn; ++k)  utab[k] = utab[k-1] | (utab[k-1]>>1);
}
```

The change patterns for $n = 5$ start as

```
    pattern  reversed pattern
     ...1     1....
     ..11     11...
     .1.1     1.1..
     .111     111..
     ...1     1....
     ..11     11...
     .1.1     1.1..
     .111     111..
     ...1     1....
     ..11     11...
```

The crucial observation is that the pattern with x set bits is used for the update of k to $k + 1$ when the lowest zero of k is at position $x - 1$:

```
                        used when the lowest
                reversed  zero of k is at index:
    utab[0]=     1....      0
    utab[1]=     11...      1
    utab[2]=     111..      2
    utab[3]=     1111.      3
    utab[4]=     11111      4
```

The update routine can now be implemented as

```
inline ulong revbin_tupd(ulong r, ulong k)
// Let r==revbin(k, ldn) then
// return revbin(k+1, ldn).
// NOTE 1: need to call make_revbin_upd_tab(ldn) before usage
//         where ldn=log_2(n)
// NOTE 2: different argument structure form revbin_upd()
{
    k = lowest_bit_idx(~k);  // lowest zero idx
    r ^= utab[k];
    return r;
}
```

The revbin-update routines are used for the revbin permutation described in section 2.1.

	30 bits	16 bits	8 bits	
Update, bit-wise	1.00	1.00	1.00	`revbin_upd()`
Update, table	0.99	1.08	1.15	`revbin_tupd()`
Full, masks	0.74	0.81	0.86	`revbin()`
Full, 8-bit table	1.77	1.94	2.06	`revbin_t()`
Full32, 8-bit table	0.83	0.90	0.96	`revbin_t_le32()`
Full16, 8-bit table	—	0.54	0.58	`revbin_t_le16()`
Full, generated masks	2.97	3.25	3.45	[page 28]
Full, bit-wise	8.76	5.77	2.50	[page 29]

Figure 1.3: Relative performance of the revbin-update and (full) revbin routines. The timing of the bit-wise update routine is normalized to one. Values in each column should be compared, smaller values correspond to faster routines. A column labeled 'N bits' gives the timing for reversing the N bits on a 64 bit machine.

Relative performance of the different algorithms

The relative performance of the different revbin routines is shown in figure 1.3. As a surprise, the full-word revbin function is consistently faster than both of the update routines. This is mainly due to the fact that the used machine (see appendix A on page 673) has a byte swap instruction. As the performance of table lookups is highly machine dependent your results can be very different.

Alternative techniques for in-order generation

The following loop, due to Brent Lehmann [priv.comm.], also generates the bit-reversed words in succession:

```
ulong n = 32;  // a power of two
ulong p = 0, s = 0, n2 = 2*n;
do
{
    // here: s is the bit-reversed word
    p += 2;
    s ^= n - (n / (p&-p));
}
while ( p<n2 );
```

The revbin-increment is branchless but involves a division which usually is an expensive operation. With a fast bit-scan function the loop should be replaced by

```
do
{
    p += 1;
    s ^= n - (n >> (lowest_bit_idx(p)+1));
}
while ( p<n );
```

A recursive algorithm for the generation of the bit-reversed words in order is given in [FXT: demo bits/revbin-rec-demo.cc]:

```
ulong N;
void revbin_rec(ulong f, ulong n)
{
    // visit( f )
    for (ulong m=N>>1; m>n; m>>=1)  revbin_rec(f+m, m);
}
```

One has to call `revbin_rec(0, 0)` to generate all `N`-bit bit-reversed words.

1.15 Bitwise zip

The bitwise zip (bit-zip) operation moves the lower half bits to even indices and higher half bits to odd indices. A straightforward implementation is

```
ulong bit_zip(ulong a, ulong b)
{
    ulong x = 0;
    ulong m = 1, s = 0;
    for (ulong k=0; k<(BITS_PER_LONG/2); ++k)
    {
        x |= (a & m) << s;
        ++s;
        x |= (b & m) << s;
        m <<= 1;
    }
    return  x;
}
```

Its inverse (bit-unzip) moves even indexed bits to the lower half-word and odd indexed bits to the higher half-word:

```
void bit_unzip(ulong x, ulong &a, ulong &b)
{
    a = 0;  b = 0;
    ulong m = 1, s = 0;
    for (ulong k=0; k<(BITS_PER_LONG/2); ++k)
    {
        a |= (x & m) >> s;
        ++s;
        m <<= 1;
        b |= (x & m) >> s;
        m <<= 1;
    }
}
```

The optimized versions (see [FXT: file bits/bitzip.h]), using ideas similar to those in `revbin()` and `bit_count()`, are

```
static inline ulong bit_zip(ulong x)
{
#if  BITS_PER_LONG == 64
    x = butterfly_16(x);
#endif
    x = butterfly_8(x);
    x = butterfly_4(x);
    x = butterfly_2(x);
    x = butterfly_1(x);
    return  x;
}
```

and

```
static inline ulong bit_unzip(ulong x)
{
    x = butterfly_1(x);
    x = butterfly_2(x);
    x = butterfly_4(x);
    x = butterfly_8(x);
#if  BITS_PER_LONG == 64
    x = butterfly_16(x);
#endif
    return  x;
}
```

Both use the `butterfly_*()`-functions which look like

```
static inline ulong butterfly_4(ulong x)
{
    ulong t, ml, mr, s;
#if  BITS_PER_LONG == 64
    ml = 0x0f000f000f000f00;
#else
    ml = 0x0f000f00;
#endif
    s = 4;
    mr = ml >> s;
    t = ((x & ml) >> s ) | ((x & mr) << s );
```

```
    x = (x & ~(ml | mr)) | t;
    return  x;
}
```

The version given by Torsten Sillke (see `http://www.mathematik.uni-bielefeld.de/~sillke/`)

```
static inline ulong Butterfly4(ulong x)
{
   ulong m = 0x00f000f0;
   return ((x & m) << 4) | ((x >> 4) & m) | (x & ~(0x11*m));
}
```

looks much nicer, but seems to use one more register (4 instead of 3) when compiled.

Equivalent functions that zip/unzip the bits of (the lower half of) two words are:

```
#define  BPLH  (BITS_PER_LONG/2)
static inline ulong bit_zip(ulong x, ulong y)
// two-word version:
// only the lower half of x and y are merged
{
    return  bit_zip( (y<<BPLH) + x );
}
```

and

```
static inline void bit_unzip(ulong t, ulong &x, ulong &y)
// two-word version:
// only the lower half of x and y are filled
{
    t = bit_unzip(t);
    y = t >> BPLH;
    x = t ^ (y<<BPLH);
}
```

1.16 Gray code and parity

The *Gray code* of a binary word can easily be computed by

```
static inline ulong gray_code(ulong x)
{
    return  x ^ (x>>1);
}
```

The inverse is slightly more expensive. Understanding the Gray code as 'bitwise derivative modulo 2' leads to the idea to compute the 'bitwise integrals modulo 2' for the inverse:

```
static inline ulong inverse_gray_code(ulong x)
{
    // VERSION 1 (integration modulo 2):
    ulong h=1, r=0;
    do
    {
        if ( x & 1 )  r^=h;
        x >>= 1;
        h = (h<<1)+1;
    }
    while ( x!=0 );
    return r;
}
```

For n-bit words, n-fold application of the Gray code gives back the original word. Using the symbol G for the Gray code (operator) we have $G^n = \mathrm{id}$, so $G^{n-1} \circ G = \mathrm{id} = G^{-1} \circ G$. That is, applying the Gray code computation $n-1$ times gives the inverse Gray code. Thus we can simplify to

```
    // VERSION 2 (apply graycode BITS_PER_LONG-1 times):
    ulong r = BITS_PER_LONG;
    while ( --r )  x ^= x>>1;
    return x;
```

Applying the Gray code twice is identical to `x ^= x>>2;` and the idea holds for all powers of two. This leads to the most efficient way to compute the inverse Gray code:

```
    // VERSION 3 (use: gray ** BITSPERLONG == id):
    x ^= x>>1;  // gray ** 1
    x ^= x>>2;  // gray ** 2
    x ^= x>>4;  // gray ** 4
    x ^= x>>8;  // gray ** 8
    x ^= x>>16; // gray ** 16
    // here: x = gray**31(input)
    // note: the statements can be reordered at will
#if  BITS_PER_LONG >= 64
    x ^= x>>32;  // for 64bit words
#endif
    return  x;
```

The described implementations of the Gray code are given in [FXT: file bits/graycode.h].

```
          k:        g(k)     g(g(k))     g(2*k)   g(2*k+1)
          0:      .......    .......    .......    ......1
          1:      ......1    ......1    .....11    .....1.
          2:      .....11    .....1.    ....11.    ....11.
          3:      .....1.    .....11    ....1.1    ....1..
          4:      ....11.    ....1.1    ...11..    ...11.1
          5:      ....111    ....1..    ...1111    ...111.
          6:      ....1.1    ....111    ...1.1.    ...1.11
          7:      ....1..    ....11.    ...1..1    ...1...
          8:      ...11..    ...1.1.    ..11...    ..11..1
          9:      ...11.1    ...1.11    ..11.11    ..11.1.
         10:      ...1111    ...1...    ..1111.    ..11111
         11:      ...111.    ...1..1    ..111.1    ..111..
         12:      ...1.1.    ...1111    ..1.1..    ..1.1.1
         13:      ...1.11    ...111.    ..1.111    ..1.11.
         14:      ...1..1    ...11.1    ..1..1.    ..1..11
         15:      ...1...    ...11..    ..1...1    ..1....
         16:      ..11...    ..1.1..    .11....    .11...1
         17:      ..11..1    ..1.1.1    .11..11    .11..1.
         18:      ..11.11    ..1.11.    .11.11.    .11.111
         19:      ..11.1.    ..1.111    .11.1.1    .11.1..
         20:      ..1111.    ..1...1    .1111..    .1111.1
         21:      ..11111    ..1....    .111111    .11111.
         22:      ..111.1    ..1..11    .111.1.    .111.11
         23:      ..111..    ..1..1.    .111..1    .111...
         24:      ..1.1..    ..1111.    .1.1...    .1.1..1
         25:      ..1.1.1    ..11111    .1.1.11    .1.1.1.
         26:      ..1.111    ..111..    .1.111.    .1.1111
         27:      ..1.11.    ..111.1    .1.11.1    .1.11..
         28:      ..1..1.    ..11.11    .1..1..    .1..1.1
         29:      ..1..11    ..11.1.    .1..111    .1..11.
         30:      ..1...1    ..11..1    .1...1.    .1...11
         31:      ..1....    ..11...    .1....1    .1.....
```

Figure 1.4: Gray code, squared Gray code and Gray codes of even and odd values.

Gray codes of consecutive values differ in one bit. Gray codes of Gray codes of consecutive values differ in one or two bits. Gray codes of values that have a difference of a power of two differ two bits. Gray codes of even/odd values have an even/odd number of bits set, respectively. This is demonstrated in [FXT: demo bits/gray2-demo.cc], whose output is given in figure 1.4.

In order two produce a random value with an even/odd number of bits set, set the lowest bit of a random number to zero/one, respectively, and take the Gray code.

1.16.1 The parity of a binary word

The *parity* of a word is its bit-count modulo two. The inverse Gray code of a word contains at each bit position the parity of all bits of the input left from it (including itself). Thereby we use the lowest bit [FXT: file bits/parity.h]:

```
static inline ulong parity(ulong x)
// return 1 if the number of set bits is even, else 0
{
    return  inverse_gray_code(x) & 1;
}
```

Be warned that the parity bit of many CPUs is the complement of the above. With the x86-architecture the parity bit only takes in account the lowest byte, therefore [FXT: file bits/bitasm-i386.h]:

```
static inline ulong asm_parity(ulong x)
{
    x ^= (x>>16);
    x ^= (x>>8);
    asm ("addl  $0, %0  \n"
         "setnp %%al    \n"
         "movzx %%al, %0"
         : "=r" (x) : "0" (x) : "eax");
    return x;
}
```

The equivalent code for the AMD64 CPU is [FXT: file bits/bitasm-amd64.h]:

```
static inline ulong asm_parity(ulong x)
{
    x ^= (x>>32);
    x ^= (x>>16);
    x ^= (x>>8);
    asm ("addq  $0, %0  \n"
         "setnp %%al    \n"
         "movzx %%al, %0"
         : "=r" (x) : "0" (x) : "eax");
    return x;
}
```

The sequence of parities of the binary words,

```
0110100110010110100101100110100110010110011010011...
```

is called the *Thue-Morse sequence* (sequence A010060 of [OEIS]). It appears in various seemingly unrelated contexts, see [2] and section 24.1 on page 633.

1.16.2 Byte-wise Gray code and parity

A byte-wise Gray code can be computed using

```
static inline ulong byte_gray_code(ulong x)
// Return the Gray code of bytes in parallel
{
    return  x ^ ((x & 0xfefefefe)>>1);
}
```

Its inverse is

```
static inline ulong byte_inverse_gray_code(ulong x)
// Return the inverse Gray code of bytes in parallel
{
    x ^= ((x & 0xfefefefeUL)>>1);
    x ^= ((x & 0xfcfcfcfcUL)>>2);
    x ^= ((x & 0xf0f0f0f0UL)>>4);
    return  x;
}
```

Thereby

```
static inline ulong byte_parity(ulong x)
// Return the parities of bytes in parallel
{
    return  byte_inverse_gray_code(x) & 0x01010101UL;
}
```

1.16.3 Incrementing (counting) in Gray code

Let $g(k)$ be the Gray code of a number k. We are interested in efficiently generating $g(k+1)$. Using the observation shown in figure 1.5 on the next page we can implement a fast Gray counter if we we use a spare bit to keep track of the parity of the Gray code word. The following routine does it [FXT: file bits/nextgray.h]:

```
        k:     g(k)      g(2*k)      g(k) p     diff p    set
        0:   .......    .......    ...... .   ...... .    {}
        1:   ......1    .....11    .....1 1   .....+ 1    {0}
        2:   .....11    ....11.    ....11 .   ....+1 .    {0, 1}
        3:   .....1.    ....1.1    ....1. 1   ....1- 1    {1}
        4:   ....11.    ...11..    ...11. .   ...+1. .    {1, 2}
        5:   ....111    ...1111    ...111 1   ...11+ 1    {0, 1, 2}
        6:   ....1.1    ...1.1.    ...1.1 .   ...1-1 .    {0, 2}
        7:   ....1..    ...1..1    ...1.. 1   ...1.- 1    {2}
        8:   ...11..    ..11...    ..11.. .   ..+1.. .    {2, 3}
        9:   ...11.1    ..11.11    ..11.1 1   ..11.+ 1    {0, 2, 3}
       10:   ...1111    ..1111.    ..1111 .   ..11+1 .    {0, 1, 2, 3}
       11:   ...111.    ..111.1    ..111. 1   ..111- 1    {1, 2, 3}
       12:   ...1.1.    ..1.1..    ..1.1. .   ..1-1. .    {1, 3}
       13:   ...1.11    ..1.111    ..1.11 1   ..1.1+ 1    {0, 1, 3}
       14:   ...1..1    ..1..1.    ..1..1 .   ..1.-1 .    {0, 3}
       15:   ...1...    ..1...1    ..1... 1   ..1..- 1    {3}
       16:   ..11...    .11....    .11... .   .+1... .    {3, 4}
       17:   ..11..1    .11..11    .11..1 1   .11..+ 1    {0, 3, 4}
```

Figure 1.5: The Gray code equals the Gray code of doubled value shifted to the right once. Equivalently, we can separate the lowest bit which equals the parity of the other bits. The last column shows that the changes with each increment always happen one position left of the rightmost bit.

```
inline ulong next_gray2(ulong x)
// With input x==gray_code(2*k) the return is gray_code(2*k+2).
// Let x1 be the word x shifted right once
// and i1 its inverse Gray code.
// Let r1 be the return r shifted right once.
// Then r1 = gray_code(i1+1).
// That is, we have a Gray code counter.
// The argument must have an even number of bits.
{
    x ^= 1;
    x ^= (lowest_bit(x) << 1);
    return x;
}
```

To obtain a Gray counter, start with $x = 0$, increment with `x=next_gray2(pg)` and use the words $g = x >> 1$:

```
    ulong x = 0;
    for (ulong k=0; k<n2; ++k)
    {
        ulong g = x>>1;
        x = next_gray2(x);
        // here:  g == gray_code(k);
    }
```

This is shown in [FXT: demo bits/bit-nextgray-demo.cc].

To start at an arbitrary (Gray code) value g compute

```
    x = (g<<1) ^ parity(g)
```

in order to use the statement `x=next_gray2(x)` for later increments.

If one works with a sets whose elements are the set bits in the Gray code then the parity is the set size k modulo two. The increment can then be achieved as follows: If k is even then, if the first element is zero, then remove it, else prepend the element zero. If k is odd then, if the first element equals the second minus one, then remove the second element, else insert at the second position the element equal to the first element plus one. Further, the decrement is obtained by simply swapping the actions for even and odd parity.

If one works with an array that contains the elements of the set it is more convenient to actually do the described operations at the end of the array. This leads to the (loopless) algorithm for subsets in minimal change order that is given in section 5.12 on page 179.

1.16.4 The Golay-Rudin-Shapiro sequence *

The function [FXT: file bits/grsnegative.h]

```
static inline ulong grs_negative_q(ulong x)
{
    return  parity( x & (x>>1) );
}
```

returns one for indices where the *Golay-Rudin-Shapiro sequence* (GRS-sequence) has a negative value. The function returns one for $x =$

```
    3, 6, 11, 12, 13, 15, 19, 22, 24, 25, 26, 30, 35, 38, 43, 44, 45,
    47, 48, 49, 50, 52, 53, 55, 59, 60, 61, 63, 67, 70, 75, 76, 77,
    79, 83, 86, 88, 89, 90, 94, 96, 97, 98, 100, 101, 103, 104, 105,
    106, 110, 115, 118, 120, 121, 122, 126, 131, 134, 139, 140, ...
```

This is sequence A020985 of [OEIS], see also section 24.3.

The sequence can be obtained by starting with a sequence of two ones an in each step appending the left half and the negated right half of the values so far. Using '+' and '-' for plus and minus one, respectively:

```
        ++
        +++-
        +++- ++-+
        +++- ++-+  +++- --+-
        +++- ++-+  +++- --+-  +++- ++-+  ---+ ++-+
        +++- ++-+  +++- --+-  +++- ++-+  ---+ ++-+  +++- ++-+  +++- --+- ...
           ^   ^      ^ ^^ ^      ^   ^  ...
           3,  6,  11,12,13,15, 19, 22, ...
```

The algorithm used is to count the bit-pairs modulo 2. Note that the sequence `[1111]` contains three bit-pairs: `[11..]`, `[.11.]` and `[..11]`. The function proves to be useful in specialized versions of the fast Fourier- and Walsh transform, see section 10.3 on page 308.

1.16.5 The green code

We define the *green code* to be the bit-reversed word of the Gray code of the bit-reversed word. That is,

```
   green_code(x) := revbin(gray_code(revbin(x)))
```

It turns out that the corresponding functions are identical to the Gray code versions up to the reversed shift operations (C-language operators '>>' replaced by '<<'). Thereby, computing the green code is a s easy as [FXT: file bits/greencode.h]:

```
static inline ulong green_code(ulong x)
{
    return  x ^ (x<<1);
}
```

Its inverse is

```
static inline ulong inverse_green_code(ulong x)
{
    // use: green ** BITSPERLONG == id:
    x ^= x<<1;  // green ** 1
    x ^= x<<2;  // green ** 2
    x ^= x<<4;  // green ** 4
    x ^= x<<8;  // green ** 8
    x ^= x<<16; // green ** 16
    // here: x = green**31(input)
    // note: the statements can be reordered at will
#if  BITS_PER_LONG >= 64
    x ^= x<<32;  // for 64bit words
#endif
    return  x;
}
```

Demonstration of Gray and green code and their inverses with different (32-bit) input words:

```
-------------------------------------------------------------
111.1111....1111.................  = 0xef0f0000  == word
1..11...1...1...1.............. = gray_code
..11...1...1...1.............. = green_code
1.11.1.11111.1.1111111111111111 = inverse_gray_code
1.1..1.1.....1.1.............. = inverse_green_code
-------------------------------------------------------------
...1....1111....1111111111111111  = 0x10f0ffff  == word
...11...1...1...1.............. = gray_code
..11...1...1...1...............1 = green_code
...11111.1.11111.1.1.1.1.1.1.1.1 = inverse_gray_code
1111.....1.1.....1.1.1.1.1.1.1.1 = inverse_green_code
-------------------------------------------------------------
......1.........................  = 0x2000000  == word
......11........................ = gray_code
.....11......................... = green_code
......11111111111111111111111111 = inverse_gray_code
1111111......................... = inverse_green_code
-------------------------------------------------------------
111111.11111111111111111111111111  = 0xfdffffff  == word
1.....11........................ = gray_code
.....11.......................1 = green_code
1.1.1..1.1.1.1.1.1.1.1.1.1.1.1 = inverse_gray_code
1.1.1.11.1.1.1.1.1.1.1.1.1.1.1 = inverse_green_code
-------------------------------------------------------------
```

The inverse green code contains at each bit position the parity of all bits of the input right from it, including the bit itself. Especially, the word parity can be found in the highest bit of the inverse green code.

The green code preserves the lowest set bit while the Gray code preserves the highest.

Let G^{-1} and E^{-1} be the inverse Gray- and Green code of X, respectively. Then the bit-wise sum (XOR) of G^{-1} and E^{-1} equals X if the parity of X is zero, else it equals the complement X.

1.17 Bit sequency

Functions concerned with the sequency (number of zero-one transitions) are given in [FXT: file bits/bitsequency.h]. Sequency counting:

```
static inline ulong bit_sequency(ulong x)
{
    return bit_count( gray_code(x) );
}
```

Computation of the minimal binary word with given sequency k:

```
static inline ulong first_sequency(ulong k)
// Return the first (i.e. smallest) word with sequency k,
// e.g.  00..00010101010 (seq 8)
// e.g.  00..00101010101 (seq 9)
// Must have:  1 <= k <= BITS_PER_LONG
{
    return inverse_gray_code( first_comb(k) );
}
```

Computation of the maximal binary word with given sequency:

```
static inline ulong last_sequency(ulong k)
// Return the last (i.e. biggest) word with sequency k.
{
    return inverse_gray_code( last_comb(k) );
}
```

The functions `first_comb(k)` and `last_comb(k)` return a word with k bits set at the low and high end, respectively (see section 1.27 on page 56).

Generation of all words with a given sequency, starting with the smallest, can be achieved with a function that computes the next word with the same sequency:

```
static inline ulong next_sequency(ulong x)
// Return smallest integer with highest bit at greater or equal
// position than the highest bit of x that has the same number
// of zero-one transitions (sequency) as x.
// The value of the lowest bit is conserved.
//
// Zero is returned when there is no further sequence.
{
    x = gray_code(x);
    x = next_colex_comb(x);
    x = inverse_gray_code(x);
    return x;
}
```

The inverse function, returning the previous word with the same sequency, is:

```
static inline ulong prev_sequency(ulong x)
{
    x = gray_code(x);
    x = prev_colex_comb(x);
    x = inverse_gray_code(x);
    return x;
}
```

```
seq=    0        1        2        3        4        5        6
      ......   .....1   ...1.   ...1.1   ..1.1.   .1.1.1   1.1.1.
               ....11   ...11.  ..11.1   .11.1.   11.1.1
               ...111   ...1..  ..1..1   .1..1.   1..1.1
               ..1111   ..111.  ..1.11   .1.11.   1.11.1
               .11111   ..11..  .111.1   .1.1..   1.1..1
               111111   ..1...  .11..1   111.1.   1.1.11
                        .1111.  .11.11   11..1.
                        .111..  .1...1   11.11.
                        .11...  .1..11   11.1..
                        .1....  .1.111   1...1.
                        11111.  1111.1   1..11.
                        1111..  111..1   1..1..
                        111...  111.11   1.111.
                        11....  11...1   1.11..
                        1.....  11..11   1.1...
                                11.111
                                1....1
                                1...11
                                1..111
                                1.1111
```

Figure 1.6: 6-bit words of prescribed sequency as generated by `next_sequency()`. Note that the transition at the lower end is not counted. This is consistent with sequency counting function `bit_sequency()`.

The list of all 6-bit words ordered by sequency is shown in figure 1.6. It was created with the program [FXT: demo bits/bitsequency-demo.cc].

1.18 Powers of the Gray code

The Gray code is a bit-wise linear transform of a binary word. The 2^k-th power of the Gray code of x can be computed as `x ^ (x>>k)`. The e-th power can be computed as the bit-wise sum of the powers corresponding to the bits in the exponent. This motivates [FXT: file bits/graypower.h]:

```
inline ulong gray_pow(ulong x, ulong e)
// Return (gray_code**e)(x)
// gray_pow(x, 1) == gray_code(x)
// gray_pow(x, BITS_PER_LONG-1) == inverse_gray_code(x)
{
    e &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = 1;
    while ( e )
    {
```

```
        if ( e & 1 )  x ^= x >> s;  // gray ** s
        s <<= 1;
        e >>= 1;
    }
    return  x;
}
```

The Gray code $g = [g_0, g_1, \ldots, g_7]$ of a 8-bit binary word $x = [x_0, x_1, \ldots, x_7]$ can be expressed as a matrix multiplication over $GF(2)$ (dots for zeros):

```
   g    =        G          x
  [g0]       [ 11...... ]  [x0]
  [g1]       [ .11..... ]  [x1]
  [g2]       [ ..11.... ]  [x2]
  [g3]  =    [ ...11... ]  [x3]
  [g4]       [ ....11.. ]  [x4]
  [g5]       [ .....11. ]  [x5]
  [g6]       [ ......11 ]  [x6]
  [g7]       [ .......1 ]  [x7]
```

The powers of the Gray code correspond to multiplication with powers of the matrix G:

```
1.......    11......    1.1.....    1111....    1...1...    11..11..    1.1.1.1.    11111111
.1......    .11.....    .1.1....    .1111...    .1...1..    .11..11.    .1.1.1.1    .1111111
..1.....    ..11....    ..1.1...    ..1111..    ..1...1.    ..11..11    ..1.1.1.    ..111111
...1....    ...11...    ...1.1..    ...1111.    ...1...1    ...11..1    ...1.1.1    ...11111
....1...    ....11..    ....1.1.    ....1111    ....1...    ....11..    ....1.1.    ....1111
.....1..    .....11.    .....1.1    .....111    .....1..    .....11.    .....1.1    .....111
......1.    ......11    ......1.    ......11    ......1.    ......11    ......1.    ......11
.......1    .......1    .......1    .......1    .......1    .......1    .......1    .......1
G**0=id     G**1=G      G**2        G**3        G**4        G**5        G**6        G**7=G**(-1)
```

The powers of the inverse Gray code for N-bit words (where N is a power of two) can be computed by the relation $G^e\, G^{N-e} = G^N = \mathrm{id}$.

```
inline ulong inverse_gray_pow(ulong x, ulong e)
// Return (inverse_gray_code**(e))(x)
//   == (gray_code**(-e))(x)
// inverse_gray_pow(x, 1) == inverse_gray_code(x)
// inverse_gray_pow(x, BITS_PER_LONG-1) == gray_code(x)
{
    return  gray_pow(x, -e);
}
```

The matrices corresponding to the powers of the green code are:

```
1.......    1.......    1.......    1.......    1.......    1.......    1.......    1.......
.1......    11......    .1......    11......    .1......    11......    .1......    11......
..1.....    .11.....    1.1.....    111.....    ..1.....    .11.....    1.1.....    111.....
...1....    ..11....    .1.1....    1111....    ...1....    ..11....    .1.1....    1111....
....1...    ...11...    ..1.1...    .1111...    1...1...    1..11...    1.1.1...    11111...
.....1..    ....11..    ...1.1..    ..1111..    .1...1..    11..11..    .1.1.1..    111111..
......1.    .....11.    ....1.1.    ...1111.    ..1...1.    .11...11    1.1.1.1.    1111111.
.......1    ......11    .....1.1    ....1111    ...1...1    ..11..11    .1.1.1.1    11111111
E**0=id     E**1=E      E**2        E**3        E**4        E**5        E**6        E**7=E**(-1)
```

We just have to reverse the shift operator in the functions:

```
inline ulong green_pow(ulong x, ulong e)
// Return (green_code**e)(x)
{
    e &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = 1;
    while ( e )
    {
        if ( e & 1 )  x ^= x << s;  // green ** s
        s <<= 1;
        e >>= 1;
    }
    return  x;
}
```

The inverse function is

```
inline ulong inverse_green_pow(ulong x, ulong e)
// Return (inverse_green_code**(e))(x)
{
    return  green_pow(x, -e);
}
```

1.19 Invertible transforms on words

The functions presented in this section are 'transforms' on binary words that are invertible. The names are chosen as 'some code', emphasizing the result of the transforms, similar to the convention used with the name 'Gray code'. The functions are given in [FXT: file bits/bittransforms.h].

Consider (*blue code*)

```
inline ulong blue_code(ulong a)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
    while ( s )
    {
        a ^= ( (a&m) >> s );
        s >>= 1;
        m ^= (m>>s);
    }
    return  a;
}
```

and (*yellow code*)

```
inline ulong yellow_code(ulong a)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    while ( s )
    {
        a ^= ( (a&m) << s );
        s >>= 1;
        m ^= (m<<s);
    }
    return  a;
}
```

Both involve a computational work $\sim \log_2(b)$ where b is the number of bits per word (`BITS_PER_LONG`). The `blue_code` can be used as a fast implementation for the composition of a binary polynomial with $x+1$, see page 584.

The output of the program [FXT: demo bits/bittransforms-blue-demo.cc] is shown in figure 1.7 on the following page. The parity of $B(a)$ is equal to the lowest bit of a. Up to the $a = 47$ the bit-count varies by ± 1 between successive values of $B(a)$, the transition $B(47) \to B(48)$ changes the bit-count by 3. The sequence of the indices a where the bit-count changes by more than one is 47, 51, 59, 67, 75, 79, 175, 179, 187, 195, 203, 207, 291, 299, 339, 347, 419, 427, 467, 475, 531, 539, ...

The yellow code might be a good candidate for 'randomization' of binary words. The blue code maps any range $[0 \dots 2^k - 1]$ onto itself. Both the blue code and the yellow code are self-inverse.

The transforms (*red code*)

```
inline ulong red_code(ulong a)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    while ( s )
    {
        ulong u = a & m;
        ulong v = a ^ u;
        a = v ^ (u<<s);
        a ^= (v>>s);
        s >>= 1;
        m ^= (m<<s);
    }
    return  a;
}
```

and (*cyan code*)

```
inline ulong cyan_code(ulong a)
{
    ulong s = BITS_PER_LONG >> 1;
```

```
 0:  k=......  0  b=......  0*  y=................................  0
 1:  k=.....1  1  b=.....1  1*  y=11111111111111111111111111111111 32
 2:  k=....1.  1  b=....11  2   y=1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1. 16
 3:  k=....11  2  b=....1.  1   y=.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1 16
 4:  k=...1..  1  b=...1.1  2   y=11..11..11..11..11..11..11..11.. 16
 5:  k=...1.1  2  b=...1..  1   y=..11..11..11..11..11..11..11..11 16
 6:  k=...11.  2  b=...11.  2*  y=.11..11..11..11..11..11..11..11. 16
 7:  k=...111  3  b=...111  3*  y=1..11..11..11..11..11..11..11..1 16
 8:  k=..1...  1  b=..1111  4   y=1...1...1...1...1...1...1...1...  8
 9:  k=..1..1  2  b=..111.  3   y=.111.111.111.111.111.111.111.111 24
10:  k=..1.1.  2  b=..11..  2   y=..1...1...1...1...1...1...1...1.  8
11:  k=..1.11  3  b=..11.1  3   y=11.111.111.111.111.111.111.111.1 24
12:  k=..11..  2  b=..1.1.  2   y=.1...1...1...1...1...1...1...1..  8
13:  k=..11.1  3  b=..1.11  3   y=1.111.111.111.111.111.111.111.11 24
14:  k=..111.  3  b=..1..1  2   y=111.111.111.111.111.111.111.111. 24
15:  k=..1111  4  b=..1...  1   y=...1...1...1...1...1...1...1...1  8
16:  k=.1....  1  b=.1...1  2   y=1111....1111....1111....1111.... 16
17:  k=.1...1  2  b=.1....  1   y=....1111....1111....1111....1111 16
18:  k=.1..1.  2  b=.1..1.  2*  y=.1.11.1..1.11.1..1.11.1..1.11.1. 16
19:  k=.1..11  3  b=.1..11  3*  y=1.1..1.11.1..1.11.1..1.11.1..1.1 16
20:  k=.1.1..  2  b=.1.1..  2*  y=..1111....1111....1111....1111.. 16
21:  k=.1.1.1  3  b=.1.1.1  3*  y=11....1111....1111....1111....11 16
22:  k=.1.11.  3  b=.1.111  4   y=1..1.11.1..1.11.1..1.11.1..1.11. 16
23:  k=.1.111  4  b=.1.11.  3   y=.11.1..1.11.1..1.11.1..1.11.1..1 16
24:  k=.11...  2  b=.1111.  4   y=.1111....1111....1111....1111... 16
25:  k=.11..1  3  b=.11111  5   y=1....1111....1111....1111....111 16
26:  k=.11.1.  3  b=.111.1  4   y=11.1..1.11.1..1.11.1..1.11.1..1. 16
27:  k=.11.11  4  b=.111..  3   y=..1.11.1..1.11.1..1.11.1..1.11.1 16
28:  k=.111..  3  b=.11.11  4   y=1.11.1..1.11.1..1.11.1..1.11.1.. 16
29:  k=.111.1  4  b=.11.1.  3   y=.1..1.11.1..1.11.1..1.11.1..1.11 16
30:  k=.1111.  4  b=.11...  2   y=...1111....1111....1111....1111. 16
31:  k=.11111  5  b=.11..1  3   y=111....1111....1111....1111....1 16
```

Figure 1.7: Blue and yellow transforms. Bit-counts are shown at the right of each column. Fixed points are marked with asterisks.

```
 0:  r=................................  0  c=................................  0
 1:  r=1...............................  1  c=11111111111111111111111111111111 32
 2:  r=11..............................  2  c=.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1 16
 3:  r=.1..............................  1  c=1.1.1.1.1.1.1.1.1.1.1.1.1.1.1. 16
 4:  r=1.1.............................  2  c=..11..11..11..11..11..11..11..11 16
 5:  r=..1.............................  1  c=11..11..11..11..11..11..11..11.. 16
 6:  r=.11.............................  2  c=.11..11..11..11..11..11..11..11. 16
 7:  r=111.............................  3  c=1..11..11..11..11..11..11..11..1 16
 8:  r=1111............................  4  c=...1...1...1...1...1...1...1...1  8
 9:  r=.111............................  3  c=111.111.111.111.111.111.111.111. 24
10:  r=..11............................  2  c=.1...1...1...1...1...1...1...1..  8
11:  r=1.11............................  3  c=1.111.111.111.111.111.111.111.11 24
12:  r=.1.1............................  2  c=..1...1...1...1...1...1...1...1.  8
13:  r=11.1............................  3  c=11.111.111.111.111.111.111.111.1 24
14:  r=1..1............................  2  c=.111.111.111.111.111.111.111.111 24
15:  r=...1............................  1  c=1...1...1...1...1...1...1...1...  8
16:  r=1...1...........................  2  c=....1111....1111....1111....1111 16
17:  r=....1...........................  1  c=1111....1111....1111....1111.... 16
18:  r=.1..1...........................  2  c=.1.11.1..1.11.1..1.11.1..1.11.1. 16
19:  r=11..1...........................  3  c=1.1..1.11.1..1.11.1..1.11.1..1.1 16
20:  r=..1.1...........................  2  c=..1111....1111....1111....1111.. 16
21:  r=1.1.1...........................  3  c=11....1111....1111....1111....11 16
22:  r=111.1...........................  4  c=.11.1..1.11.1..1.11.1..1.11.1..1 16
23:  r=.11.1...........................  3  c=1..1.11.1..1.11.1..1.11.1..1.11. 16
24:  r=.1111...........................  4  c=...1111....1111....1111....1111. 16
25:  r=11111...........................  5  c=111....1111....1111....1111....1 16
26:  r=1.111...........................  4  c=.1..1.11.1..1.11.1..1.11.1..1.11 16
27:  r=..111...........................  3  c=1.11.1..1.11.1..1.11.1..1.11.1.. 16
28:  r=11.11...........................  4  c=..1.11.1..1.11.1..1.11.1..1.11.1 16
29:  r=.1.11...........................  3  c=11.1..1.11.1..1.11.1..1.11.1..1. 16
30:  r=...11...........................  2  c=.1111....1111....1111....1111... 16
31:  r=1..11...........................  3  c=1....1111....1111....1111....111 16
```

Figure 1.8: Red and cyan transforms. Bit-counts are shown at the right of each column.

```
    ulong m = ~0UL << s;
    while ( s )
    {
        ulong u = a & m;
        ulong v = a ^ u;
        a = v ^ (u>>s);
        a ^= (v<<s);
        s >>= 1;
        m ^= (m>>s);
    }
    return  a;
}
```

are demonstrated in [FXT: demo bits/bittransforms-red-demo.cc] whose output is shown in figure 1.8 on the preceding page.

Fixed points of the blue code

```
       0 = .....  :    .........  =     0
       1 = .....1 :    .........1 =     1
       2 = ....1. :    .......11. =     6
       3 = ....11 :    .......111 =     7
       4 = ...1.. :    .....1.1.. =    20
       5 = ...1.1 :    .....1..1. =    18
       6 = ...11. :    .....1.1.1 =    21
       7 = ...111 :    .....1..11 =    19
       8 = ..1... :    ...1111... =   120
       9 = ..1..1 :    ...11.11.. =   108
      10 = ..1.1. :    ...111111. =   126
      11 = ..1.11 :    ...11.1.1. =   106
      12 = ..11.. :    ...1111..1 =   121
      13 = ..11.1 :    ...11.11.1 =   109
      14 = ..111. :    ...1111111 =   127
      15 = ..1111 :    ...11.1.11 =   107
      16 = .1.... :    .1...1.... =   272
      17 = .1...1 :    .1.11.1... =   360
      18 = .1..1. :    .1.....1.. =   260
      19 = .1..11 :    .1.11111.. =   380
      20 = .1.1.. :    .1...1.11. =   278
      21 = .1.1.1 :    .1.11.111. =   366
      22 = .1.11. :    .1......1. =   258
      23 = .1.111 :    .1.1111.1. =   378
      24 = .11... :    .1...1...1 =   273
      25 = .11..1 :    .1.11.1..1 =   361
      26 = .11.1. :    .1.....1.1 =   261
      27 = .11.11 :    .1.11111.1 =   381
      28 = .111.. :    .1...1.111 =   279
      29 = .111.1 :    .1.11.1111 =   367
      30 = .1111. :    .1......11 =   259
      31 = .11111 :    .1.1111.11 =   379
```

Figure 1.9: The first fixed points of the blue code. The highest bit of all fixed points lies at an even index. There are $2^{n/2}$ fixed points with highest bit at index n.

The sequence of fixed points of the blue code is

```
0, 1, 6, 7, 18, 19, 20, 21, 106, 107, 108, 109, 120, 121, 126, 127, 258, 259, 260, 261, 272, ...
```

If f is a fixed point then f XOR 1 is also a fixed point. Further, $2\,(f \text{ XOR } (2\,f))$ is a fixed point. These facts can be cast into a function that returns a unique fixed point for each argument [FXT: file bits/blue-fixed-points.h]:

```
ulong blue_fixed_point(ulong s)
{
    if ( 0==s )  return 0;
    ulong f = 1;
    while ( s>1 )
    {
        f ^= (f<<1);
        f <<= 1;
        f |= (s&1);
        s >>= 1;
    }
    return f;
}
```

The output for the first few arguments is shown in figure 1.9 on the preceding page. Note that the fixed points are not in ascending order. The list was created by the program [FXT: demo bits/bittransforms-blue-fp-demo.cc]. In terms of binary polynomials (see chapter 22 on page 569), if $f(x)$ is a fixed point (that is, $f(x) = f(x+1)$), then both $(x^2+x)\,f(x)$ and $1 + (x^2+x)\,f(x)$ are fixed points. The function `blue_fixed_point()` repeatedly multiplies by x^2+x and adds one if the corresponding bit of the argument is set.

The inverse function uses the fact that polynomial division by $x+1$ can be achieved with the inverse green code (see section 1.16.5 on page 37) if the polynomial is divisible by $x+1$:

```
ulong blue_fixed_point_idx(ulong f)
// Inverse of blue_fixed_point()
{
    ulong s = 1;
    while ( f )
    {
        s <<= 1;
        s ^= (f & 1);
        f >>= 1;
        f = inverse_green_code(f);  // == bitpol_div(f, 3);
    }
    return s >> 1;
}
```

Relations between the transforms

We write B for the blue code (transform), Y for the yellow code and r for bit-reversal (the `revbin`-function). Then B and Y are connected by the relations

$$B = Y\,r\,Y \tag{1.1a}$$
$$Y = B\,r\,B \tag{1.1b}$$
$$B = r\,Y\,r \tag{1.1c}$$
$$Y = r\,B\,r \tag{1.1d}$$
$$r = Y\,B\,Y \tag{1.1e}$$
$$r = B\,Y\,B \tag{1.1f}$$

As said, B and Y are self-inverse:

$$B^{-1} = B \qquad B\,B = \mathrm{id} \tag{1.2a}$$
$$Y^{-1} = Y \qquad Y\,Y = \mathrm{id} \tag{1.2b}$$

The red code and the cyan code are not involutions ('square roots of identity') but third roots of identity (Using R for the red code, C for the cyan code):

$$R\,R\,R = \mathrm{id} \qquad R^{-1} = R\,R = C \tag{1.3a}$$
$$C\,C\,C = \mathrm{id} \qquad C^{-1} = C\,C = R \tag{1.3b}$$
$$R\,C = C\,R = \mathrm{id} \tag{1.3c}$$

By construction

$$R = r\,B \tag{1.4a}$$
$$C = r\,Y \tag{1.4b}$$

Similar inter-relations as for B and Y hold for R and C:

$$R = C\,r\,C \tag{1.5a}$$
$$C = R\,r\,R \tag{1.5b}$$
$$R = r\,C\,r \tag{1.5c}$$
$$C = r\,R\,r \tag{1.5d}$$
$$R = R\,C\,R \tag{1.5e}$$
$$C = C\,R\,C \tag{1.5f}$$

One has

$$r = Y\,R = R\,B = B\,C = C\,Y \tag{1.6}$$

Further

$$B = RY = Y\,C = R\,B\,R = C\,B\,C \tag{1.7a}$$
$$Y = C\,B = B\,R = R\,Y\,R = C\,Y\,C \tag{1.7b}$$

$$R = B\,Y = B\,C\,B = Y\,C\,Y \tag{1.7c}$$
$$C = Y\,B = B\,R\,B = Y\,R\,Y \tag{1.7d}$$

and

$$\mathrm{id} = B\,Y\,C = R\,Y\,B \tag{1.8a}$$
$$\mathrm{id} = C\,B\,Y = B\,R\,Y \tag{1.8b}$$
$$\mathrm{id} = Y\,C\,B = Y\,B\,R \tag{1.8c}$$

The following multiplication table lists $Z = Y\,X$. The R in the third column of the second row says that $r\,B = R$. The letter i is used for identity (id). An asterisk says that $X\,Y = Y\,X$.

	i	r	B	Y	R	C
i	i*	r*	B*	Y*	R*	C*
r	r*	i*	R	C	B	Y
B	B*	C	i*	R	Y	r
Y	Y*	R	C	i*	r	B
R	R*	Y	r	B	C*	i*
C	C*	B	Y	r	i*	R*

Relations to Gray code and green code

Write g for the Gray code, then:

$$g\,B\,g\,B = \mathrm{id} \tag{1.9a}$$
$$g\,B\,g = B \tag{1.9b}$$
$$g^{-1}\,B\,g^{-1} = B \tag{1.9c}$$
$$g\,B = B\,g^{-1} \tag{1.9d}$$

Let S_k be the operator that rotates a word by k bits (bit zero is moved to position k, use [FXT: `bit_rotate_sgn()` in bits/bitrotate.h]) then

$$Y\,S_{+1}\,Y = g \tag{1.10a}$$
$$Y\,S_{-1}\,Y = g^{-1} \tag{1.10b}$$
$$Y\,S_k\,Y = g^k \tag{1.10c}$$

'Shift in the frequency domain is derivative in time domain'.

Similarly, let e be the green code operator, then

$$B\,S_{+1}\,B = e^{-1} \tag{1.11a}$$
$$B\,S_{-1}\,B = e \tag{1.11b}$$
$$B\,S_{k}\,B = e^{-k} \tag{1.11c}$$

More transforms by symbolic powering

The idea of powering a transform (as done for the Gray code in section 1.18 on page 39) can be applied to the 'color'-transforms as exemplified for the blue code:

```
inline ulong blue_xcode(ulong a, ulong x)
{
    x &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
    while ( s )
    {
        if ( x & 1 )  a ^= ( (a&m) >> s );
        x >>= 1;
        s >>= 1;
        m ^= (m>>s);
    }
    return  a;
}
```

The result is *not* the power of the blue code which would be pretty boring as $B\,B = \mathrm{id}$. Instead the transform (and the equivalents for Y, R and C, see [FXT: file bits/bitxtransforms.h]) are more interesting: All relations between the transforms are still valid, if the symbolic exponent is identical with all terms. For example, we had $B\,B = \mathrm{id}$, now $B^x\,B^x = \mathrm{id}$ is true for all x (there are essentially `BITS_PER_LONG` different x). Similarly, $C\,C = R$ now has to be $C^x\,C^x = R^x$. That is, we have `BITS_PER_LONG` different versions of our four transforms that share their properties with the 'simple' versions. Among them `BITS_PER_LONG` transforms B^x and Y^x that are involutions and C^x and R^x that are third roots of the identity: $C^x\,C^x\,C^x = R^x\,R^x\,R^x = \mathrm{id}$.

While not powers of the simple versions, we still have $B^0 = Y^0 = R^0 = C^0 = \mathrm{id}$. Further, let e be the 'exponent' of all ones and Z be any of the transforms, then $Z^e = Z$, Writing '+' for the XOR operation, then $Z^x\,Z^y = Z^{x+y}$ and so $Z^x\,Z^y = Z$ whenever $x + y = e$.

The building blocks of the transforms

Consider the following transforms on two-bit words where addition is bit-wise (that is, XOR):

$$\mathrm{id}_2\ v = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \tag{1.12a}$$
$$r_2\,v = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} b \\ a \end{bmatrix} \tag{1.12b}$$
$$B_2\,v = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a+b \\ b \end{bmatrix} \tag{1.12c}$$
$$Y_2\,v = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ a+b \end{bmatrix} \tag{1.12d}$$
$$R_2\,v = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} b \\ a+b \end{bmatrix} \tag{1.12e}$$
$$C_2\,v = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a+b \\ a \end{bmatrix} \tag{1.12f}$$

It can easily be verified that for these the same relations hold as for id, r, B, Y, R, C. In fact the 'color-transforms', bit-reversion and (trivially) id are the transforms obtained by the repeated Kronecker-products of the matrices. The transforms are linear over $GF(2)$:

$$Z(\alpha\, a + \beta\, b) \quad = \quad \alpha\, Z(a) + \beta\, Z(b) \tag{1.13}$$

The corresponding version of the bit-reversal is [FXT: `xrevbin()` in bits/revbin.h]:

```
inline ulong xrevbin(ulong a, ulong x)
{
    x &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    while ( s )
    {
        if ( x & 1 )  a = ( (a & m) << s ) ^ ( (a & (~m)) >> s );
        x >>= 1;
        s >>= 1;
        m ^= (m<<s);
    }
    return  a;
}
```

Then, for example $R^x = r^x\, B^x$ (see relation 1.4a on page 44).

The yellow code the bit-wise Reed-Muller transform described in section 10.10 on page 328. The symbolic powering is equivalent to selecting individual levels of the transform.

1.20 Moves of the Hilbert curve

An efficient algorithm that computes the direction of the n-th move of the Hilbert curve is based on the parity of the number of threes in the radix-4 representation of n (see section 24.9.1 on page 654).

Let d_x and d_y correspond to the moves at step n in the Hilbert curve. Then $d_x, d_y \in \{-1, 0, +1\}$ and exactly one of them is zero. Thereby for both $p := d_x + d_y$ and $m := d_x - d_y$ we have $p, m \in \{-1, +1\}$.

The following function computes p and returns $0, 1$ if $p = -1, +1$, respectively [FXT: file bits/hilbert.h]:

```
inline ulong hilbert_p(ulong t)
// Let dx,dy be the horizontal,vertical move
// with step t of the Hilbert curve.
// Return  zero if (dx+dy)==-1, else one (then: (dx+dy)==+1).
// Algorithm: count number of threes in radix 4
{
    ulong d = (t & 0x5555555555555555UL) & ((t & 0xaaaaaaaaaaaaaaaaUL) >> 1);
    return  parity( d );
}
```

The function can be slightly optimized as follows (64-bit version only):

```
inline ulong hilbert_p(ulong t)
{
    t &= ((t & 0xaaaaaaaaaaaaaaaaUL) >> 1);
    t ^= t>>2;
    t ^= t>>4;
    t ^= t>>8;
    t ^= t>>16;
    t ^= t>>32;
    return  t & 1;
}
```

The value of m can be computed as:

```
inline ulong hilbert_m(ulong t)
// Let dx,dy be the horizontal,vertical move
// with step t of the Hilbert curve.
// Return  zero if (dx-dy)==-1, else one (then: (dx-dy)==+1).
{
    return  hilbert_p( -t );
}
```

It remains to merge the values of p and m into a two-bit value d that encodes the direction of the move:

```
inline ulong hilbert_dir(ulong t)
// Return d encoding the following move with the Hilbert curve.
//
// d \in {0,1,2,3} as follows:
//   d : direction
//   0 : right (+x:  dx=+1, dy= 0)
//   1 : down  (-y:  dx= 0, dy=-1)
//   2 : up    (+y:  dx= 0, dy=+1)
//   3 : left  (-x:  dx=-1, dy= 0)
{
    ulong p = hilbert_p(t);
    ulong m = hilbert_m(t);
    ulong d = p ^ (m<<1);
    return  d;
}
```

To print the value of d symbolically, one can use the C++ statement `cout << ("v><^")[d];`.

The turn u between steps can be computed as

```
inline int hilbert_turn(ulong t)
// Return the turn (left or right) with the steps
//   t and t-1 of the Hilbert curve.
// Returned value is
//   0 for no turn
//  +1 for right turn
//  -1 for left turn
{
    ulong d1 = hilbert_dir(t);
    ulong d2 = hilbert_dir(t-1);
    d1 ^= (d1>>1);
    d2 ^= (d2>>1);
    ulong u = d1 - d2;
    // at this point, symbolically:  cout << ("+.-0+.-")[ u + 3 ];
    if ( 0==u )  return 0;
    if ( (long)u<0 )  u += 4;
    return  (1==u ? +1 : -1);
}
```

To print the value of u symbolically, one can use `cout << ("-0+")[d+1];`.

The program [FXT: demo bits/hilbert-moves-demo.cc] prints the values p, m followed by the direction and turn:

```
    dx+dy: ++-+++-+++---+++-+++-+++---+++-+++-+++---+---+---+---++++-
    dx-dy: +---+++-+++-+++-+++---+---+---+++---+---+---+---+++---+---+--
      dir: >^<^^>v>^>vv<v>>^>v>>^<^>^<<v<^^^>v>>^<^>^<<v<^<<v>vv<^<v<^^>^<
     turn: 0--+0++--++0+--0-++-0--++--0-++00++-0--++--0-++-0--+0++--++0+--
```

1.21 The Z-order

A 2-dimensional space-filling curve that traverses all points in each quadrant before it enters the next can be obtained by the *Z-order*. The conversion between a linear parameter to a pair coordinates achieved by separating the bits at the even and odd indices. The simple routine is [FXT: file bits/zorder.h]:

```
inline void lin2zorder(ulong t, ulong &x, ulong &y)  { bit_unzip(t, x, y); }
```

The routine `bit_zip()` is described in section 1.15 on page 32. The inverse is

```
inline ulong zorder2lin(ulong x, ulong y)  { return bit_zip(x, y); }
```

From any coordinate pair the next pair can be computed with the following (constant amortized time) routine:

```
inline void zorder_next(ulong &x, ulong &y)
{
    ulong b = 1;
    do
    {
        x ^= b;  b &= ~x;
```

```
        y ^= b;  b &= ~y;
        b <<= 1;
    }
    while ( b );
}
```

The previous pair is obtained similarly:

```
inline void zorder_prev(ulong &x, ulong &y)
{
    ulong b = 1;
    do
    {
        x ^= b;  b &= x;
        y ^= b;  b &= y;
        b <<= 1;
    }
    while ( b );
}
```

The routines are written in a way that generalizes trivially to more dimensions:

```
inline void zorder3d_next(ulong &x, ulong &y, ulong &z)
{
    ulong b = 1;
    do
    {
        x ^= b;  b &= ~x;
        y ^= b;  b &= ~y;
        z ^= b;  b &= ~z;
        b <<= 1;
    }
    while ( b );
}
```

Unlike the Hilbert curve there are steps where the curve advances more than one unit.

1.22 Scanning for zero bytes

The function

```
static inline ulong contains_zero_byte(ulong x)
{
#if  BITS_PER_LONG == 32
    return ((x-0x01010101UL)^x) & (~x) & 0x80808080UL;
#endif
#if  BITS_PER_LONG == 64
    return ((x-0x0101010101010101UL) ^ x) & (~x) & 0x8080808080808080UL;
#endif
}
```

from [FXT: file bits/zerobyte.h] determines if any sub-byte of the argument is zero. It returns zero when x contains no zero-byte and nonzero when it does. The idea is to subtract 1 from each of the bytes and then look for bytes where the borrow propagated all the way to the most significant bit. In order to scan for other values than zero (e.g. 0xa5) one can use `contains_zero_byte( x ^ 0xa5a5a5a5UL )`.

Using the simplified version

```
    return ((x-0x01010101UL) ^ x) & 0x80808080UL;
```

gives false alarms when a byte equals 0x80. For one byte (in hex, omitting prefixes '0x'):

```
      x-01          = 80-01   = 7f
     (x-01)^x       = 7f ^ 80 = ff
    ((x-01)^x) & 80 = ff & 80 = 80   != 0
```

For strings where the high bit of every byte is known to be zero (for example ASCII-strings) the simple version can be used.

The function [FXT: file aux1/bytescan.cc]

```
ulong long_strlen(const char *str)
// Return length of string starting at str.
```

```
{
    ulong x;
    const char *p = str;

    // Alignment: scan bytes up to word boundary:
    while ( (ulong)p % BYTES_PER_LONG )
    {
        if ( 0 == *p )  return  (ulong)(p-str);
        ++p;
    }

    x = *(ulong *)p;
    while ( ! contains_zero_byte(x) )
    {
        p += BYTES_PER_LONG;
        x = *(ulong *)p;
    }

    // now a zero byte is somewhere in x:
    while ( 0 != *p )  { ++p; }

    return  (ulong)(p-str);
}
```

may be a win for very long strings and word sizes of 64 or more bits.

1.23 Manipulation of colors

In the following it is assumed that the type `uint` (unsigned integer) is at least 32 bit. In this section this data type is exclusively used as a container for three color channels that are assumed to be 8 bit each and lie at the lower end of the word. The functions do not depend on how the channels are ordered.

The following functions are obviously candidates for your CPU's SIMD-extensions (if it has any). However, having the functionality in a platform independent manner is sometimes convenient [FXT: file bits/colormix.h]. The functions offer a performance that is sufficiently fast for many practical purposes: A software rendering program that uses them operates at a not too small fraction of memory bandwidth when all of environment mapping, texture mapping and translucent objects are shown with (very) simple scenes.

Scaling a color by factor between zero and one, given as integer value:

```
static inline uint color01(uint c, ulong v)
// Return color with each channel scaled by v
// 0 <= v <= (1<<16)  corresponding to 0.0 ... 1.0
{
    uint t;
    t = c & 0xff00ff00;  // must include alpha channel bits ...
    c ^= t;  // ... because they must be removed here
    t *= v;
    t >>= 24; t <<= 8;
    v >>= 8;
    c *= v;
    c >>= 8;
    c &= 0xff00ff;
    return  c | t;
}
```

The function can alternatively be implemented as

```
    uint t;
    v >>= 8;
    t  = (((0xff00ff & c) * v) >> 8 ) & 0xff00ff;
    t |= (((0x00ff00 & c) * v) >> 8 ) & 0x00ff00;
    return  t;
```

It is used in the computation of the weighted average of two colors:

```
static inline uint color_mix(uint c1, uint c2, ulong v)
// Return channel-wise average of colors:  (1.0-v)*c1 + v*c2
// 0 <= v <= (1<<16)  corresponding to 0.0 ... 1.0
{
    ulong w = ((ulong)1<<16)-v;
```

```
    c1 = color01(c1, w);
    c2 = color01(c2, v);
    return  c1 + c2;  // no overflow in color channels
}
```

Computing the channel-wise average of two colors:

```
static inline uint color_mix_50(uint c1, uint c2)
// Return channel-wise average of colors c1 and c2
// Shortcut for the special case (50% transparency)
//   of color_mix(c1, c2, "0.5")
// The least significant bits are ignored
{
    return  ((c1 & 0xfefefe) + (c2 & 0xfefefe)) >> 1;  // 50% c1
}
```

Channel-wise mixing with higher weight (75 percent) of the first color:

```
static inline uint color_mix_75(uint c1, uint c2)
// The least significant bits are ignored
{
    return  color_mix_50(c1, color_mix_50(c1, c2));  // 75% c1
}
```

Saturated addition of color channels:

```
static inline uint color_sum(uint c1, uint c2)
// Return channel-wise saturated sum of colors c1 and c2.
// The least significant bits are ignored.
{
    uint s = color_mix_50(c1, c2);
    return  color_sum_adjust(s);
}
```

The function uses

```
static inline uint color_sum_adjust(uint s)
// Set color channel to max (0xff) iff an overflow occurred
//  (that is, leftmost bit in channel is set)
{
    uint m = s & 0x808080;  // 1000 0000 // overflow bits
    s ^= m;
    m >>= 7;    // 0000 0001
    m *= 0xff;  // 1111 1111  // optimized to (m<<8)-m by gcc
    return  (s << 1) | m;
}
```

Channel-wise product of two colors, corresponding to the appearance of an object of color c_1 illuminated by a light of color c_2:

```
static inline uint color_mult(uint c1, uint c2)
{
    uint t = ((c1 & 0xff) * (c2 & 0xff)) >> 8;
    c1 >>= 8;  c2 >>= 8;
    t |= ((c1 & 0xff) * (c2 & 0xff)) & 0xff00;
    c1 &= 0xff00;  c2 >>= 8;
    t |= ((c1 * c2) & 0xff0000);
    return  t;
}
```

When one does not want to discard the lowest channel bits (for example, because numerous such operations appear in a row) a more 'perfect' version is required [FXT: file bits/colormixp.h]:

```
static inline uint perfect_color_mix_50(uint c1, uint c2)
// Return channel-wise average of colors c1 and c2
{
//    uint t = (c1 | c2) & 0x010101; // lowest channels bits in any arg
    uint t = (c1 & c2) & 0x010101; // lowest channels bits in both args
    return  color_mix_50(c1, c2) + t;
}

static inline uint perfect_color_sum(uint c1, uint c2)
{
    uint srb = (c1 & 0xff00ff) + (c2 & 0xff00ff) + 0x010001;
    uint mrb = srb & 0x01000100;
    srb ^= mrb;
    uint sg = (c1 & 0xff00) + (c2 & 0xff00) + 0x0100;
    uint mg = (sg & 0x010000);
```

```
    sg ^= mg;
    uint m = (mrb | mg) >> 1;  // 1000 0000 // overflow bits
    m |= (m >> 1);  // 1100 0000
    m |= (m >> 2);  // 1111 0000
    m |= (m >> 4);  // 1111 1111
    return  srb | sg | m;
}
```

Note that the last two functions are overkill for most practical purposes.

1.24 2-adic inverse and square root

The 2-adic inverse can be computed using an iteration (see section 15.1 on page 399) with quadratic convergence. The number to be inverted has to be odd.

```
inline ulong inv2adic(ulong x)
// Return inverse modulo 2**BITS_PER_LONG
// x must be odd
// The number of correct bits is doubled with each step
// ==> loop is executed prop. log_2(BITS_PER_LONG) times
// precision is 3, 6, 12, 24, 48, 96, ... bits (or better)
{
    if ( 0==(x&1) )  return 0;  // not invertible
    ulong i = x;  // correct to three bits at least
    ulong p;
    do
    {
        p = i * x;
        i *= (2UL - p);
    }
    while ( p!=1 );
    return  i;
}
```

Let m be the modulus (a power of two), then the computed value i is the inverse of x modulo m: $i \equiv x^{-1} \bmod m$. It can be used for the so-called *exact division*: To compute the quotient a/x for a number a that is known to be divisible by x, simply multiply by i. This works because $a = b\,x$ (a is divisible by x), so $a\,i \equiv b\,x\,i \equiv b \bmod m$.

With the inverse square root we choose the start value to match $\lfloor d/2 \rfloor + 1$ as that guarantees four bits of initial precision. Moreover, we get control to which of the two possible values the inverse square root is finally reached. The argument modulo 8 has to be equal to one.

```
inline ulong invsqrt2adic(ulong d)
// Return inverse square root modulo 2**BITS_PER_LONG
// Must have:  d==1 mod 8
// The number of correct bits is doubled with each step
// ==> loop is executed prop. log_2(BITS_PER_LONG) times
// precision is 4, 8, 16, 32, 64, ... bits (or better)
{
    if ( 1 != (d&7) )  return 0;  // no inverse sqrt
    // start value: if d == ****10001 ==> x := ****1001
    ulong x = (d >> 1) | 1;
    ulong p, y;
    do
    {
        y = x;
        p = (3 - d * y * y);
        x = (y * p) >> 1;
    }
    while ( x!=y );
    return  x;
}
```

The square root can be obtained by a final multiplication with d:

```
inline ulong sqrt2adic(ulong d)
// Return square root modulo 2**BITS_PER_LONG
// Must have: d==1 mod 8  or  d==4 mod 32,  d==16 mod 128
//   ... d==4**k mod 4**(k+3)
```

```
// Result undefined if condition does not hold
{
    if ( 0==d )  return  0;
    ulong s = 0;
    while ( 0==(d&1) )  { d >>= 1; ++s; }
    d *= invsqrt2adic(d);
    d <<= (s>>1);
    return   d;
}
```

Note that the 2-adic square root is something completely different from the integer square root.

```
       x   = ...............................1  =    1
       inv = ...............................1
      sqrt = ...............................1

       x   = 11111111111111111111111111111111  =   -1
       inv = 11111111111111111111111111111111

       x   = ............................1.  =    2

       x   = 11111111111111111111111111111111.  =   -2

       x   = ..i.i.i.i.i.i.i.i.i.i.i.i.i.i.i.11  =    3
       inv = i.i.i.i.i.i.i.i.i.i.i.i.i.i.i.i.11

       x   = 11111111111111111111111111111111.1  =   -3
       inv = .1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1

       x   = ............................1..  =    4
      sqrt = ............................1.

       x   = 11111111111111111111111111111111..  =   -4

       x   = ii..ii..ii..ii..ii..ii..ii..ii..1.1  =    5
       inv = ii..ii..ii..ii..ii..ii..ii..ii..1.1

       x   = 11111111111111111111111111111111.11  =   -5
       inv = ..11..11..11..11..11..11..11..11..11

       x   = ............................11.  =    6

       x   = 11111111111111111111111111111111.1.  =   -6

       x   = i..ii..ii..ii..ii..ii..ii..ii..ii.111  =    7
       inv = i.ii.ii.ii.ii.ii.ii.ii.ii.ii.ii.111

       x   = 11111111111111111111111111111111..1  =   -7
       inv = .1..1.1..1..1..1..1..1..1..1..1..1.1
      sqrt = 1..111..1..11...11.....1.11.1.1

       x   = ............................1...  =    8

       x   = 11111111111111111111111111111111...  =   -8

       x   = ..i..i..i..i..i..i..i..i..i..1..1  =    9
       inv = .ii..ii..ii..ii..ii..ii..ii..ii..i.1
      sqrt = 1111111111111111111111111111111.1
```

Figure 1.10: Examples of the 2-adic inverse and square root. Where no inverse or square root is given, it does not exist.

The described functions are given in [FXT: file bits/bit2adic.h]. The output of the demo [FXT: demo bits/bit2adic-demo.cc] is shown in figure 1.10.

For further information on 2-adic (more generally p-adic) numbers see [DEK2], [Num] and also [68].

1.25 Radix −2 representations

Item 128 of [HAKMEM] gives a surprisingly simple algorithm to obtain the radix −2 representation of a binary number. The digits are again zero and one so the result can be represented as binary number [FXT: file bits/negbin.h]:

```
inline ulong bin2neg(ulong x)
// binary --> radix(-2)
{
    ulong m = 0xaaaaaaaaUL; // 32 bit
    x += m;
    x ^= m;
    return  x;
}
```

```
 k :   bin(k)  m=negbin(k)  g=gray(m)   bin(g)
 0 : ......    ......     ......      0 <= 0
 1 : .....1    .....1     .....1      1 <= 1
 2 : ....1.    ....11     ....1.      5
 3 : ....11    ....111    ....1.      4
 4 : ...1..    ....1..    ....11      2
 5 : ...1.1    ...1.1     ...111      3 <= 5
 6 : ...11.    ..11.1     ..1.11     19
 7 : ...111    ..11.11    ..1.1.     18
 8 : ..1...    ..11...    ..1.1.     20
 9 : ..1..1    ..11..1    ..1.1.1    21
10 : ..1.1.    ..1111.    ..1...1    17
11 : ..1.11    ..11111    ..1....    16
12 : ..11..    ..111..    ..1..1.    14
13 : ..11.1    ..111.1    ..1..11    15
14 : ..111.    ..1..1.    ..11.11     7
15 : ..1111    ..1..11    ..11.1.     6
16 : .1....    ..1....    ..11...     8
17 : .1...1    ..1...1    ..11..1     9
18 : .1..1.    ..1.11.    ..111.1    13
19 : .1..11    ..1.111    ..111..    12
20 : .1.1..    ..1.1..    ..1111.    10
21 : .1.1.1    ..1.1.1    ..11111    11 <= 21
22 : .1.11.    11.1.1.    1.11111    75
23 : .1.111    11.1.11    1.1111.    74
24 : .11...    11.1...    1.111..    76
25 : .11..1    11.1..1    1.111.1    77
26 : .11.1.    11.111.    1.11..1    73
27 : .11.11    11.1111    1.11...    72
28 : .111..    11.11..    1.11.1.    70
29 : .111.1    11.11.1    1.11.11    71
30 : .1111.    11...1.    1.1..11    79
31 : .11111    11...11    1.1..1.    78
```

Figure 1.11: Radix -2 representations and their Gray codes. Lines ending in '`<=N`' indicate that all values less or equal N occur in the last column up to that point.

An example:

```
14 -->  ..1..1. == 16 - 2 == (-2)^4 + (-2)^1
```

The inverse routine is obtained by executing the inverse of the two steps in reversed order:

```
inline ulong neg2bin(ulong x)
// radix(-2) --> binary
// inverse of bin2neg()
{
    ulong m = 0xaaaaaaaaUL;
    x ^= m;
    x -= m;
    return  x;
}
```

Figure 1.11 shows the output of the demo [FXT: demo bits/negbin-demo.cc]. The sequence of Gray codes of the radix -2 representation is a Gray code for

$$k \;=\; 0,\, 1,\, 5,\, 21,\, 85,\, 341,\, \ldots,\, (4^x - 1)/3$$

The sequence of fixed points of the conversion starts as

```
 0 : ..........
 1 : .........1
 4 : ........1.
 5 : ........1.1
16 : .....1....
17 : .....1...1
20 : .....1.1..
21 : .....1.1.1
64 : ...1......
65 : ...1.....1
68 : ...1....1.
69 : ...1....1.1
80 : ...1.1....
81 : ...1.1...1
84 : ...1.1.1..
85 : ...1.1.1.1
```

This the *Moser – De Bruijn sequence*, sequence A000695 of [OEIS]. A related power series is

$$\prod_{k=0}^{\infty}\left(1+x^{4^k}\right) \quad = \quad 1+x+x^4+x^5+x^{16}+x^{17}+x^{20}+x^{21}+x^{64}+x^{65}+x^{68}+\ldots \tag{1.14}$$

The k-th ($k \geq 0$) fixed point can be obtained moving all bits of the binary representation of k to position $2\,x$ where $x \geq 0$ is the index of the bit under consideration:

```
inline ulong negbin_fixed_point(ulong k)
// Return the k-th (0<=k) fixed point of bin2neg()
{
    ulong f = 0;
    for (ulong m=1, s=0;  m!=0;  m<<=1, ++s)  f ^= ( (k&m)<<s );
    return  f;
}
```

The sequence of radix -2 representations of 0, 1, 2, ... interpreted as binary numbers is entry A005351 of [OEIS]:

```
0,1,6,7,4,5,26,27,24,25,30,31,28,29,18,19,16,17,22,23,20,21,106,107,104,105,110,111, ...
```

The corresponding sequence for the negative numbers -1, -2, -3, ... is entry A005352:

```
3,2,13,12,15,14,9,8,11,10,53,52,55,54,49,48,51,50,61,60,63,62,57,56,59,58,37,36,39,38, ...
```

More information about 'non-standard' representations of numbers can be found in [DEK2].

1.26 A sparse signed binary representation

```
 0:    ......    ......     0 =
 1:    .....1    .....P     1 =  +1
 2:    ....1.    ....P.     2 =  +2
 3:    ....11    ...P.M     3 =  +4 -1
 4:    ...1..    ...P..     4 =  +4
 5:    ...1.1    ...P.P     5 =  +4 +1
 6:    ...11.    ..P.M.     6 =  +8 -2
 7:    ...111    ..P..M     7 =  +8 -1
 8:    ..1...    ..P...     8 =  +8
 9:    ..1..1    ..P..P     9 =  +8 +1
10:    ..1.1.    ..P.P.    10 =  +8 +2
11:    ..1.11    ..P.M.M   11 =  +16 -4 -1
12:    ..11..    ..P.M.    12 =  +16 -4
13:    ..11.1    ..P.M.P   13 =  +16 -4 +1
14:    ..111.    ..P..M.   14 =  +16 -2
15:    ..1111    ..P...M   15 =  +16 -1
16:    .1....    ..P....   16 =  +16
17:    .1...1    ..P...P   17 =  +16 +1
18:    .1..1.    ..P..P.   18 =  +16 +2
19:    .1..11    ..P.P.M   19 =  +16 +4 -1
20:    .1.1..    ..P.P.    20 =  +16 +4
21:    .1.1.1    ..P.P.P   21 =  +16 +4 +1
22:    .1.11.    .P.M.M.   22 =  +32 -8 -2
23:    .1.111    .P.M..M   23 =  +32 -8 -1
24:    .11...    .P.M...   24 =  +32 -8
25:    .11..1    .P.M..P   25 =  +32 -8 +1
26:    .11.1.    .P.M.P.   26 =  +32 -8 +2
27:    .11.11    .P..M.M   27 =  +32 -4 -1
28:    .111..    .P..M.    28 =  +32 -4
29:    .111.1    .P..M.P   29 =  +32 -4 +1
30:    .1111.    .P...M.   30 =  +32 -2
31:    .11111    .P....M   31 =  +32 -1
32:    1.....    .P.....   32 =  +32
```

Figure 1.12: M.Paul's sparse signed binary representations. The symbols '`P`' and '`N`' are used for $+1$ and -1, respectively.

An algorithm to compute a representation of a number x as

$$x \quad = \quad \sum_{k=0}^{\infty} s_k \cdot 2^k \quad \text{where} \quad s_k \in \{-1,\, 0,\, +1\} \tag{1.15}$$

and to consecutive digits s_k, s_{k+1} are never simultaneously nonzero is given in [84]. Figure 1.12 on the previous page gives the representation of several small numbers, it is the output of [FXT: demo bits/bin2paul-demo.cc].

We can convert the binary representation of x into a pair of binary numbers that correspond to the positive and negative digits [FXT: file bits/bin2paul.h]:

```
inline void bin2paul(ulong x, ulong &pp, ulong &pm)
// Compute M.Paul's signed binary representation of x:
// the unique representation of x as
//   x=\sum_{k}{d_k*2^k} where d_j \in {-1,0,+1}
//   and no two adjacent digits d_j, d_{j+1} are both nonzero.
// pp has bits j set where d_j==+1
// pm has bits j set where d_j==-1
// Thereby:  x = pp - pm
{
    ulong xh = x >> 1;  // x/2
    ulong x3 = x + xh;  // 3*x/2
    ulong c = xh ^ x3;
    pp = x3 & c;
    pm = xh & c;
}
```

The cited source refers to the the described representation as 'M.Paul's system', therefore the name of the routine. Converting back to binary is trivial:

```
inline ulong paul2bin(ulong pp, ulong pm)
// Inverse of bin2paul()
{
    return ( pp - pm );
}
```

A Gray code for the signed binary words is given in section 5.19 on page 204.

The sequence of values where the negative part is zero is [0, 1, 2, 4, 5, 8, 9, 10, 16, ...], sequence A003714 of [OEIS] is shown in figure 1.13. The numbers are called the *Fibbinary numbers*.

```
 0:    .......    .......     0 =
 1:    .......1   .......P    1 =  +1
 2:    ......1.   ......P.    2 =  +2
 4:    .....1..   .....P..    4 =  +4
 5:    .....1.1   .....P.P    5 =  +4 +1
 8:    ....1...   ....P...    8 =  +8
 9:    ....1..1   ....P..P    9 =  +8 +1
10:    ....1.1.   ....P.P.   10 =  +8 +2
16:    ...1....   ...P....   16 =  +16
17:    ...1...1   ...P...P   17 =  +16 +1
18:    ...1..1.   ...P..P.   18 =  +16 +2
20:    ...1.1..   ...P.P..   20 =  +16 +4
21:    ...1.1.1   ...P.P.P   21 =  +16 +4 +1
32:    ..1.....   ..P.....   32 =  +32
33:    ..1....1   ..P....P   33 =  +32 +1
34:    ..1...1.   ..P...P.   34 =  +32 +2
36:    ..1..1..   ..P..P..   36 =  +32 +4
37:    ..1..1.1   ..P..P.P   37 =  +32 +4 +1
40:    ..1.1...   ..P.P...   40 =  +32 +8
41:    ..1.1..1   ..P.P..P   41 =  +32 +8 +1
42:    ..1.1.1.   ..P.P.P.   42 =  +32 +8 +2
64:    .1......   .P......   64 =  +64
```

Figure 1.13: The numbers whose negative part in M.Paul's representation is zero, the so-called *Fibbinary numbers*.

1.27 Generating bit combinations

The following functions are taken from [FXT: file bits/bitcombcolex.h] and [FXT: file bits/bitcomblex.h]. Given a binary word with k bits set the following routine computes the binary word that is the next combination of k bits in colex order.

The underlying mechanism is to determine the lowest block of ones and move its highest bit one position up. The rest of the block is moved to the low end of the word.

```
inline ulong next_colex_comb(ulong x)
// Return smallest integer greater than x with the same number of bits set.
//
// colex order: (5,3);
//  0  1  2   ..111
//  0  1  3   .1.11
//  0  2  3   .11.1
//  1  2  3   .111.
//  0  1  4   1..11
//  0  2  4   1.1.1
//  1  2  4   1.11.
//  0  3  4   11..1
//  1  3  4   11.1.
//  2  3  4   111..
//
//  Examples:
//    000001 -> 000010 -> 000100 -> 001000 -> 010000 -> 100000
//    000011 -> 000101 -> 000110 -> 001001 -> 001010 -> 001100 -> 010001 -> ...
//    000111 -> 001011 -> 001101 -> 001110 -> 010011 -> 010101 -> 010110 -> ...
//
//  Special cases:
//    0 -> 0
//    all bits on the high side (i.e. last combination) -> 0
//
{
    ulong r = x & -x; // lowest set bit
    x += r;           // replace lowest block by a one left to it

    if ( 0==l )  return 0;  // input was last comb

    ulong l = x & -x; // first zero beyond low block
    l -= r;           // low block

    while ( 0==(l&1) )  { l >>= 1; }  // move block to low end of word
    return  x | (l>>1);  // need one bit less of low block
}
```

One might consider replacing the while-loop by a bit scan and shift combination.

Moving backwards goes like

```
inline ulong prev_colex_comb(ulong x)
// inverse of next_colex_comb()
{
    x = next_colex_comb( ~x);
    if ( 0!=x )  x = ~x;
    return  x;
}
```

Figure 1.14 on the following page shows the relation to lex order enumeration is that the ones in lex-order(k,n) behave like the zeros in reversed colex-order(n-k,n). That is, in order to obtain the next lex-combination let the zeros move to the lower end in the same manner as the ones go to the higher end in next colex-combination:

```
inline ulong next_lex_comb(ulong x)
{
    x = revbin(~x);
    x = next_colex_comb(x);
    if ( 0!=x )  x = revbin(~x);
    return  x;
}
```

The bit-reversal routine `revbin()` is shown in section 1.14 on page 28. Moving backwards goes like:

```
inline ulong prev_lex_comb(ulong x)
// Inverse of next_lex_comb()
{
    x = revbin(x);
    x = next_colex_comb(x);
    x = revbin(x);
    return  x;
}
```

The first and last combination for both colex- and lex- order are

```
inline ulong first_comb(ulong k)
```

```
        Lex( n = 5,  k = 3 )                Colex( n = 5,  k = 2 )

      forward order:                      forward order:
        [  0  1  2 ]  ..111  #  0           [  0  1 ]  ...11  #  0
        [  0  1  3 ]  .1.11  #  1           [  0  2 ]  ..1.1  #  1
        [  0  1  4 ]  1..11  #  2           [  1  2 ]  ..11.  #  2
        [  0  2  3 ]  .11.1  #  3           [  0  3 ]  .1..1  #  3
        [  0  2  4 ]  1.1.1  #  4           [  1  3 ]  .1.1.  #  4
        [  0  3  4 ]  11..1  #  5           [  2  3 ]  .11..  #  5
        [  1  2  3 ]  .111.  #  6           [  0  4 ]  1...1  #  6
        [  1  2  4 ]  1.11.  #  7           [  1  4 ]  1..1.  #  7
        [  1  3  4 ]  11.1.  #  8           [  2  4 ]  1.1..  #  8
        [  2  3  4 ]  111..  #  9           [  3  4 ]  11...  #  9

      reverse order:                      reverse order:
        [  2  3  4 ]  111..  #  9           [  3  4 ]  11...  #  9
        [  1  3  4 ]  11.1.  #  8           [  2  4 ]  1.1..  #  8
        [  1  2  4 ]  1.11.  #  7           [  1  4 ]  1..1.  #  7
        [  1  2  3 ]  .111.  #  6           [  0  4 ]  1...1  #  6
        [  0  3  4 ]  11..1  #  5           [  2  3 ]  .11..  #  5
        [  0  2  4 ]  1.1.1  #  4           [  1  3 ]  .1.1.  #  4
        [  0  2  3 ]  .11.1  #  3           [  0  3 ]  .1..1  #  3
        [  0  1  4 ]  1..11  #  2           [  1  2 ]  ..11.  #  2
        [  0  1  3 ]  .1.11  #  1           [  0  2 ]  ..1.1  #  1
        [  0  1  2 ]  ..111  #  0           [  0  1 ]  ...11  #  0
```

Figure 1.14: Relation between combinations in lex and colex order when generated forward and backward. Bit-reversing and complementing one order gives the other.

```
// Return the first combination of (i.e. smallest word with) k bits,
// i.e.  00..001111..1 (k low bits set)
// Must have:  0 <= k <= BITS_PER_LONG
{
    return  ~0UL >> ( BITS_PER_LONG - k );
}
```

and

```
inline ulong last_comb(ulong k, ulong n=BITS_PER_LONG)
// return the last combination of (biggest n-bit word with) k bits
// i.e.  1111..100..00 (k high bits set)
// Must have:  0 <= k <= n <= BITS_PER_LONG
{
    return  first_comb(k) << (n - k);
}
```

The colex-combinations in reversed order are identical to the bit-reversed lex-combinations, thereby:

```
inline ulong first_rev_comb(ulong k, ulong n=BITS_PER_LONG)
{
    return  last_comb(k, n);
}

inline ulong last_rev_comb(ulong k)
{
    return  first_comb(k);
}

inline ulong next_lexrev_comb(ulong x)
// Return next bit-reversed lex-order combination.
{
    return  prev_colex_comb(x);
}

inline ulong prev_lexrev_comb(ulong x)
// Return previous bit-reversed lex-order combination.
{
    return  next_colex_comb( x );
}
```

These are often the functions of choice when fast generation of bit-combinations is required as they allow to restrict the pattern to a certain length without any overhead.

A variant of the presented (colex-) algorithm appears in [HAKMEM] as item 175. The variant used here

avoids the division of the HAKMEM-version and is given at `http://www.caam.rice.edu/~dougm/` by Doug Moore and Glenn Rhoads `http://remus.rutgers.edu/~rhoads/`.

1.28 Generating bit subsets

In order to generate all bit-subsets of a binary word one can use the sparse counting idea shown in section 1.8 on page 19. The utility class is [FXT: `class bit_subset` in bits/bitsubset.h]:

```
class bit_subset
// generate all all subsets of bits of a given word
{
public:
    ulong u_, v_;
public:
    bit_subset(ulong v) : u_(0), v_(v) { ; }
    ~bit_subset() { ; }
    ulong current()  const { return u_; }
    ulong next()  { u_ = (u_ - v_) & v_;  return u_; }
    ulong prev()  { u_ = (u_ - 1 ) & v_;  return u_; }
};
```

With the word `[...11.1.]` the following sequence of words is produced by subsequent `next()`-calls:

```
   .....1.
   ....1..
   ....1.1.
   ...1....
   ...1...1.
   ...11...
   ...11.1.
   ........
```

A block of ones at the right will result in the binary counting sequence, see [FXT: demo bits/bitsubset-demo.cc].

1.29 Binary words in lexicographic order

Next and previous word in lex-oder

The (bit-reversed) binary words in lexicographic order can be generated by successive calls to the following function [FXT: file bits/bitlex.h]:

```
static inline ulong next_lexrev(ulong x)
// Return next word in (reversed) lex order.
{
    ulong x0 = x & -x;
    if ( 1==x0 )
    {
        x ^= x0;
        x0 = x & -x;
        x0 ^= (x0>>1);
    }
    else  x0 >>= 1;

    x ^= x0;
    return  x;
}
```

The bit-reversed representation was chosen because the isolation of the lowest bit is often cheaper than the same operation on the highest bit. The routine was derived from the code in section 5.11 on page 177. Starting with a one-bit word at position $n-1$ one generates the 2^n bit-subsets of length n:

```
    n==4:
     1...
     11..
     111.
     1111
     11.1
     1.1.
     1.11
     1..1
     .1..
```

```
.11.
.111
.1.1
..1.
..11
...1
```

A similar function allows to go backward:

```
static inline ulong prev_lexrev(ulong x)
// Return previous word in (reversed) lex order.
{
    ulong x0 = x & -x;
    if ( x & (x0<<1) )  x ^= x0;
    else
    {
        x0 ^= (x0<<1);
        x ^= x0;
        x |= 1;
    }
    return x;
}
```

Starting with zero one obtains, by repeated calls to `prev_lexrev()`, a sequence of words that just before the 2^n-th call has visited every word of length n:

```
 n==4:
 0:  ....  =  0 *
 1:  ...1  =  1 *
 2:  ..11  =  3
 3:  ..1.  =  2
 4:  .1.1  =  5
 5:  .111  =  7
 6:  .11.  =  6 *
 7:  .1..  =  4
 8:  1..1  =  9
 9:  1.11  = 11
10:  1.1.  = 10 *
11:  1111  = 13
12:  111.  = 14
13:  11..  = 12
14:  1...  =  8
15:  1.1.  = 10 *
```

The sequence 1, 3, 2, 5, 7, 6, 4, 9, ... is entry A108918 of [OEIS]. It turns out to be useful for a special version of fast Walsh transforms, this is described in section 10.5 on page 316.

The asterisks mark fixed points of the sequence.

The sequence of fixed points

The sequence of fixed points is: 0, 1, 6, 10, 18, 34, 60, 66, 92, 108, 116, 130, 156, 172, 180, 204, 212, 228, 258, 284, 300, 308, 332, 340, 356, 396, 404, 420, 452, 514, 540, 556, ... This is sequence A079471 of [OEIS]. Their values as bit patterns are shown in figure 1.15 on the facing page. The crucial observation is that a word is a fixed point exactly if (it equals zero or) its bit-count equals 2^j where j is the index of the lowest set bit.

Now we can find out whether x is a fixed point of the sequence by the following function:

```
static inline bool is_lexrev_fixed_point(ulong x)
// Return whether x is a fixed point in the prev_lexrev() - sequence
{
    if ( x & 1 )
    {
        if ( 1==x )  return  true;
        else         return  false;
    }
    else
    {
        ulong w = bit_count(x);
        if ( w != (w & -w) )  return  false;
        if ( 0==x )  return  true;
        return  0 != ( (x & -x) & w );
    }
}
```

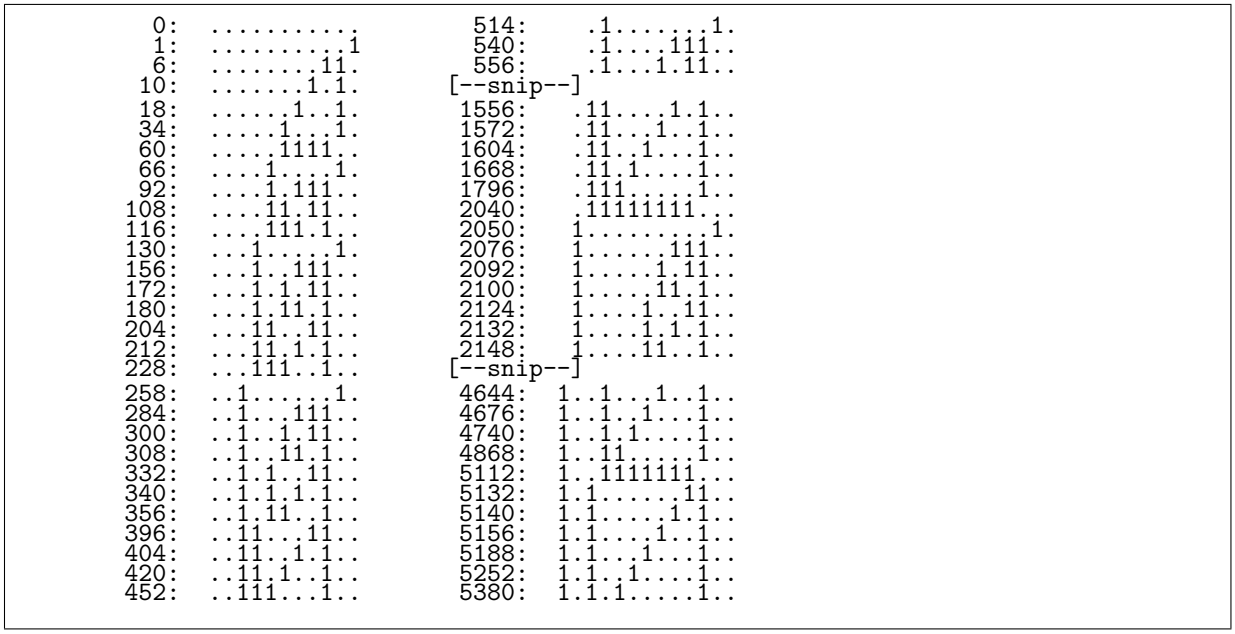


Figure 1.15: Fixed points of the binary to lex-rev conversion.

Conversion between binary- and lex-ordered words

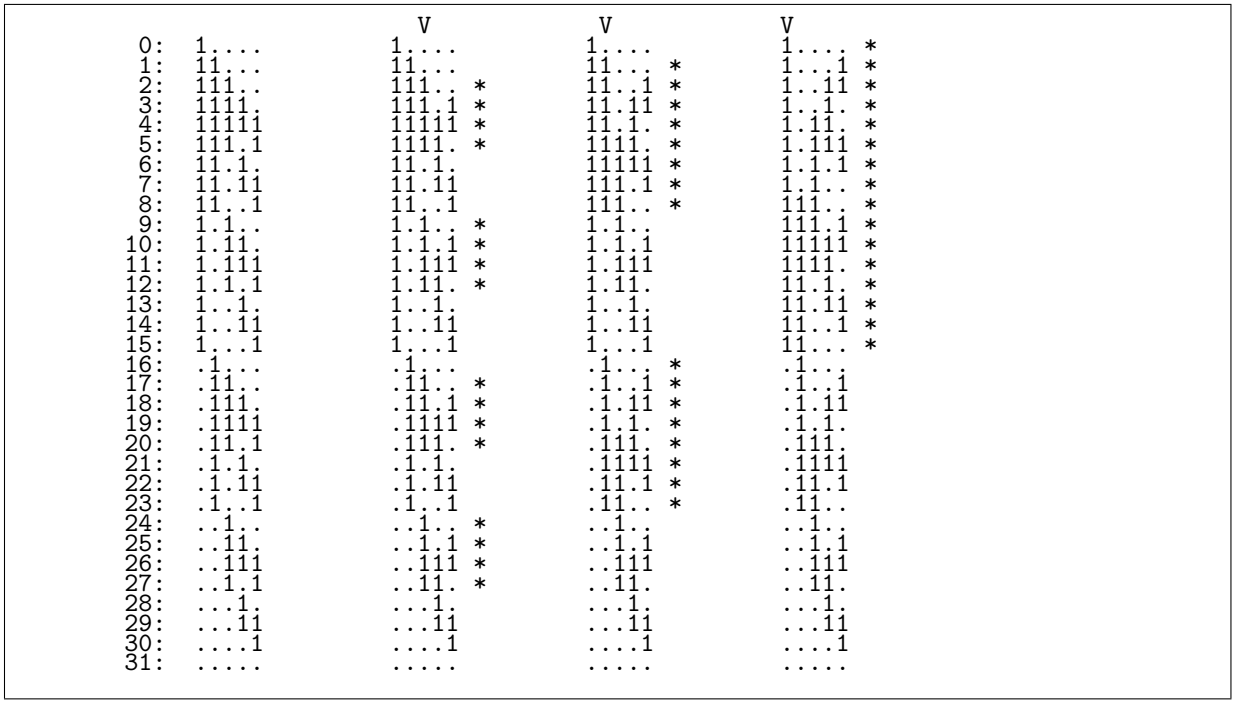
A little contemplation on the structure of the binary words in lex-order leads to the routine that allows random access to the k-th lex-rev word:

```
inline ulong negidx2lexrev(ulong k)
//   k:  negidx2revlex(k)
//   0:  .....
//   1:  ....1
//   2:  ...11
//   3:  ...1.
//   4:  ..1.1
//   5:  ..111
//   6:  ..11.
//   7:  ..1..
//   8:  .1..1
{
    ulong z = 0;
    ulong h = highest_bit(k);
    while ( k )
    {
        while ( 0==(h&k) )  h >>= 1;
        z ^= h;
        ++k;
        k &= h - 1;
    }
    return  z;
}
```

and its inverse

```
inline ulong lexrev2negidx(ulong x)
{
    ulong r = 0;
    while ( x )
    {
        ulong h = highest_bit(x);
        r ^= h;
        x = next_lexrev(x);
        h = highest_bit(x);
        x ^= h;
    }
    return  r;
}
```

Note that this function is computationally more expensive than the 'forward' version.



```
                          V             V             V
         0:  1....        1....         1....         1.... *
         1:  11...        11...         11... *       1...1 *
         2:  111..        111.. *       11..1 *       1..11 *
         3:  1111.        111.1 *       11.11 *       1..1. *
         4:  11111        11111 *       11.1. *       1.11. *
         5:  111.1        1111. *       1111. *       1.111 *
         6:  11.1.        11.1.         11111 *       1.1.1 *
         7:  11.11        11.11         111.1 *       1.1.. *
         8:  11..1        11..1         111.. *       111.. *
         9:  1.1..        1.1.. *       1.1..         111.1 *
        10:  1.11.        1.1.1 *       1.1.1         11111 *
        11:  1.111        1.111 *       1.111         1111. *
        12:  1.1.1        1.11. *       1.11.         11.1. *
        13:  1..1.        1..1.         1..1.         11.11 *
        14:  1..11        1..11         1..11         11..1 *
        15:  1...1        1...1         1...1         11... *
        16:  .1...        .1...         .1... *       .1...
        17:  .11..        .11.. *       .1..1 *       .1..1
        18:  .111.        .11.1 *       .1.11 *       .1.11
        19:  .1111        .1111 *       .1.1. *       .1.1.
        20:  .11.1        .111. *       .111. *       .111.
        21:  .1.1.        .1.1.         .1111 *       .1111
        22:  .1.11        .1.11         .11.1 *       .11.1
        23:  .1..1        .1..1         .11.. *       .11..
        24:  ..1..        ..1.. *       ..1..         ..1..
        25:  ..11.        ..1.1 *       ..1.1         ..1.1
        26:  ..111        ..111 *       ..111         ..111
        27:  ..1.1        ..11. *       ..11.         ..11.
        28:  ...1.        ...1.         ...1.         ...1.
        29:  ...11        ...11         ...11         ...11
        30:  ....1        ....1         ....1         ....1
        31:  .....        .....         .....         .....
```

Figure 1.16: Conversion of the list of bit-reversed binary words in lexicographic order (left column) to reversed Gray order (right column). From bit 2 to the highest bit, find consecutive runs where the bit in question is set. Then in each run (symbol '`*`'), reverse the sublist from the second to the last word. The columns just processed are marked by the symbol '`V`'.

Recursive generation and relation to a Taylor series

The following function generates the bit-reversed binary words in reversed lexicographic order:

```
void C(ulong f, ulong n, ulong w)
{
    for (ulong m=1; m<n; m<<=1)  C(f+m, m, w^m);
    print_bin("  ", w, 10);  // visit
}
```

Calling `C(0, 32, 0)` we obtain

```
        .........1
        ........11
        ........1.
        .......1.1
        .......111
        .......11.
        .......1..
        ......1..1
          [-snip-]
        .....111.1
        .....11111
        .....1111.
        .....111..
        .....11...
        .....1....
        ..........  // note zero appears at end
```

A slight modification of the function

```
void A(ulong f, ulong n)
{
    cout << "1,";
    for (ulong m=1; m<n; m<<=1)  A(f+m, m);
    cout << "0,";
}
```

generates the Taylor series (sequence A079559 of [OEIS])

$$\prod_{n=1}^{\infty}\left(1+x^{2^n-1}\right) \quad = \quad 1+x+x^3+x^4+x^7+x^8+x^{10}+x^{11}+x^{15}+x^{16}+\ldots \tag{1.16}$$

Calling `A(0, 32)` we obtain:

```
1,1,0,1,1,0,0,1,1,0,1,1,0,0,0,1,1,0,1,1,0,0,1,1,0,1,1,0,0,0,0, ...
```

Indeed, the lowest bit of the k-th word of the bit-reversed sequence in reversed lexicographic order equals the $(k-1)$-st coefficient in the Taylor series. The sequence can also be generated with a string substitution engine (see section 5.21 on page 206):

```
Start: 1x
Rules:
  0 --> 0
  1 --> 110
  x --> x
-------------
0:   (#=2)
  1x
1:   (#=4)
  110x
2:   (#=8)
  1101100x
3:   (#=16)
  110110011011000x
4:   (#=32)
  11011001101100011011001100110000x
5:   (#=64)
  1101100110110001101100110110000011011001101100011011001101100000x
```

1.30 Minimal-change bit combinations

The wonderful

```
static inline ulong igc_next_minchange_comb(ulong x)
// Return the inverse graycode of the next combination in minchange order.
// Input must be the inverse graycode of the current combination.
{
    ulong g = green_code(x);
    ulong i = 2;
    ulong cb; // ==candidateBits;
    do
    {
        ulong y = (x & ~(i-1)) + i;
        ulong j = lowest_bit(y) << 1;
        ulong h = !!(y & j);
        cb = ((j-h) ^ g) & (j-i);
        i = j;
    }
    while ( 0==cb );

    return  x + lowest_bit(cb);
}
```

together with

```
static inline ulong igc_last_comb(ulong k, ulong n)
// Return the (inverse graycode of the) last combination
// as in igc_next_minchange_comb()
{
    if ( 0==k )  return 0;
    ulong f = first_sequency(k);
    ulong c = first_comb(n);
    return c ^ (f>>1);
    // =^=  (by Doug Moore)
//    return  ((1UL<<n) - 1) ^ (((1UL<<k) - 1) / 3);
}
```

could be used as demonstrated in [FXT: file bits/bitcombminchange.h]

```
static inline ulong next_minchange_comb(ulong x, ulong last)
// not efficient, just to explain the usage
```

```
// of igc_next_minchange_comb()
// Must have: last==igc_last_comb(k, n)
//
// Example with  k==3, n==5:
//      x       inverse_gray_code(x)
//    ..111       ..1.1 == first_sequency(k)
//    .11.1       .1..1
//    .111.       .1.11
//    .1.11       .11.1
//    11..1       1...1
//    11.1.       1..11
//    111..       1.111
//    1.1.1       11..1
//    1.11.       11.11
//    1..11       111.1 == igc_last_comb(k, n)
{
    x = inverse_gray_code(x);
    if ( x==last )  return 0;
    x = igc_next_minchange_comb(x);
    return  gray_code(x);
}
```

Each combination is different from the preceding one in exactly two positions. The same run of bit combinations could be obtained by going through the Gray codes and omitting all words where the bit-count is not equal to k. The algorithm shown here, however, is much more efficient.

For reasons of efficiency one may prefer code as

```
ulong last = igc_last_comb(k, n);
ulong c, nc = first_sequency(k);
do
{
    c = nc;
    nc = igc_next_minchange_comb(c);
    ulong g = gray_code(c);
    // Here g contains the bitcombination
}
while ( c!=last );
```

which avoids the repeated computation of the inverse Gray code.

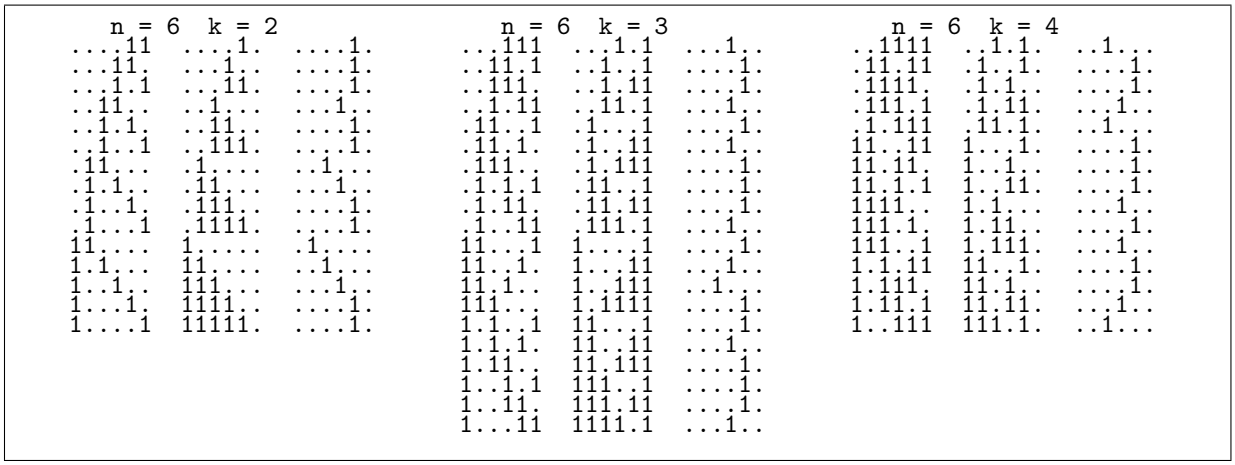


Figure 1.17: Minimal-change combinations, their inverse Gray codes and the differences of the inverse Gray codes. The differences are powers of two.

As Doug Moore explains [priv.comm.], the algorithm in `igc_next_minchange_comb()` uses the fact that the difference of two (inverse gray codes of) successive combinations is always a power of two, see figure 1.17. See also [FXT: demo bits/bitcombminchange-demo.cc].

Using this observation one can derive a different version that checks the pattern of the change:

```
static inline ulong igc_next_minchange_comb(ulong x)
// Alternative version.
```

```
// Amortized time = O(1).
{
    ulong gx = gray_code( x );
    ulong y, i = 2;
    do
    {
        y = x + i;
        ulong gy = gray_code( y );
        ulong r = gx ^ gy;
        // Check that change consists of exactly one bit
        // of the new and one bit of the old pattern:
        if ( is_pow_of_2( r & gy ) && is_pow_of_2( r & gx ) )  break;
        // is_pow_of_2(x):=((x & -x) == x)  returns 1 also for x==0.
        // But this cannot happen for both tests at the same time
        i <<= 1;
    }
    while ( 1 );
    return  y;
}
```

Still another version which needs `k`, the number of set bits, as a second parameter:

```
static inline ulong igc_next_minchange_comb(ulong x, ulong k)
// Alternative version, uses the fact that the difference
// of two successive x is the smallest possible power of 2.
// Should be fast if the CPU has a bitcount instruction.
// Amortized time = O(1).
{
    ulong y, i = 2;
    do
    {
        y = x + i;
        i <<= 1;
    }
    while ( bit_count( gray_code(y) ) != k );
    return  y;
}
```

The necessary modification for the generation of the previous combination is trivial:

```
static inline ulong igc_prev_minchange_comb(ulong x, ulong k)
// Returns the inverse graycode of the previous combination in minchange order.
// Input must be the inverse graycode of the current combination.
// Amortized time = O(1).
// With input==first the output is the last for n=BITS_PER_LONG
{
    ulong y, i = 2;
    do
    {
        y = x - i;
        i <<= 1;
    }
    while ( bit_count( gray_code(y) ) != k );
    return  y;
}
```

1.31 Minimal-change Fibonacci words *

A Gray code for the binary Fibonacci words (shown in figure 1.18 on the next page) can be obtained by using the Gray code of the radix −2 representations (see section 1.25 on page 53) of binary words whose difference is of the form

```
    1    ...............1
    3    ..............i1
    5    .............i.1
    9    ............i..1
   19    ...........i..i1
   37    ..........i..i.1
   73    .........i..i..1
  147    ........i..i..i1
  293    .......i..i..i.1
  585    ......i..i..i..1
 1171    .....i..i..i..i1
 2341    ....i..i..i..i.1
 4681    ...i..i..i..i..1
 9363    ..i..i..i..i..i1
```

```
     j:        k(j)      k(j)-k(j-1)   x=bin2neg(k)    gray(x)
     1:    ....11...1    .........    ...111...1    ...1..1..1 =  27
     2:    ....11....    .........1   ...111....    ...1..1...  =  26
     3:    ....1.1111    .........1   ...111..11    ...1..1.1.  =  28
     4:    ....1.11..    ........11   ...11111..    ...1....1.  =  23
     5:    ....1.1.11    .........1   ...1111111    ...1......  =  21
     6:    ....1.1.1.    .........1   ...111111.    ...1.....1  =  22
     7:    ....1.1..1    .........1   ...1111..1    ...1...1.1  =  25
     8:    ....1.1...    .........1   ...1111...    ...1...1..  =  24
     9:    ....1...11    .......1.1   ...11..111    ...1.1.1..  =  32
    10:    ....1...1.    .........1   ...11..11.    ...1.1.1.1  =  33
    11:    ....1....1    .........1   ...11....1    ...1.1...1  =  30
    12:    ....1.....    .........1   ...11.....    ...1.1....  =  29
    13:    .....11111    .........1   ...11...11    ...1.1..1.  =  31
    14:    ......11..    .....1..11   .....111..    .....1..1.  =  10
    15:    ......1.11    .........1   .....11111    .....1....  =   8
    16:    ......1.1.    .........1   .....1111.    .....1...1  =   9
    17:    ......1..1    .........1   .....11..1    .....1.1.1  =  12
    18:    ......1...    .........1   .....11...    .....1.1..  =  11
    19:    ........11    .......1.1   .......111    .......1..  =   3
    20:    ........1.    .........1   .......11.    .......1.1  =   4
    21:    .........1    .........1   .........1    .........1  =   1
    22:    ..........    .........1   ..........    ..........  =   0
    23:    1111111111    .........1   ........11    ........1.  =   2
    24:    11111111..    ........11   ......11..    ......1.1.  =   7
    25:    1111111.11    .........1   ......1111    ......1...  =   5
    26:    1111111.1.    .........1   ......111.    ......1..1  =   6
    27:    111111...1    ......1..1   ....11...1    ....1.1..1  =  19
    28:    111111....    .........1   ....11....    ....1.1...  =  18
    29:    11111.1111    .........1   ....11..11    ....1.1.1.  =  20
    30:    11111.11..    ........11   ....1111..    ....1...1.  =  15
    31:    11111.1.11    .........1   ....111111    ....1.....  =  13
    32:    11111.1.1.    .........1   ....11111.    ....1....1  =  14
    33:    11111.1..1    .........1   ....111..1    ....1..1.1  =  17
    34:    11111.1...    .........1   ....111...    ....1..1..  =  16
```

Figure 1.18: Gray code for the binary Fibonacci words (rightmost column).

The algorithm is to try these values as increments starting from the least, same as for the minimal-change combination described in section 1.30 on page 63. The next valid word is encountered if it is a valid Fibonacci word, that is, it does not contain two consecutive set bits. The implementation is [FXT: `class bit_fibgray` in bits/bitfibgray.h]:

```
class bit_fibgray
// Fibonacci Gray code with binary words.
{
public:
    ulong x_;  // current Fibonacci word
    ulong k_;  // aux
    ulong fw_, lw_;  // first and last Fibonacci word in Gray code
    ulong mw_;  // max(fw_, lw_)
    ulong n_;  // Number of bits

public:
    bit_fibgray(ulong n)
    {
        n_ = n;

        fw_=0;
        for (ulong m=(1UL<<(n-1)); m!=0; m>>=3)  fw_ |= m;
        lw_ = fw_ >> 1;
        if ( 0==(n&1) )  { ulong t=fw_; fw_=lw_; lw_=t; }
        mw_ = ( lw_>fw_ ? lw_ : fw_ );
        x_ = fw_;

        k_ = inverse_gray_code(fw_);
        k_ = neg2bin(k_);
    }

    ~bit_fibgray() {;}

    ulong next()
    // Return next word in Gray code.
    // Return ~0 if current word is the last one.
    {
        if ( x_ == lw_ )  return ~0UL;

        ulong s = n_;
        while ( 1 )
```

```
        {
            --s;
            ulong c = 1 | (mw_ >> s);
            ulong i = k_ - c;
            ulong x = bin2neg(i);
            x ^= (x>>1);
            if ( 0==(x&(x>>1)) )
            {
                k_ = i;
                x_ = x;
                return x;
            }
        }
    }
};
```

The algorithm is CAT (*constant amortized time*) and fast in practice. More than 100 million words are produced per second. The program [FXT: demo bits/bitfibgray-demo.cc] shows how to use the class, figure 1.18 was created with it.

Section 5.18 on page 202 gives a recursive algorithm for Fibonacci words in Gray code order.

1.32 Binary words and parentheses *

A subset of the binary words can be interpreted as a (well formed) string of parentheses. The 4-bit binary words that have this property are marked with a 'P' in the following list which was produced with [FXT: demo bits/parenword-demo.cc]:

```
 0    .... P   [empty string]
 1    ...1 P   ()
 2    ..1.
 3    ..11 P   (())
 4    .1..
 5    .1.1 P   ()()
 6    .11.
 7    .111 P   ((()))
 8    1...
 9    1..1
10    1.1.
11    1.11 P   (()())
12    11..
13    11.1 P   ()(())
14    111.
15    1111 P   (((())))
```

The functions shown in what follows are given in [FXT: file bits/parenwords.h].

The strings are constructed by scanning the word from the low end and printing a '(' with each one and a ')' with each zero. In order to find out when to terminate one adds up +1 for each open parenthesis and −1 for a closing parenthesis. When the ones in the binary word have been scanned then s closing parentheses have to be added where s is the value of the sum.

```
inline void parenword2str(ulong x, char *str)
{
    int s = 0;
    ulong j = 0;
    for (j=0; x!=0; ++j)
    {
        s += ( x&1 ? +1 : -1 );
        str[j] = ")("[x&1];
        x >>= 1;
    }
    while ( s-- > 0 )  str[j++] = ')';  // finish string
    str[j] = 0;  // terminate string
}
```

The 5-bit binary words that are valid 'paren words' together with the corresponding strings are

```
    .....   [empty string]
    ....1   ()
    ...11   (())
    ..1.1   ()()
    ..111   ((()))
```

```
    .1.11   (()())
    .11.1   ()(())
    .1111   ((()))
    1..11   (())()
    1.1.1   ()()()
    1.111   ((()))
    11.11   (()(()))
    111.1   ()((()))
    11111   ((((()))))
```

Note that the lower bits in the word (right end) correspond to the begin of the string (left end).

If a negative value for the sums occurs at any time of the computation then the word is not a paren word. We use that fact to create a routine that determines whether a word is a paren word:

```
inline bool is_parenword(ulong x)
{
    int s = 0;
    for (ulong j=0; x!=0; ++j)
    {
        s += ( x&1 ? +1 : -1 );
        if ( s<0 )  break;  // invalid word
        x >>= 1;
    }
    return  (s>=0);
}
```

The sequence

```
  1, 3, 5, 7, 11, 13, 15, 19, 21, 23, 27, 29, 31, 39, 43, 45, 47, 51, 53, 55, 59, 61, 63, ...
```

of nonzero integers x so that `is_parenword(x)` returns `true` is sequence A036991 of [OEIS].

If we fix the number of paren pairs then the following functions generate the least and biggest valid binary word:

```
inline ulong first_parenword(ulong n)
// Return least binary word corresponding to n pairs of parens
{
    return first_comb(n);
}
// Example, n=5:  .....11111   (((((())))))
```

The function simply returns a word with a block of n ones at the low end.

```
inline ulong last_parenword(ulong n)
// Return biggest binary word corresponding to n pairs of parens
{
    ulong x = 1UL << (2*n-2),  s = 2;
    do
    {
        x |= (x>>s);
        s <<= 1;
    }
    while ( s < BITS_PER_LONG );
    return x;
}
```

The function returns a word with a sequence of n blocks '01' at the low end. For Example, with $n = 5$:

```
    .1.......  == 1 << (2*5-2)  == 1 << 8  =: x
    .1.1.....  == x | (x>>2) =: x
    .1.1.1.1..  == x | (x>>4) =: x
    .1.1.1.1.1  == x | (x>>8) =: x
    .1.1.1.1.1  =^= "()()()()()"
```

The sequence of all valid words corresponding to n pairs of parens in colex order can be generated with the following (slightly cryptic) function:

```
inline ulong next_parenword(ulong x)
// Next (colex order) binary word that is a paren word.
{
    ulong b = lowest_block(x);
    if ( b!=1 )  // split off highest bit of lowest block:
    {
        ++b;
        b ^= (b>>1);
        x ^= b;
```

```
    }
    else // Gather all low "01"s and split lowest nontrival block:
    {
        if ( 0==(x & (x>>1)) )  return 0;
        ulong w = 0;  // word where the bits are assembled
        ulong s = 0;  // shift for lowest block
        ulong b = 1;  // == lowest_bit(x)
        do
        {
            x ^= b;
            w <<= 1;
            w |= 1;
            ++s;
            b = lowest_bit(x);
        }
        while ( 0==(x&(b<<1)) );

        ulong z = x ^ (x+b);  // lowest block
        x ^= z;
        z &= (z>>1);
        z &= (z>>1);
        w ^= (z>>s);
        x |= w;
    }
    return x;
}
```

The program [FXT: demo bits/parenword-colex-demo.cc] shows how to create a list of binary words corresponding to n pairs of parens (code slightly shortened):

```
int
main(int argc, char **argv)
{
    ulong n = 5;  // Number of paren pairs
    ulong pn = 2*n+1;
    char *str = new char[n+1];  str[n] = 0;
    ulong x = first_parenword(n);
    while ( x )
    {
        ++pct;
        print_bin_nn("  ", x, pn);
        parenword2str(x, str);
        cout << "   " << str;
        cout << endl;

        x = next_parenword(x);
    }
    return 0;
}
```

Its output with $n = 4$ is

```
    .....1111   (((())))
    ....1.111   ((()()))
    ....11.11   (()(()))
    ....111.1   ()((()))
    ...1..111   ((())())
    ...1.1.11   (()()())
    ...1.11.1   ()(()())
    ...11..11   (())(())
    ...11.1.1   ()()(())
    ..1...111   ((()))()
    ..1..1.11   (()())()
    ..1..11.1   ()(())()
    ..1.1..11   (())()()
    ..1.1.1.1   ()()()()
```

Section 5.5 on page 159 gives a different formulation of the algorithm.

1.33 Error detection by hashing: the CRC

A *hash value* is an element from a set H that is computed via a *hash function* f that maps any (finite) sequence of input data to H.

For the sake of simplicity we now consider the case that the input sequences are of fixed size so they are in a fixed set, say S. We further assume that the set S is (much) bigger than H.

$$f : S \to H, \qquad s \mapsto h \tag{1.17}$$

where $s \in S$ and $h \in H$.

Two input sequences with different hash values are necessarily different. But, as the hash function maps a larger set to a smaller one, there are different input sequences with identical hash values.

A trivial example is the set $H = \{0, 1\}$ together with a function that count binary digits modulo two, the parity function. Another example is the sum-of-digits test (see section 14.5 on page 394) used to check the multiplication of large numbers. In the test we compute the value of a multi digit number decimal $s \in S$ modulo nine. The crucial additional property of this hash is that with $f(A) = a$, $f(B) = b$, $f(C) = c$ (were A, B, C are decimal numbers), then $A \cdot B = C$ implies $a \cdot b = c$.

A hash function f that is actually useful should have the *mixing* property: It should map the elements $s \in S$ 'randomly' to H. With the sum-of-digits test we could have used rather arbitrary moduli for the hash function. With one exception: the value modulo ten as hash would be pretty useless. No change in any digit except for the last could ever be detected.

The so-called *cyclic redundancy check* (CRC) is a hash where the hash values are binary words of fixed length. The hash function (basically) computes $h = s \bmod c$ where s is a binary polynomial build from the input sequence and c is a binary polynomial that is primitive (see chapter 22 on page 569). We will use polynomials c of degree 64 so the hash values (CRCs) are 64-bit words.

An C++ implementation is given as [FXT: `class crc64` in bits/crc64.h]:

```
class crc64
// 64-bit CRC (cyclic redundancy check)
{
public:
    uint64 a_;  // internal state (polynomial modulo c)
    uint64 c_;  // a binary primitive polynomial
    // (non-primitive c lead to smaller periods)
    // The leading coefficient needs not be present.
    uint64 h_;  // auxiliary

    static const uint64 cc[];  // 16 "random" 64-bit primitive polynomials
public:
    crc64(uint64 c=0)
    {
        if ( 0==c )  c = 0x1bULL; // =^= 64,4,3,1,0 (default)
        init(c);
    }

    ~crc64()  {;}

    void init(uint64 c)
    {
        c_ = c;
        c_ >>= 1;
        h_ = 1ULL<<63;
        c_ |= h_;  // leading coefficient
        reset();
    }

    void reset()  { set_a(~0ULL); }  // all ones
    void set_a(uint64 a)  { a_=a; }

    uint64 get_a()  const  { return a_; }
   [-snip-]
```

Note that a nonzero initial state (member variable `a`) is used: Starting with zero will only go to a nonzero state with the first nonzero bit in the input sequence. That is, input sequences differing only by initial runs of of zeros would get the same CRC.

After an instance is created one can feed in bits via `bit_in(b)` where lowest bit of `b` must contain the bit to be used:

```
   [-snip-]
    void shift()
    {
        bool s = (a_ & 1);
        a_ >>= 1;
        if ( 0!=s )  a_ ^= c_;
    }

    uint64 bit_in(unsigned char b)
    {
        a_ ^= (b&1);
        shift();
        return  a_;
    }
   [-snip-]
```

When a byte is to be checksummed we can do better than just feeding in the bits one by one. This is achieved adding the byte followed by eight calls to `shift()`:

```
   [-snip-]
    uint64 byte_in(unsigned char b)
    {
#if 1
        a_ ^= b;
        shift();  shift();  shift();  shift();
        shift();  shift();  shift();  shift();
#else  // identical but slower:
        bit_in(b);  b>>=1;  // bit 0
        bit_in(b);  b>>=1;  // bit 1
        bit_in(b);  b>>=1;  // bit 2
        bit_in(b);  b>>=1;  // bit 3
        bit_in(b);  b>>=1;  // bit 4
        bit_in(b);  b>>=1;  // bit 5
        bit_in(b);  b>>=1;  // bit 6
        bit_in(b);  b>>=1;  // bit 7
#endif
        return  a_;
    }
   [-snip-]
```

The lower block implements the straightforward idea. The program [FXT: demo bits/crc64-demo.cc] computes the 64-bit CRC of a single byte in both ways.

Binary words are fed in byte by byte, starting from the lower end:

```
    uint64 word_in(uint64 w)
    {
        ulong k = BYTES_PER_LONG_LONG;
        while ( k-- )  { byte_in( (uchar)w );  w>>=8; }
        return  a_;
    }
```

Finally, if one wants to feed in a given number of bits of a word, the following member function can be used:

```
    uint64 bits_in(uint64 w, uchar k)
    {
        if ( k&1 )  { a_ ^= (w&1); w >>= 1;  shift(); }
        k >>= 1;

        if ( k&1 )  { a_ ^= (w&3); w >>= 2; shift(); shift(); }
        k >>= 1;

        if ( k&1 )  { a_ ^= (w&15); w >>= 4; shift(); shift(); shift(); shift(); }
        k >>= 1;

        while ( k-- )  { byte_in( (uchar)w );  w>>=8; }
    }
```

The operation is the optimized equivalent to

```
        while ( k-- )  { bit_in( (uchar)w );  w>>=1; }
```

If two sequences differ in a single block of up to 64 bits, their CRCs will be different. The probability that two different sequences have the same CRC equals $2^{-64} \approx 5.42 \cdot 10^{-20}$. If that is not enough (and one does not want to write a CRC with more than 64 bits) then one can simply use two (or more) instances where different polynomials must be used. The first 16 'random' primitive polynomials are given [FXT: file bits/crc64.cc] as static class member `ulong cc[]`:

```
const uint64 crc64::cc[] = {
    0x5a0127dd34af1e81ULL,  // [0]
    0x4ef12e145d0e3ccdULL,  // [1]
    0x16503f45acce9345ULL,  // [2]
    0x24e8034491298b3fULL,  // [3]
    0x9e4a8ad2261db8b1ULL,  // [4]
    0xb199aecfbb17a13fULL,  // [5]
    0x3f1fa2cc0dfbbf51ULL,  // [6]
    0xfb6e45b2f694fb1fULL,  // [7]
    0xd4597140a01d32edULL,  // [8]
    0xbd08ba1a2d621bffULL,  // [9]
    0xae2b680542730db1ULL,  // [10]
    0x8ec06ec4a8fe8f6dULL,  // [11]
    0xb89a2ecea2233001ULL,  // [12]
    0x8b996e790b615ad1ULL,  // [13]
    0x7eaef8397265e1f9ULL,  // [14]
    0xf368ae22deecc7c3ULL,  // [15]
};
```

These are taken from the list [FXT: file data/rand64-hex-primpoly.txt]. Initialize multiple CRCs as follows

```
crc64 crca( crc64::cc[0] );
crc64 crcb( crc64::cc[1] );
```

A class for 32-bit CRCs is given in [FXT: `class crc32` in bits/crc32.h]. Its usage is completely equivalent.

The CRC can easily be implemented in hardware and is, for example, used to detect errors in hard disk blocks. When a block is written its CRC is computed and stored in an extra word. When the block is read, the CRC is computed from the data and compared to the stored CRC. A mismatch indicates an error.

One property that the CRC does not have is the cryptographic security. It is possible to intentionally create a data set with a prescribed CRC. With *secure hashes* (like MD5 and SHA1) it is (practically) not possible to do so. Secure hashes can be used to 'sign' data. Imagine you distribute a file (for example an binary executable) over the internet. You want to make sure that someone downloading the file (from any source) can verify that it is not an altered version (like, in the case of an executable, a malicious program). To do so you create a (secure!) hash value which you publish on your web site. Any person can verify the authenticity of the file by computing the hash and comparing it to the published version. [Note added: recently successful attacks on both MD5 (see [104]) and SHA1 were reported].

1.33.1 Optimization via lookup tables

One can feed in an n-bit word w into the CRC in one step (instead of n steps) as follows: Add w to (the CRC word) a. Save the lowest n bits of the result to a variable x. Right shift a by n bits. Add to a the entry x of an auxiliary table t.

For $n = 8$ the operation can be implemented as [FXT: `class tcrc64` in bits/tcrc64.h]:

```
uint64 byte_in(uchar b)
{
    a_ ^= b;
    uint64 x =  t_[a_ & 255];
    a_ >>= 8;
    a_ ^= x;
    return  a_;
}
```

The size of the table is $2^n = 256$ words. For $n = 1$ the table would have only two entries, zero and c, the polynomial used. Then the implementation reduces to

```
uint64 bit_in(uchar b)
{
    a_ ^= (b&1);
    bool s = (a_ & 1);
    a_ >>= 1;
    if ( 0!=s )  a_ ^= c_;  // t[0]=0; t[1]=c_;
    return  a_;
}
```

which is equivalent to the `bit_in()` routine of the unoptimized CRC.

The lookup table is computed upon class initialization as follows:

```
for (ulong w=0; w<256; ++w)
{
    set_a(0);
    for (ulong k = 0; k<8; ++k)  bit_in( (uchar)w>>k );
    t_[w] = a_;
}
```

The class actually allows to use tables of either 16 or 256 words. When a table of size 16 is used, the computation is about 6 times faster than with the non-optimized routine. A table of size 256 gives a speedup by a factor of 12. Optimization techniques based on lookup tables are often used in practical applications, both in hardware and in software, see [29].

1.33.2 Parallel CRCs

A very fast method for checksumming is to compute the CRCs for each bit of the fed-in words in parallel. An array of 64 words is used [FXT: `class pcrc64` in bits/pcrc64.h]:

```
template <typename Type>
class pcrc64
// Parallel computation of 64-bit CRCs for each bit of the input words.
// Primitive polynomial used is x^64 + x^4 + x^3 + x^2 + 1
{
public:
    Type x_[64];    // CRC data
    // bit(i) of x_[0], x_[1], ..., x_[63] is a 64-bit CRC
    //   of bit(i) of all input words
    uint pos_;      // position of constant polynomial term
    const uint m_;  // mask to compute mod 64
```

Upon initialization all words are set to all ones:

```
public:
    pcrc64()
        : m_(63)
    {
        reset();
    }
    ~pcrc64()  { ; }
    void reset()
    {
        pos_ = 0;
        Type ff = 0;  ff = ~ff;
        for (uint k=0; k<64; ++k)  x_[k] = ff;
    }
```

The cyclic shift of the array is avoided by working modulo 64 when feeding in words:

```
    void word_in(Type w)
    {
        uint p = pos_;
        pos_ = (p+1) & m_;
        uint h = (p-1) & m_;
        Type a = x_[p & m_];  // 0
        p += 2;
        a ^= x_[p & m_];  // 2
        ++p;
        a ^= x_[p & m_];  // 3
        ++p;
        a ^= x_[p & m_];  // 4
        x_[h] = a ^ w;
    }
```

The algorithm corresponds to the Fibonacci setup of the linear feedback shift registers (see section 6.2 on page 220). There is no primitive trinomial with degree a multiple of eight so we use the pentanomial

$x^{64} + x^4 + x^3 + x^2 + 1$. With an array size where a primitive trinomial exists the modulo computations would be more expensive.

A unrolled routine can be used to feed in multiple words:

```
void words_in(Type *w, ulong n)
{
    if ( n&1 )  { word_in(w[0]); ++w; }
    n >>= 1;
    if ( n&1 )  { word_in(w[0]); word_in(w[1]); w+=2; }
    n >>= 1;
    for (ulong k=0; k<n; ++k)
    {
        word_in(w[0]);
        word_in(w[1]);
        word_in(w[2]);
        word_in(w[3]);
        w += 4;
    }
}
```

The program [FXT: demo bits/pcrc64-demo.cc] feeds the numbers up to a given value into a `pcrc64<uint>`:

```
int main()
{
    Type n = 32768;
    pcrc64<Type> P;
    for (Type k=0; k<n; ++k)  P.word_in(k);
    // print array P.x_[] here
}
```

This rather untypical type of input data illustrates the independence of the bits in the array `x_[]`:

```
 0:  ................1..1.1.11.111.
 1:  iiiiiiiiiiiiiiii.ii.i.1111.111i.
 2:  iiiiiiiiiiiiiiii..i.11...1...ii.
 3:  iiiiiiiiiiiiiiii.i..1111....ii.
 4:  iiiiiiiiiiiiiiii.i..1.1111.11..
 5:  ................ii...i..1.....i
 6:  ................1.1.11111.1.ii.
 7:  iiiiiiiiiiiiiiii.1.1.11.i..1...
 8:  iiiiiiiiiiiiiiii.i...i.1.1111.
 9:  iiiiiiiiiiiiiiii.1111.1.1.1..i
10:  iiiiiiiiiiiiiiii..111..i..1..i
11:  iiiiiiiiiiiiiiii..1111.i..i.1.1
12:  ................ii..1.11.11..1.
13:  ................ii.1.1..11.1111
[--snip--]
59:  ................1...1.11.11.1.1
60:  iiiiiiiiiiiiiiii.i..ii.111..1.1
61:  ................1.i111...i..ii.
62:  iiiiiiiiiiiiiiii.i..ii...i...i.
63:  ................iii..1.1111.iii
```

The implementation can process about 900 MB of data per second when integer types are used and about 850 MB/sec with character type.

1.34 Permutations via primitives

This section gives two methods to specify permutations of the bits of a binary word via one or more control words. The methods are suggestions for machine instructions that can serve as primitives for permutations of the bits of a word.

A restricted method

Generalizing the symbolic-powering idea given in chapter 1.19 on page 41 we can specify a subset of all permutations by selecting bit-blocks of the masks (for 32-bit words)

```
................1111111111111111
........11111111........11111111
....1111....1111....1111....1111
..11..11..11..11..11..11..11..11
```

```
.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1
```

Subsets of the blocks of the masks can be determined with the bits of a 32-bit word by considering the highest bit of each block:

```
................1................    bits  15 ...
........1...............1........    bits   7 ...
....1.......1.......1.......1....    bits   3  11 ...
..1...1...1...1...1...1...1...1..    bits   1   5   9  13 ...
.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1    bits   0   2   4   6   8  10  12  14 ...
```

Thereby one can use all (except for the highest) bits of a word to select the blocks where the bits defined by the block and those left to it should be swapped. A implementation of the implied algorithm can be found in [FXT: demo bits/bitperm1-demo.cc]. Arrays are used to give more readable code:

```
void perm1(uchar *a, ulong ldn, const uchar *x)
// Permute a[] according to the 'control word' x[].
// The length of a[] must be 2**ldn.
{
    long n = 1L<<ldn;
    for (long s=n/2; s>0; s/=2)
    {
        for (long k=0; k<n; k+=s+s)
        {
            if ( x[k+s-1]!='0' )
            {
                // swap regions [a+k,...,a+k+s-1], [a+k+s,...,a+k+2*s-1]:
                swap(a+k, a+k+s, s);
            }
        }
    }
}
```

No attempt has been made to optimize or parallelize the algorithm. We rather seek to explore how useful a machine instruction for the permutation of bits would be.

The program uses a fixed size of 16 bits, an 'x' is printed whenever the corresponding bit is set:

```
a=0123456789ABCDEF   bits of the input word
x=0010011000110110   control word
   8:    7
   4:    3  11x
   2:    1   5x  9  13x
   1:    0   2x  4   6x  8  10x 12  14x
a=01326754CDFEAB98   result
```

The control word used leads to the Gray code permutation (see 2.8 on page 91). Assume we use words with N bits. We cannot (for $N > 2$) obtain all $N!$ permutations as we can choose between only 2^{N-1} control words. Now set the word length to $N := 2^n$. The reachable permutations are those where the intervals $[k \cdot 2^j, \ldots, (k+1) \cdot 2^j - 1]$ contain all numbers $[p \cdot 2^j, \ldots, (p+1) \cdot 2^j - 1]$ for all $j \le n$ and $0 \le k < 2^{n-j}$, choosing p for each interval arbitrarily ($0 \le p < 2^{n-j}$). For example, the lower half of the permuted array must contain a permutation of either the lower or the upper half ($j = n-1$) and each pair a_{2y}, a_{2y+1} must contain two elements $2z$, $2z+1$ ($j = 1$). The bit-reversal can be obtained using a control word with all bits set. Alas, the (important!) zip permutation (bit-zip, see section 1.15 on page 32) is unreachable.

The inverse permutation is achieved by changing a single line:

```
for (long s=1; s<n; s+=s)
```

A machine instruction could choose between the two routines via the highest bit in the control word.

A general method

All permutations of $N = 2^n$ elements can be obtained using n control words of N bits. Assume we have a machine instruction that collects bits according to a control word. An eight bit example:

```
a = abcdefgh   input data
x = ..1.11.1   control word (dots for zeros)
      cefh     bits of a where x has a one
    abdg       bits of a where x has a zero
    abdgcefh   result, bits separated according to x
```

We need n such instructions that work on all length-2^k sub-words for $1 \le k \le n$. For example, the instruction working on half words of 16-bit word would work as

```
a = abcdefgh ABCDEFGH   input data
x = ..1.11.1 1111....   control word (dots for zeros)
      cefh     ABCD     bits of a where x has a one
    abdg     EFGH       bits of a where x has a zero
    abdgcefh EFGHABCD   result, bits separated according to x
```

Note the bits of the different sub-words are not mixed. Now all permutations can be reached if the control word for the 2^k-bit sub-words have exactly 2^{k-1} bits set in all ranges $[j \cdot 2^k, \ldots, (j+1) \cdot 2^k]$.

A control word together with the specification of the instruction used defines the action taken. The following leads to a swap of adjacent bit pairs

```
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.    k= 1  (2-bit sub-words)
```

while this

```
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.    k= 5 (32 bit sub-words)
```

results in gathering the even and odd indexed bits in the halfwords.

A complete set of permutation primitives for 16-bit words and their effect on a symbolic array of bits (split into groups of four elements for readability) is

```
                                0123 4567 89ab cdef
11111111........    k= 4  ==> 89ab cdef 0123 4567
1111....1111....    k= 3  ==> cdef 89ab 4567 0123
11..11..11..11..    k= 2  ==> efcd ab89 6745 2301
1.1.1.1.1.1.1.1.    k= 1  ==> fedc ba98 7654 3210
```

The top primitive leads to a swap of the left and right half of the bits, the next to a swap of the halves of the half words and so on. The obtained permutation is array reversal. Note that we use array notation (least index left) here.

The resulting permutation depends on the order in which the primitives are used. Starting with full words

```
                                0123 4567 89ab cdef
 1.1. 1.1. 1.1. 1.1.  k= 4  ==>  1357 9bdf 0246 8ace
 1.1. 1.1. 1.1. 1.1.  k= 3  ==>  37bf 159d 26ae 048c
 1.1. 1.1. 1.1. 1.1.  k= 2  ==>  7f3b 5d19 6e2a 4c08
 1.1. 1.1. 1.1. 1.1.  k= 1  ==>  f7b3 d591 e6a2 c480
```

The result is different when starting with 2-bit sub-words:

```
                                0123 4567 89ab cdef
 1.1. 1.1. 1.1. 1.1.  k= 1  ==>  1032 5476 98ba dcfe
 1.1. 1.1. 1.1. 1.1.  k= 2  ==>  0213 4657 8a9b cedf
 1.1. 1.1. 1.1. 1.1.  k= 3  ==>  2367 0145 abef 89cd
 1.1. 1.1. 1.1. 1.1.  k= 4  ==>  3715 bf9d 2604 ae8c
```

There are $\binom{2z}{z}$ possibilities to have z bits set in a $2z$-bit word. There are 2^{n-k} length-2^k sub-words in a 2^n-bit word so the number of valid control words for that step is

$$\left[\binom{2^k}{2^{k-1}}\right]^{2^{n-k}}$$

The product of the number of valid words in all steps gives the number of permutations:

$$(2^n)! \;=\; \prod_{k=1}^{n} \left[\binom{2^k}{2^{k-1}}\right]^{2^{n-k}} \tag{1.18}$$

1.35 CPU instructions often missed

Essential

- Bit-shift and bit-rotate instructions that work properly for shifts greater or equal to the word length: The shift instruction should zero the word, the rotate instruction should take the shift modulo word length. The C-language standards leave the results for these operations undefined and compilers simply emit the corresponding assembler instructions. The resulting CPU dependent behavior is both a source of errors and makes certain optimizations impossible.
- A bit-reverse instruction. A fast byte-swap mitigates the problem, see section 1.14 on page 28.
- Instructions that return the index of highest or lowest set bit in a word.
- Fast conversion from integer to float and double (both directions).
- A fused multiply-add instruction for floats.
- Instructions for the multiplication of complex floating point numbers. For example, with 128-bit (SIMD) registers and 64-bit floats:

```
// Here: R1 == A, B;  R2 == C, D;
CMUL R1, R2;
// Here: R2 == A*C-B*D, A*D+B*C;
```

- A sum-diff instruction, such as:

```
// Here:  R1 == A;  R2 == B;
SUMDIFF A, B;
// Here:  R1 == A+B;  R2 == A-B;
```

  It serves as a primitive for fast orthogonal transforms.

- An instruction to swap registers. Even better, a conditional version of that.

Nice to have

- A parity bit for the complete machine word. The parity of a word is the number of bits modulo two, not the complement of it. Even better would be an instruction for the inverse Gray code, see section 1.16 on page 33.
- A bit-count instruction, see section 1.8 on page 18. This would also give the parity at bit zero.
- A random number generator: LHCAs (see section 6.6 on page 225) may be candidates.
- A conditional version of more than just the move instruction, possibly as an instruction prefix.
- An instruction to detect zero bytes in a word, see section 1.22 on page 49. The C-convention is to use a zero byte as string terminator. Performance of the string related functions in the C-library could thereby be increased significantly. Ideally the instruction should exist for different word sizes: 4-byte, 8-byte and 16-byte (possibly using SIMD extensions).
- A bit-zip and a bit-unzip instruction, see section 1.15 on page 32 and [FXT: file bits/bitzip.h]. Note this is polynomial squaring over $GF(2)$.
- A bit-gather and a bit-scatter instruction, see [FXT: file bits/bitgather.h] and [FXT: file bits/bitseparate.h]. This would include bit-zip and its inverse.
- Primitives for permutations of bits, see section 1.34 on page 74.

Luxury items

- Multiplication corresponding to XOR as addition. That is, a multiplication without carries, the one used for polynomials over $GF(2)$. See section 22.1 on page 569 and [FXT: `bitpol_mult()` in bpol/bitpolarith.h].
- The mentioned multiplication for polynomials over $GF(2)$ with a modulus, see section 22.3 on page 579 and [FXT: `bitpolmod_mult()` in bpol/bitpolmodarith.h].

Chapter 2

Permutations

In this chapter we first study several special permutations that are used as part of other algorithms like the revbin permutation used in radix-2 FFT algorithms. Then permutations in general together with the operations on them, like composition and inversion, are described.

Algorithms for the the generation of all permutations of a given number of objects are given in section 5.17.

2.1 The revbin permutation

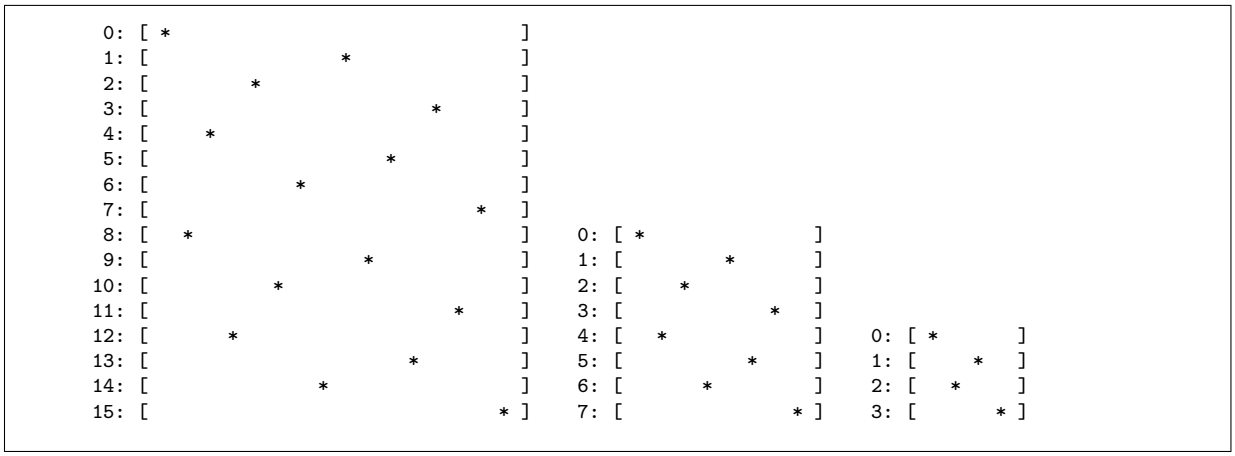


Figure 2.1: Permutation matrices of the revbin permutation for sizes 16, 8 and 4. The permutation is self-inverse.

The permutation that swaps elements whose indices are mutually bit-reversals is called *revbin permutation* (sometimes also *bit-reversal-* or *bitrev permutation*). For example, for length $n = 256$ the element with index $x = 43_{10} = 00101011_2$ is swapped with the element whose index is $\tilde{x} = 11010100_2 = 212_{10}$. Note that $\tilde{x}$ depends on both x and on n.

A first implementation might look like

```
procedure revbin_permute(a[], n)
// a[0..n-1] input,result
{
    for x:=0 to n-1
    {
        r := revbin(x, n)
        if r>x then  swap(a[x], a[r])
    }
```

```
}
```

The condition `r>x` before the `swap()` statement makes sure that the swapping isn't undone later when the loop variable `x` has the value of the present `r`. As we have seen in section 1.14 on page 28 the function function `revbin(x, n)` that computes the bit-reversed word is usually expensive.

Speedup via revbin-update

The key ingredient for a fast permutation routine is the observation that we only need to update as bit-reversed values. Given $\tilde{x}$ we can compute $\widetilde{x+1}$ efficiently. Methods to do this are given in section 1.14 on page 28. A faster routine will be of the form

```
procedure revbin_permute(a[], n)
// a[0..n-1] input,result
{
    if n<=2  return
    r := 0  // the reversed 0
    for x:=1 to n-1
    {
        r := revbin_update(r, n)  // inline me
        if r>x then  swap(a[x], a[r])
    }
}
```

The update of the reversed word (`revbin_update()`) needs for half of the calls just one iteration because in half of the updates just the leftmost bit changes[1], in half of the remaining updates it needs two iterations, in half of the still remaining updates it needs three and so on. The total number of operations done by `revbin_update()` is therefore proportional to

$$n\left(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\cdots+\frac{\log_2(n)}{n}\right) \quad = \quad n\sum_{j=1}^{\log_2(n)}\frac{j}{2^j} \tag{2.1}$$

For n large this sum is close to $2n$ so the average cost of the update is constant.

The number of swaps

How many `swap()`-statements will be executed in total for different n? About $n-\sqrt{n}$, as there are only few numbers with symmetric bit patterns: for even $log_2(n) =: 2\,b$ the left half of the bit pattern must be the reversed of the right half. There are $2^b = \sqrt{2^{2b}}$ such numbers. For odd $log_2(n) =: 2\,b+1$ there are twice as much symmetric patterns: the bit in the middle does not matter and can be 0 or 1.

n	2 # swaps	# symm. pairs
2	0	2
4	2	2
8	4	4
16	12	4
32	24	8
64	56	8
2^{10}	992	32
2^{20}	$0.999\cdot 2^{20}$	2^{10}
∞	$n-\sqrt{n}$	$\sqrt{n}$

That is, almost all 'revbin-pairs' will be swapped by the permutation. The sequence

```
0, 0, 2, 4, 12, 24, 56, 112, 238, 480, 992, 1980, 4032, 8064, 16242, 32512, 65280, ...
```

is entry A045687 of [OEIS].

[1] corresponding to the change in only the rightmost bit if one is added to an even number

Symmetries of the permutation

The following symmetry can be used for further optimization: If for even $x < \frac{n}{2}$ there is a swap (for the pair x, $\tilde{x}$) then there is also a swap for the pair $n-1-x$, $n-1-\tilde{x}$. As $x < \frac{n}{2}$ and $\tilde{x} < \frac{n}{2}$ one has $n-1-x > \frac{n}{2}$ and $n-1-\tilde{x} > \frac{n}{2}$. That is, the swaps are independent. A routine that uses these observations is

```
procedure revbin_permute(a[], n)
{
    if n<=2  return
    nh := n/2
    r := 0  // the reversed 0
    x := 1
    while x<nh
    {
        // x odd:
        r := r + nh
        swap(a[x], a[r])
        x := x + 1

        // x even:
        r := revbin_update(r,n)  // inline me
        if r>x then
        {
            swap(a[x], a[r])
            swap(a[n-1-x], a[n-1-r])
        }
        x := x + 1
    }
}
```

The code above is an ideal candidate to derive an optimized version for zero padded data:

```
procedure revbin_permute0(a[], n)
{
    if n<=2  return
    nh := n/2
    r := 0  // the reversed 0
    x := 1
    while x<nh
    {
        // x odd:
        r := r + nh
        a[r] := a[x]
        a[x] := 0
        x := x + 1

        // x even:
        r := revbin_update(r, n)  // inline me
        if r>x then  swap(a[x], a[r])
        // Omit swap of a[n-1-x] and a[n-1-r] as these are zero
        x := x + 1
    }
}
```

One can carry the scheme further, distinguishing whether $x \bmod 4 = 0$, 1, 2 or 3. This is done in the actual C++ implementation [FXT: `revbin_permute()` in perm/revbinpermute.h]. The parameters

```
#define  RBP_SYMM  4  // 1, 2, 4 (default is 4)
#define  FAST_BIT_SCAN  // define if machine has fast bit scan
```

allow to adjust how much of the symmetry is used (`RBP_SYMM`) and which flavor of the revbin-update routine is chosen (`FAST_BIT_SCAN`). With a fast bit-scan instruction the table driven version is slightly faster. We further define a convenient macro to swap elements:

```
#define  idx_swap(k, r)  { ulong kx=(k), rx=(r);  swap2(f[kx], f[rx]); }
```

The main routine uses unrolled versions of the revbin permutation for small values of n. These are given in [FXT: file perm/shortrevbinpermute.h]. For example, the unrolled routine for $n = 16$ is

```
template <typename Type>
inline void revbin_permute_16(Type *f)
{
    swap2(f[1], f[8]);
    swap2(f[2], f[4]);
    swap2(f[3], f[12]);
    swap2(f[5], f[10]);
    swap2(f[7], f[14]);
```

```
    swap2(f[11], f[13]);
}
```

The routine `revbin_permute_le_64(f,n)` that is called for $n \leq 64$ selects the correct routine for the parameter n:

```
template <typename Type>
void revbin_permute(Type *f, ulong n)
{
    if ( n<=64 )
    {
        revbin_permute_le_64(f, n);
        return;
    }
}
```

If the table driven update is used, the table has to be initialized and, depending on the amount of symmetry used, also some auxiliary variables:

```
    const ulong nh = (n>>1);
#ifdef FAST_BIT_SCAN
    make_revbin_upd_tab(nh);
#endif

#if  ( RBP_SYMM >= 2 )
    const ulong n1  = n - 1;    // = 11111111
#if  ( RBP_SYMM >= 4 )
    const ulong nx1 = nh - 2;   // = 01111110
    const ulong nx2 = n1 - nx1; // = 10111101
#endif //  ( RBP_SYMM >= 4 )
#endif //  ( RBP_SYMM >= 2 )
```

In what follows we set `RBP_SYMM` to 4 and omit the corresponding preprocessor statements. The cryptic and powerful main loop is

```
    ulong k = 0,  r = 0;
    while ( k<n/RBP_SYMM  )  // n>=16, n/2>=8, n/4>=4
    {
        // ----- k%4 == 0:
        if ( r>k )
        {
            idx_swap(k, r);  // <nh, <nh 11
            idx_swap(n1^k, n1^r);  // >nh, >nh 00
            idx_swap(nx1^k, nx1^r);  // <nh, <nh 11
            idx_swap(nx2^k, nx2^r);  // >nh, >nh 00
        }

        r ^= nh;
        ++k;

        // ----- k%4 == 1:
        if ( r>k )
        {
            idx_swap(k, r);  // <nh, >nh 10
            idx_swap(n1^k, n1^r);  // >nh, <nh 01
        }
#ifdef FAST_BIT_SCAN
        r = revbin_tupd(r, k);
#else
        r = revbin_upd(r, nh);
#endif
        ++k;

        // ----- k%4 == 2:
        if ( r>k )
        {
            idx_swap(k, r);  // <nh, <nh 11
            idx_swap(n1^k, n1^r); // >nh, >nh 00
        }

        r ^= nh;
        ++k;

        // ----- k%4 == 3:
        if ( r>k )
        {
            idx_swap(k, r);    // <nh, >nh 10
            idx_swap(nx1^k, nx1^r);   // <nh, >nh 10
        }
```

```
#ifdef FAST_BIT_SCAN
        r = revbin_tupd(r, k);
#else
        r = revbin_upd(r, nh);
#endif
        ++k;
    }
}  // end revbin_permute()
```

It turns out that the routine takes, for large n, about six times of the simple `reverse()` operation that swaps elements k with $n-k-k$. Much of the time is spend waiting for memory which suggests that further optimizations would best be attempted with special (machine) instructions to bypass the cache or with non-temporal writes.

The routine [FXT: `revbin_permute0()` in perm/revbinpermute0.h] is a specialized version optimized for zero padded data. Some memory access can be avoided for that case. For example, revbin-pairs with both indices larger than $n/2$ need no processing at all. Therefore the routine is faster than the general version.

If, for complex data, one works with separate arrays for real and imaginary part[2] one might be tempted to do away with half of the bookkeeping as follows: write a special procedure `revbin_permute(a[],b[],n)` that shall replace the two successive calls `revbin_permute(a[],n)` and `revbin_permute(b[],n)` and after each statement `swap(a[x],a[r])` has inserted a `swap(b[x],b[r])`. If you do so, be prepared for disaster! Very likely the real and imaginary element for the same index lie apart in memory by a power of two, leading to one hundred percent cache miss for the typical computer. Even in the most favorable case the cache miss rate will be increased. Do expect to hardly ever win anything noticeable but in most cases to lose big. Think about it, whisper "*direct mapped cache*", and forget it.

2.2 The radix permutation

The *radix permutation* is the generalization of the revbin permutation to arbitrary radices. Pairs of elements are swapped when their indices, written in radix r are reversed. For example, in radix 10 and $n = 1000$ the elements with indices 123 and 321 will be swapped. The radix permutation is self-inverse.

C++ code for the radix r permutation of the array `f[]` is given in [FXT: file perm/radixpermute.h]. The routine must be called with n a perfect power of the radix r. Using radix $r = 2$ gives the revbin permutation.

```
extern ulong radix_permute_nt[];  // == 9, 90, 900, ...  for r=10
extern ulong radix_permute_kt[];  // == 1, 10, 100, ...  for r=10
#define  NT  radix_permute_nt
#define  KT  radix_permute_kt

template <typename Type>
void radix_permute(Type *f, ulong n, ulong r)
{
    ulong x = 0;
    NT[0] = r-1;
    KT[0] = 1;
    while ( 1 )
    {
        ulong z = KT[x] * r;
        if ( z>n )  break;
        ++x;
        KT[x] = z;

        NT[x] = NT[x-1] * r;
    }
    // here: n == p**x

    for (ulong i=0, j=0;  i < n-1;  i++)
    {
        if ( i<j )  swap2(f[i], f[j]);

        ulong t = x - 1;
```

[2] as opposed to: using a data type 'complex' with real and imaginary part of each number in consecutive places

```
        ulong k = NT[t];  // =^=  k = (r-1) * n / r;
        while ( k<=j )
        {
            j -= k;
            k = NT[--t];  // =^=  k /= r;
        }
        j += KT[t]; // =^=  j += (k/(r-1));
    }
}
```

2.3 In-place matrix transposition

Transposing a matrix is easy when it is not done in-place. The simple routine [FXT: `transpose()` in aux2/transpose.h] does the job:

```
template <typename Type>
void transpose(const Type * restrict f, Type * restrict g, ulong nr, ulong nc)
// Transpose nr x nc matrix f[]  into an nc x nr matrix g[].
{
    for (ulong r=0; r<nr; r++)
    {
        ulong isrc = r * nc;
        ulong idst = r;
        for (ulong c=0; c<nc; c++)
        {
            g[idst] = f[isrc];
            isrc += 1;
            idst += nr;
        }
    }
}
```

Matters get more complicated for the in-place equivalent. We have to find the cycles (see section 2.13 on page 99) of the underlying permutation. To transpose a $n_r \times n_c$- matrix first identify the position i of the entry in row r and column c:

$$i = r \cdot n_c + c \tag{2.2}$$

After the transposition the element will be at position i' in the transposed $n'_r \times n'_c$- matrix

$$i' = r' \cdot n'_c + c' \tag{2.3}$$

Obviously, $r' = c$, $c' = r$, $n'_r = n_c$ and $n'_c = n_r$, so:

$$i' = c \cdot n_r + r \tag{2.4}$$

Multiply the last equation by n_c

$$i' \cdot n_c = c \cdot n_r \cdot n_c + r \cdot n_c \tag{2.5}$$

With $n := n_r \cdot n_c$ and $r \cdot n_c = i - c$ we get

$$i' \cdot n_c = c \cdot n + i - c \tag{2.6}$$

$$i = i' \cdot n_c - c \cdot (n-1) \tag{2.7}$$

Take the equation modulo $n - 1$ to get

$$i \equiv i' \cdot n_c \mod (n-1) \tag{2.8}$$

That is, the transposition moves the element $i = i' \cdot n_c$ to position i'. Multiply by n_r to get the inverse:

$$i \cdot n_r \equiv i' \cdot n_c \cdot n_r \tag{2.9}$$

$$i \cdot n_r \equiv i' \cdot (n - 1 + 1) \tag{2.10}$$

$$i \cdot n_r \equiv i' \tag{2.11}$$

That is, element i will be moved to $i' = i \cdot n_r \mod (n-1)$.

The routine [FXT: `transpose()` in aux2/transpose.h] uses the a bit-array to keep track of the elements that have been processed so far:

```
#define  SRC(k)  (((unsigned long long)(k)*nc)%n1)

template <typename Type>
void transpose(Type *f, ulong nr, ulong nc, bitarray *ba=0)
// In-place transposition of an  nr X nc  array
//   that lies in contiguous memory.
{
    if ( 1>=nr )  return;
    if ( 1>=nc )  return;

    if ( nr==nc )  transpose_square(f, nr);
    else
    {
        const ulong n1 = nr * nc - 1;

        bitarray *tba = 0;
        if ( 0==ba )  tba = new bitarray(n1);
        else          tba = ba;
        tba->clear_all();

        for (ulong k=1;  k<n1;  k=tba->next_clear(++k) )  //  0 and n1 are fixed points
        {
            // do a cycle:
            ulong ks = SRC(k);
            ulong kd = k;
            tba->set(kd);
            Type t = f[kd];
            while ( ks != k )
            {
                f[kd] = f[ks];
                kd = ks;
                tba->set(kd);
                ks = SRC(ks);
            }
            f[kd] = t;
        }

        if ( 0==ba )  delete tba;
    }
}
```

Note that one should take care of possible overflows in the calculation $i \cdot n_c$. For the case that n is a power of two (and so are both n_r and n_c) the multiplications modulo $n-1$ are cyclic shifts. Thus any overflow can be avoided and the computation is also significantly cheaper. A C++ implementation is given in [FXT: file aux2/transpose2.h].

2.4 Revbin permutation and matrix transposition *

How would you rotate an (length-n) array by s positions (left or right), *without* using any scratch space. If you do not know the solution then try to find it before reading on.

The trick is to use `reverse()` three times as in the following [FXT: `rotate_left()` in perm/rotate.h]:

```
template <typename Type>
void rotate_left(Type *f, ulong n, ulong s)
// Rotate towards element #0
// Shift is taken modulo n
{
    if ( s==0 )  return;
    if ( s>=n )
    {
        if (n<2) return;
        s %= n;
    }

    reverse(f,   s);
    reverse(f+s, n-s);
    reverse(f,   n);
```

```
}
```

For example left-rotating an 8-element array by 3 positions is achieved by the following steps:

```
    [ 1 2 3 4 5 6 7 8 ]
    [ 3 2 1 4 5 6 7 8 ]  reverse first 3 elements
    [ 3 2 1 8 7 6 5 4 ]  reverse last 8-3=5 elements
    [ 4 5 6 7 8 1 2 3 ]  reverse whole array
```

Similarly for the other direction. A right rotation of an n-element array by s positions is identical to a left rotation by $n - s$ positions:

```
    [ 1 2 3 4 5 6 7 8 ]
    [ 5 4 3 2 1 6 7 8 ]  reverse first 8-3=5 elements
    [ 5 4 3 2 1 8 7 6 ]  reverse last 3 elements
    [ 6 7 8 1 2 3 4 5 ]  reverse whole array
```

```
template <typename Type>
void rotate_right(Type *f, ulong n, ulong s)
// Rotate away from element #0
// Shift is taken modulo n
{
    if ( s==0 )  return;
    if ( s>=n )
    {
        if (n<2) return;
        s %= n;
    }

    reverse(f,     n-s);
    reverse(f+n-s, s);
    reverse(f,     n);
}
```

What this has to do with our subject? When transposing an $n_r \times n_c$ matrix whose size is a power of two (thereby both n_r and n_c are also powers of two) the above mentioned rotation is done with the *indices* (written in base two) of the elements. We know how to do a permutation that reverses the complete indices and reversing a few bits at the least significant end is not any harder:

```
template <typename Type>
void revbin_permute_rows(Type *f, ulong ldn, ulong ldnc)
// Revbin-permute the length 2**ldnc rows  of f[0..2**ldn-1]
// (f[] considered as an 2**(ldn-ldnc) x 2**ldnc matrix)
{
    ulong n = 1UL<<ldn;
    ulong nc = 1UL<<ldnc;
    for (ulong k=0; k<n; k+=nc)  revbin_permute(f+k, nc);
}
```

And there we go:

```
template <typename Type>
void transpose_by_rbp(Type *f, ulong ldn, ulong ldnc)
// Transpose f[] considered as an 2**(ldn-ldnc) x 2**ldnc matrix
{
    revbin_permute_rows(f, ldn, ldnc);
    ulong n = 1UL<<ldn;
    revbin_permute(f, n);
    revbin_permute_rows(f, ldn, ldn-ldnc);  // ... that is, columns
}
```

The triple-reversion trick can also be used to swap two blocks in an array: First reverse the three ranges (first blocks, range between block, last block), then reverse the range that consists of all three. This is the *quadruple reversion trick*. The corresponding code is given in [FXT: file perm/swapblocks.h]:

```
template <typename Type>
void swap_blocks(Type *f, ulong x1, ulong n1, ulong x2, ulong n2)
// Swap the blocks starting at indices x1 and x2
// n1 and n2 are the block lengths
{
    if ( x1>x2 ) { swap2(x1,x2); swap2(n1,n2); }
    f += x1;
    x2 -= x1;
    ulong n = x2 + n2;
    reverse(f, n1);
    reverse(f+n1, n-n1-n2);
    reverse(f+x2, n2);
```

```
    reverse(f, n);
}
```

The elements before `x1` and after `x2+n2` are not accessed. An example [FXT: demo perm/swap-blocks-demo.cc]:

```
                v v v v v     v v v v         <--= want to swap these blocks
    [ 0 1 2 3 4  a b c d e 7 8 w x y z  N N ]  orig. data
    [ 0 1 2 3 4  e d c b a 7 8 w x y z  N N ]  reverse first block
    [ 0 1 2 3 4  e d c b a 8 7 w x y z  N N ]  reverse range between blocks
    [ 0 1 2 3 4  e d c b a 8 7 z y x w  N N ]  reverse second block
    [ 0 1 2 3 4  w x y z 7 8 a b c d e  N N ]  reverse whole range
                 ^ ^ ^ ^     ^ ^ ^ ^ ^         <--= the swapped blocks
```

The effect of `swap_blocks(f, x1, n1, x2, n2)` can be undone via `swap_blocks(f, x1, n2, x2+n2-n1, n1)`.

2.5 The zip permutation

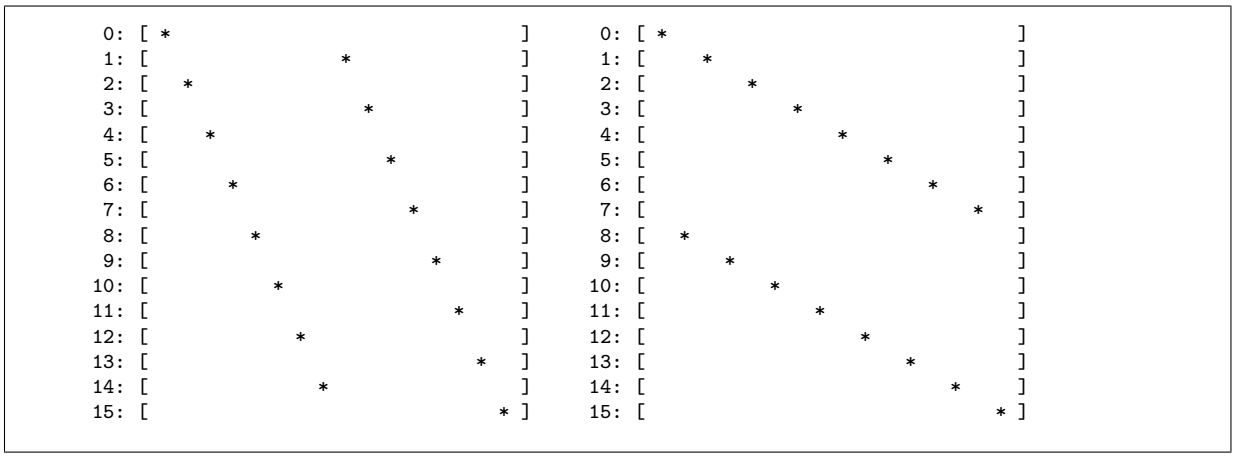


Figure 2.2: Permutation matrices of the zip permutation (left) and its inverse, the unzip permutation (right). The zip permutation moves the lower half of the array to the even indices, the upper half to the odd indices.

The *zip permutation* moves the elements form the lower half to the even indices and the elements of the upper half to the odd indices. Symbolically,

```
    [ a b c d A B C D ]  |--> [ a A b B c C d D ]
```

The size of the array must be even. A routine for the permutation is

```
template <typename Type>
void zip(const Type * restrict f, Type * restrict g, ulong n)
{
    ulong nh = n/2;
    for (ulong k=0,  k2=0;  k<nh;  ++k, k2+=2)  g[k2] = f[k];
    for (ulong k=nh, k2=1;  k<n;   ++k, k2+=2)  g[k2] = f[k];
}
```

When the array size is a power of two we can use a special case of the 'transposition by revbin permutation' idea to do the operation in-place [FXT: `zip()` in perm/zip.h]:

```
template <typename Type>
void zip(Type *f, ulong n)
{
    ulong nh = n/2;
    revbin_permute(f, nh);  revbin_permute(f+nh, nh);
    revbin_permute(f, n);
}
```

If we have a type `Complex` consisting of two `double`s lying contiguous in memory we can optimize the procedure:

```
void zip(double *f, long n)
{
    revbin_permute(f, n);
    revbin_permute((Complex *)f, n/2);
}
```

The inverse of the zip permutation is the *unzip permutation*. We only give the in-place version based on the revbin permutation. The array size must be a power of two:

```
template <typename Type>
void unzip(Type *f, ulong n)
{
    ulong nh = n/2;
    revbin_permute(f, n);
    revbin_permute(f, nh);  revbin_permute(f+nh, nh);
}
```

The routine can for the type `double` again be optimized as

```
void unzip(double *f, long n)
{
    revbin_permute((Complex *)f, n/2);
    revbin_permute(f, n);
}
```

For arrays whose size n is not a power of two the in-place zip permutation can be obtained by transposing the data as a $2 \times n/2$ matrix:

```
    transpose(f, 2, n/2);  // =^= zip(f, n)
```

The routines for in-place transposition are given in section 2.3 on page 84. The inverse is clearly obtained by transposing the data as a $n/2 \times 2$ matrix:

```
    transpose(f, n/2, 2);  // =^= unzip(f, n)
```

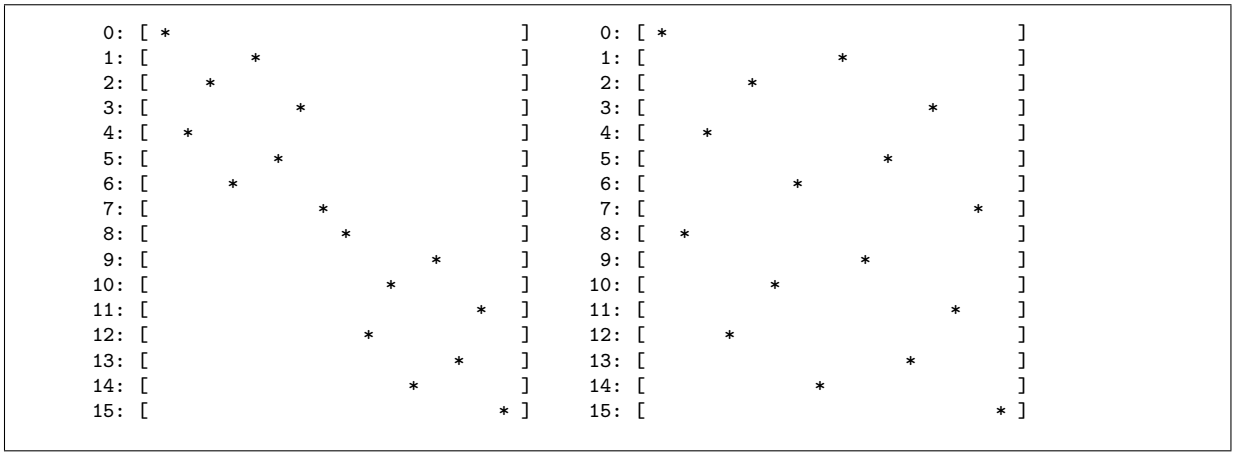


Figure 2.3: Revbin permutation matrices that, when multiplied together, give the zip permutation and its inverse. Let L and R be the permutations given on the left and right side, respectively. Then $Z = R\,L$ and $Z^{-1} = L\,R$.

While the above mentioned technique is usually *not* a gain for doing a transposition it may be used to speed up the revbin permutation itself. We operator-ize the idea to see how. Let R be the revbin permutation `revbin_permute()`, $T(n_r, n_c)$ the transposition of the $n_r \times n_c$ matrix and $R(n_c)$ the operation done by `revbin_permute_rows()` (see section 2.4 on page 85). Then

$$T(n_r, n_c) \quad = \quad R(n_r) \cdot R \cdot R(n_c) \tag{2.12}$$

The R-operators are their own inverses while T is in general not self inverse[3].

$$R \quad = \quad R(n_r) \cdot T(n_r, n_c) \cdot R(n_c) \tag{2.13}$$

There is a degree of freedom in this formula: for fixed $n = n_r \times n_c$ one can choose one of n_r and n_c (only their product is given).

2.6 The reversed zip permutation

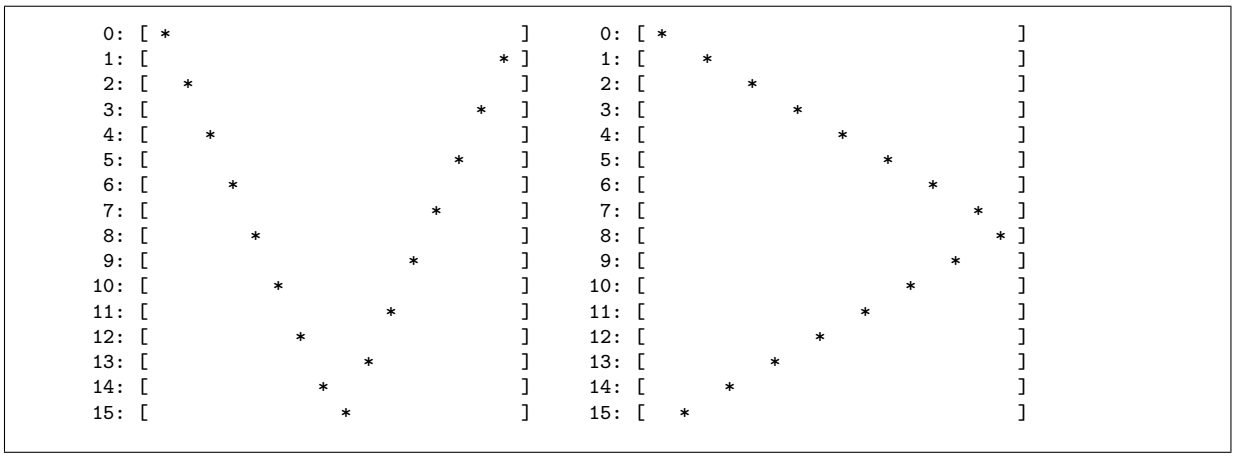


Figure 2.4: Permutation matrices of the reversed zip permutation (left) and its inverse (right).

A permutation closely related to the zip permutation is the *reversed zip permutation*. It moves the lower half of an array to the even indices and the upper half to the odd indices in reversed order. Symbolically,

```
[ a b c d A B C D ]  |--> [ a D b C c B d A ]
```

A C++ routine is [FXT: `zip_rev()` in perm/ziprev.h]:

```
template <typename Type>
void zip_rev(const Type * restrict x, Type * restrict y, ulong n)
// n must be even
{
    const ulong nh = n/2;
    for (ulong k=0, k2=0;     k<nh;  k++, k2+=2)  y[k2] = x[k];
    for (ulong k=nh, k2=n-1;  k<n;   k++, k2-=2)  y[k2] = x[k];
}
```

The in-place version can, if the array length is a power of two, be implemented as

```
template <typename Type>
void zip_rev(Type *x, ulong n)
// n must be a power of two
{
    const ulong nh = n/2;
    reverse(x+nh, nh);
    revbin_permute(x, nh); revbin_permute(x+nh, nh);
    revbin_permute(x, n);
}
```

The inverse permutation [FXT: `unzip_rev()` in perm/ziprev.h] can be implemented as

```
template <typename Type>
void unzip_rev(const Type * restrict x, Type * restrict y, ulong n)
// n must be even
{
    const ulong nh = n/2;
    for (ulong k=0, k2=0;     k<nh;  k++, k2+=2)  y[k] = x[k2];
```

[3] For $n_r = n_c$ it of course is.

```
    for (ulong k=nh, k2=n-1;  k<n;   k++, k2-=2)  y[k] = x[k2];
}
```

The in-place version is

```
template <typename Type>
void unzip_rev(Type *x, ulong n)
// n must be a power of two
{
    const ulong nh = n/2;
    revbin_permute(x, n);
    revbin_permute(x, nh); revbin_permute(x+nh, nh);
    reverse(x+nh, nh);
}
```

The given permutation is used in an algorithm where the cosine transform is computed using the Hartley transform, see section 12.11 on page 373.

We write Z and Z^{-1} for the zip permutation and its inverse, $\overline{Z}$ and $\overline{Z}^{-1}$ for the reversed zip permutation and its inverse, and R for the revbin permutation. Then the following relations hold:

$$Z = R\,Z^{-1}\,R = \overline{Z}\,Z^{-1}\,\overline{Z} \tag{2.14a}$$
$$\overline{Z} = R\,\overline{Z}^{-1}\,R = Z\,\overline{Z}^{-1}\,Z \tag{2.14b}$$
$$Z^{-1} = R\,Z\,R = \overline{Z}^{-1}\,Z\,\overline{Z}^{-1} \tag{2.14c}$$
$$\overline{Z}^{-1} = R\,\overline{Z}\,R = Z^{-1}\,\overline{Z}\,Z^{-1} \tag{2.14d}$$

2.7 The XOR permutation

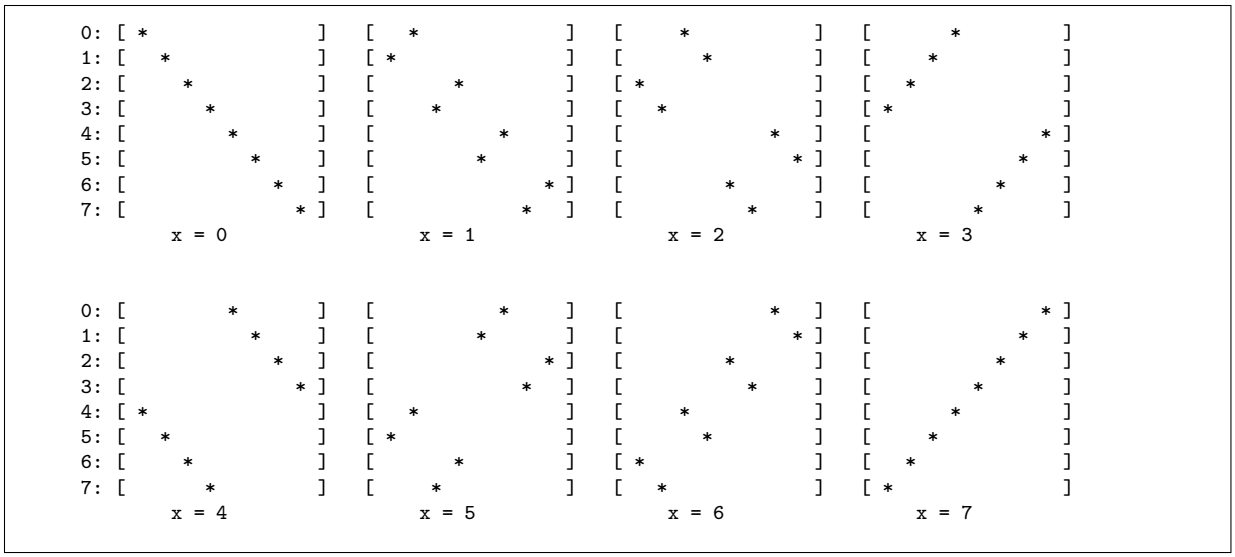


Figure 2.5: Permutation matrices of the XOR permutation for length 8 with parameter $x = 0 \ldots 7$. Compare to the table for the dyadic convolution (figure 10.4 on page 318).

The XOR permutation may be explained most simply by its trivial implementation: [FXT: `xor_permute()` in perm/xorpermute.h]:

```
template <typename Type>
void xor_permute(Type *f, ulong n, ulong x)
{
    if ( 0==x )  return;
    for (ulong k=0; k<n; ++k)
    {
        ulong r = k^x;
        if ( r>k )  swap2(f[r], f[k]);
```

```
    }
}
```

The XOR permutation is evidently self-inverse. The array length n must be divisible by the smallest power of two that is greater than x: for example, n must be even if $x = 1$, n must be divisible by four if $x = 2$ or $x = 3$. With n a power of two and $x < n$ one is on the safe side.

The XOR permutation contains a few other permutations as important special cases (for simplicity assume that the array length n is a power of two): when the third argument x equals $n-1$ then the permutation is the reversion, with $x = 1$ neighboring even and odd indexed elements are swapped, with $x = n/2$ the upper and the lower half of the array are swapped.

One has

$$X_a\, X_b \;\; = \;\; X_b\, X_a \;=\; X_c \quad \text{where} \quad c = a \text{ XOR } b \tag{2.15}$$

For the special case $a = b$ the relation expresses the self-inverse property as X_0 is the identity. The XOR permutation often occurs in relations between other permutations where we will use the symbol X_x, the subscript denoting the third argument.

2.8 The Gray code permutation

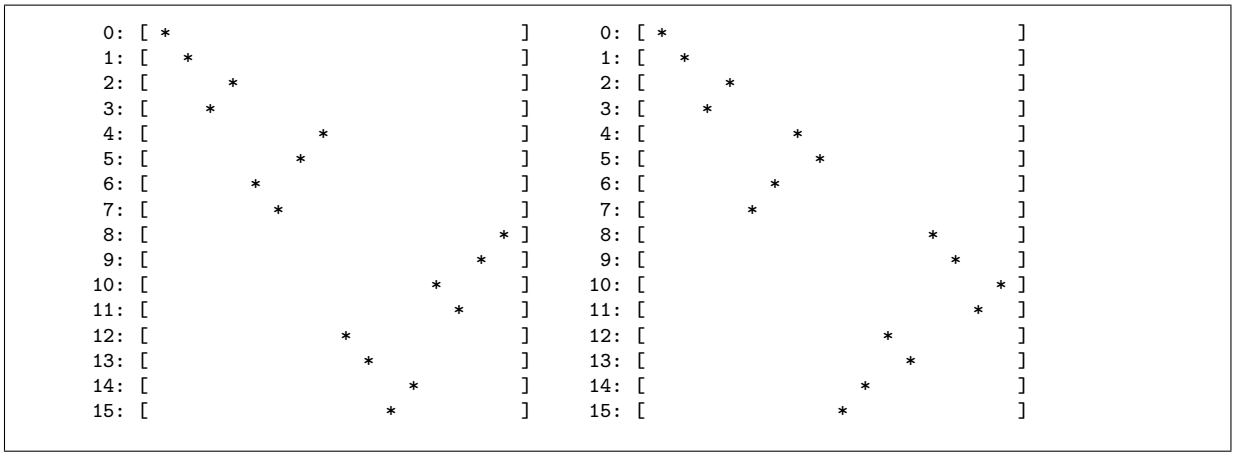


Figure 2.6: Permutation matrices of the Gray code permutation (left) and its inverse (right).

The Gray code permutation reorders (length-2^n) arrays according to the Gray code

```
static inline ulong gray_code(ulong x)
{
    return  x ^ (x>>1);
}
```

which is most easily demonstrated with the according routine that does not work in-place [FXT: file perm/graypermute.h]:

```
template <typename Type>
inline void gray_permute(const Type *f, Type * restrict g, ulong n)
// after this routine
// g[gray_code(k)] == f[k]
{
    for (ulong k=0; k<n; ++k)  g[gray_code(k)] = f[k];
}
```

Its inverse is

```
template <typename Type>
inline void inverse_gray_permute(const Type *f, Type * restrict g, ulong n)
// after this routine
```

```
// g[k] == f[gray_code(k)]
// (same as: g[inverse_gray_code(k)] == f[k])
{
    for (ulong k=0; k<n; ++k)  g[k] = f[gray_code(k)];
}
```

It also uses calls to `gray_code()` because they are cheaper than the computation of `inverse_gray_code()`, see section 1.16 on page 33.

We will write in-place versions of the above routines that offer very good performance. It is necessary to identify the cycle leaders (see section 2.13 on page 99) of the permutation and find an efficient way to generate them.

2.8.1 Cycles of the permutation

```
     0:  (  2,  3) #=2
     1:  (  4,  7,  5,  6) #=4
     2:  (  8, 15, 10, 12) #=4
     3:  (  9, 14, 11, 13) #=4
     4:  ( 16, 31, 21, 25, 17, 30, 20, 24) #=8
     5:  ( 18, 28, 23, 26, 19, 29, 22, 27) #=8
     6:  ( 32, 63, 42, 51, 34, 60, 40, 48) #=8
     7:  ( 33, 62, 43, 50, 35, 61, 41, 49) #=8
     8:  ( 36, 56, 47, 53, 38, 59, 45, 54) #=8
     9:  ( 37, 57, 46, 52, 39, 58, 44, 55) #=8
    10:  ( 64,127, 85,102, 68,120, 80, 96) #=8
    11:  ( 65,126, 84,103, 69,121, 81, 97) #=8
    12:  ( 66,124, 87,101, 70,123, 82, 99) #=8
    13:  ( 67,125, 86,100, 71,122, 83, 98) #=8
    14:  ( 72,112, 95,106, 76,119, 90,108) #=8
    15:  ( 73,113, 94,107, 77,118, 91,109) #=8
    16:  ( 74,115, 93,105, 78,116, 88,111) #=8
    17:  ( 75,114, 92,104, 79,117, 89,110) #=8
    126 elements in  18 nontrivial cycles.
    cycle lengths:   2 ...   8
   2 fixed points: [0. 1]
```

Figure 2.7: Cycles of the Gray code permutation of length 128.

It is instructive to study the complementary masks that occur for cycles (see section 2.13 on page 99) of different lengths. The cycles of the Gray code permutation for length 128 are shown in figure 2.7. No structure is immediately visible.

However, one can generate the cycle maxima as follows: for each range $2^k \ldots 2^{k+1}-1$ generate a bit-mask z that is obtained form the $k+1$ leftmost bits of the infinite word that has bits set at positions 0, 1, 2, 4, 8, ..., 2^i, ...:

```
[11101000100000001000000000000001000 ...]
```

An example: For $k=6$ we have $z=$`[1110100]`. Then take v to be $k+1$ leftmost bits of the complement, $v=$`[0001011]` in our example. Now the set of words $c=z+s$ where s is a subset of v contains exactly one element of each cycle in the range $2^k \ldots 2^{k+1} = 64 \ldots 127$:

```
117 =  .111.1.1
118 =  .111.11.
119 =  .111.111
124 =  .11111..
125 =  .11111.1
126 =  .111111.
127 =  .1111111
116 =  .111.1..
  maxima = z XOR subsets(v)  where
  z =  .111.1..
  v =  ....1.11
```

The words obtained are actually the cycle maxima.The list can be generated with the program [FXT: demo perm/permgray-leaders-demo.cc].

2.8.2 In-place routines

For the in-place versions of the permutation routines we use the bit-subset class described in section 1.28 on page 59. The forward version is [FXT: file perm/graypermute.h]:

```
template <typename Type>
void gray_permute(Type *f, ulong n)
{
    ulong z = 1; // mask for cycle maxima
    ulong v = 0; // ~z
    ulong cl = 1;  // cycle length
    for (ulong ldm=1, m=2;  m<n;  ++ldm, m<<=1)
    {
        z <<= 1;
        v <<= 1;
        if ( is_pow_of_2(ldm) )
        {
            ++z;
            cl <<= 1;
        }
        else  ++v;

        bit_subset b(v);
        do
        {
            // --- do cycle: ---
            ulong i = z | b.next();  // start of cycle
            Type t = f[i];           // save start value
            ulong g = gray_code(i);  // next in cycle
            for (ulong k=cl-1; k!=0; --k)
            {
                Type tt = f[g];
                f[g] = t;
                t = tt;
                g = gray_code(g);
            }
            f[g] = t;
            // --- end (do cycle) ---
        }
        while ( b.current() );
    }
}
```

The function `is_pow_of_2()` is described in section 1.7 on page 17. The inverse routine differs only in the block that processes the cycles:

```
template <typename Type>
void inverse_gray_permute(Type *f, ulong n)
{
    [--snip--]
            // --- do cycle: ---
            ulong i = z | b.next();  // start of cycle
            Type t = f[i];           // save start value
            ulong g = gray_code(i);  // next in cycle
            for (ulong k=cl-1; k!=0; --k)
            {
                f[i] = f[g];
                i = g;
                g = gray_code(i);
            }
            f[i] = t;
            // --- end (do cycle) ---
    [--snip--]
}
```

2.8.3 Performance of the routines

How fast is the Gray code permutation? We use the convention that the speed of the trivial (and completely cache-friendly, therefore running at memory bandwidth) `reverse()` is 1.0, our hereby declared time unit for comparison [FXT: file perm/reverse.h]. A little benchmark gives, for large (16 MB) arrays:

```
arg 1: 21 == ldn  [Using 2**ldn elements]  default=21
```

```
arg 2: 10 == rep  [Number of repetitions]  default=10
Memsize = 16384 kiloByte  ==  2097152 doubles
                reverse(f,n);    dt=  0.0103524    MB/s= 1546   rel=          1
         revbin_permute(f,n);    dt=  0.0674235    MB/s=  237   rel=    6.51282
        revbin_permute0(f,n);    dt=   0.061507    MB/s=  260   rel=    5.94131
           gray_permute(f,n);    dt=  0.0155019    MB/s= 1032   rel=    1.49742
    inverse_gray_permute(f,n);    dt=  0.0150641    MB/s= 1062   rel=    1.45512
                reverse(f,n);    dt=  0.0104008    MB/s= 1538   rel=    1.00467
```

We timed `reverse()` twice to get an impression how much we can trust the observed numbers.

While the revbin permutation takes about 6 units (due to its memory access pattern that is very problematic with respect to cache usage) the Gray code permutation needs only 1.50 units. The difference gets bigger for machines with relatively (to the CPU) slow memory. The Gray code permutation can be used to speed up fast transforms of large lengths a power of two, notably the Walsh transform, see chapter 10 on page 303.

The bandwidth of the `reverse()` is about 1500 MB/sec which should be compared to the output of a memory testing program, revealing that it actually runs at about the bandwidth of copying via a simple loop using pointers to `double`s:

```
    avg:   16777216  [ 0]"memcpy"                  2522.084 MB/s
    avg:   16777216  [ 1]"char *"                   471.873 MB/s
    avg:   16777216  [ 2]"short *"                  711.853 MB/s
    avg:   16777216  [ 3]"int *"                    956.682 MB/s
    avg:   16777216  [ 4]"long *"                  1514.360 MB/s
    avg:   16777216  [ 5]"long * (4x unrolled)"    1330.786 MB/s
    avg:   16777216  [ 6]"int64 *"                 1329.902 MB/s
    avg:   16777216  [ 7]"double *"                1329.507 MB/s  // <--=
    avg:   16777216  [ 8]"double * (4x unrolled)"  1325.437 MB/s
```

The relative speeds are quite different for small arrays. Using a size of 16 kB (2048 `double`s) we obtain

```
arg 1: 11 == ldn  [Using 2**ldn elements]  default=21
arg 2: 100000 == rep  [Number of repetitions]  default=512
Memsize = 16 kiloByte  ==  2048 doubles
                reverse(f,n);    dt=1.88726e-06    MB/s= 8279   rel=          1
         revbin_permute(f,n);    dt=3.22166e-06    MB/s= 4850   rel=    1.70706
        revbin_permute0(f,n);    dt=2.69212e-06    MB/s= 5804   rel=    1.42647
           gray_permute(f,n);    dt=4.75155e-06    MB/s= 3288   rel=    2.51769
    inverse_gray_permute(f,n);    dt=3.69237e-06    MB/s= 4232   rel=    1.95647
                reverse(f,n);    dt=1.88833e-06    MB/s= 8275   rel=    1.00057
```

due to the small size, the cache problems are gone.

The memory benchmark gives for that size

```
    avg:      16384  [ 0]"memcpy"                  3290.353 MB/s
    avg:      16384  [ 1]"char *"                   572.922 MB/s
    avg:      16384  [ 2]"short *"                  973.552 MB/s
    avg:      16384  [ 3]"int *"                   1495.920 MB/s
    avg:      16384  [ 4]"long *"                  3560.506 MB/s
    avg:      16384  [ 5]"long * (4x unrolled)"    3220.792 MB/s
    avg:      16384  [ 6]"int64 *"                 2498.137 MB/s
    avg:      16384  [ 7]"double *"                2498.285 MB/s  // <--=
    avg:      16384  [ 8]"double * (4x unrolled)"  3219.784 MB/s
```

2.9 The reversed Gray code permutation

If the length-n array is permuted in the way the upper half of the length-$2n$ array would be permuted by `gray_permute()` then all cycles are of the same length. The resulting permutation is equivalent to the reversed Gray code permutation:

```
template <typename Type>
inline void gray_rev_permute(const Type *f, Type * restrict g, ulong n)
// gray_rev_permute() =^=
```

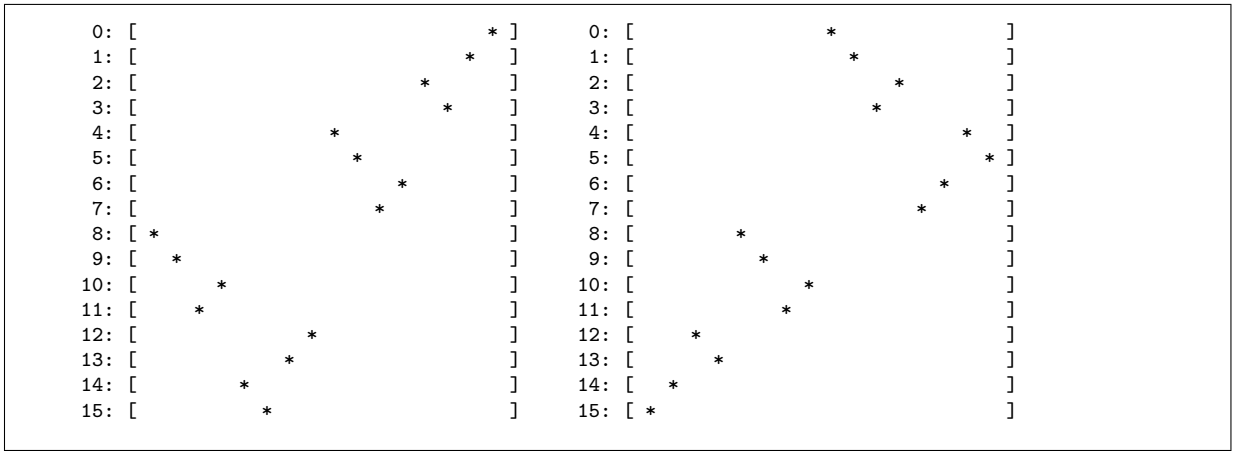


Figure 2.8: Permutation matrices of the reversed Gray code permutation (left) and its inverse (right).

```
//  { reverse(); gray_permute(); }
{
    for (ulong k=0, m=n-1;  k<n;  ++k, --m)  g[gray_code(m)] = f[k];
}
```

The routine, its inverse and in-place versions are given in [FXT: file perm/grayrevpermute.h].

All cycles have the same length, `gray_rev_permute(f, 64)` gives:

```
0:  (  0, 63, 21, 38,  4, 56, 16, 32) #=8
1:  (  1, 62, 20, 39,  5, 57, 17, 33) #=8
2:  (  2, 60, 23, 37,  6, 59, 18, 35) #=8
3:  (  3, 61, 22, 36,  7, 58, 19, 34) #=8
4:  (  8, 48, 31, 42, 12, 55, 26, 44) #=8
5:  (  9, 49, 30, 43, 13, 54, 27, 45) #=8
6:  ( 10, 51, 29, 41, 14, 52, 24, 47) #=8
7:  ( 11, 50, 28, 40, 15, 53, 25, 46) #=8
64 elements in   8 nontrivial cycles.
cycle length is == 8
No fixed points.
```

If 64 is added to the indices then the cycles in the upper half of the array as in `gray_permute(f, 128)` are reproduced (by construction).

Let G denote the Gray code permutation, $\overline{G}$ the reversed Gray code permutation. Symbolically one can write

$$G(n) = \{\dots, \overline{G}(n/8), \overline{G}(n/4), \overline{G}(n/2)\} \tag{2.16a}$$
$$G^{-1}(n) = \{\dots, \overline{G}^{-1}(n/8), \overline{G}^{-1}(n/4), \overline{G}^{-1}(n/2)\} \tag{2.16b}$$

Now let r the reversion and h the permutation that swaps the upper and the lower half of an array. Then

$$\overline{G} = G\,r = h\,G \tag{2.17a}$$
$$\overline{G}^{-1} = r\,G^{-1} \tag{2.17b}$$
$$\overline{G}^{-1}\,G = G^{-1}\,\overline{G} = r = X_{n-1} \tag{2.17c}$$
$$G\,\overline{G}^{-1} = \overline{G}\,G^{-1} = h = X_{n/2} \tag{2.17d}$$

Throughout it is assumed that the array length n is a power of two.

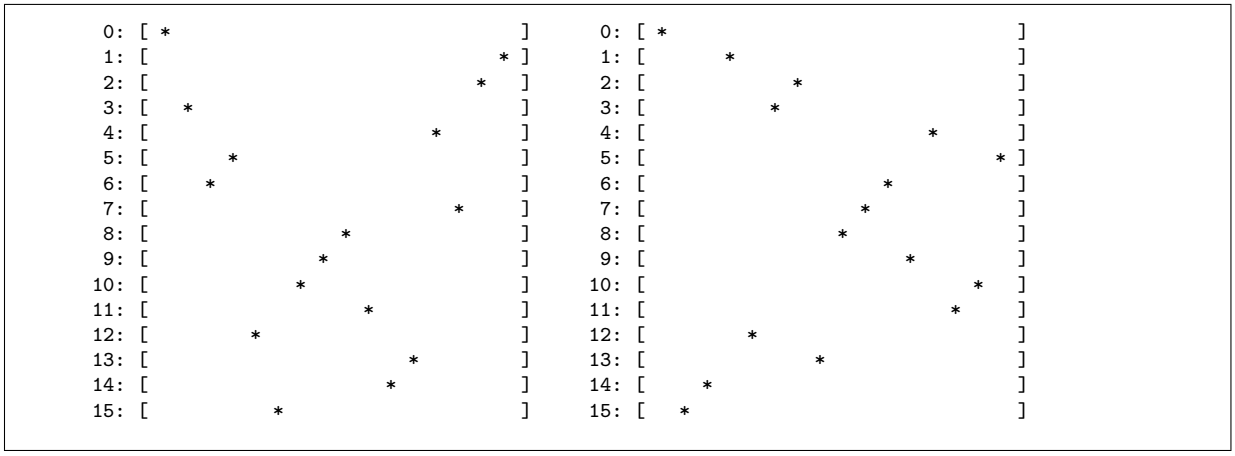


Figure 2.9: Permutation matrices of the green code permutation (left) and its inverse (right).

2.10 The green code permutation

The green code permutation is obtained by replacing the calls to `gray_code` by `green_code` in the Gray code permutation. Additionally, the highest bit must be masked out in order to keep indices in the range [FXT: `green_permute()` in perm/greenpermute.h]:

```
template <typename Type>
inline void green_permute(const Type *f, Type * restrict g, ulong n)
{
    for (ulong k=0;  k<n;  ++k)
    {
        ulong r = green_code(k);
        r &= (n-1);
        g[r] = f[k];
    }
}
```

The inverse is

```
template <typename Type>
inline void inverse_green_permute(const Type *f, Type * restrict g, ulong n)
{
    for (ulong k=0;  k<n;  ++k)
    {
        ulong r = green_code(k);
        r &= (n-1);
        g[k] = f[r];
    }
}
```

2.11 The reversed green code permutation

The 'reversed' version of the green permutation equivalent to array reversal followed by a green permutation. The permutation can be implemented as [FXT: file perm/greenrevpermute.h]:

```
template <typename Type>
inline void green_rev_permute(const Type *f, Type * restrict g, ulong n)
{
    for (ulong k=0, m=n-1;  k<n;  ++k, --m)
    {
        ulong r = green_code(m);
        r &= (n-1);
        g[r] = f[k];
    }
}
```

The inverse routine is

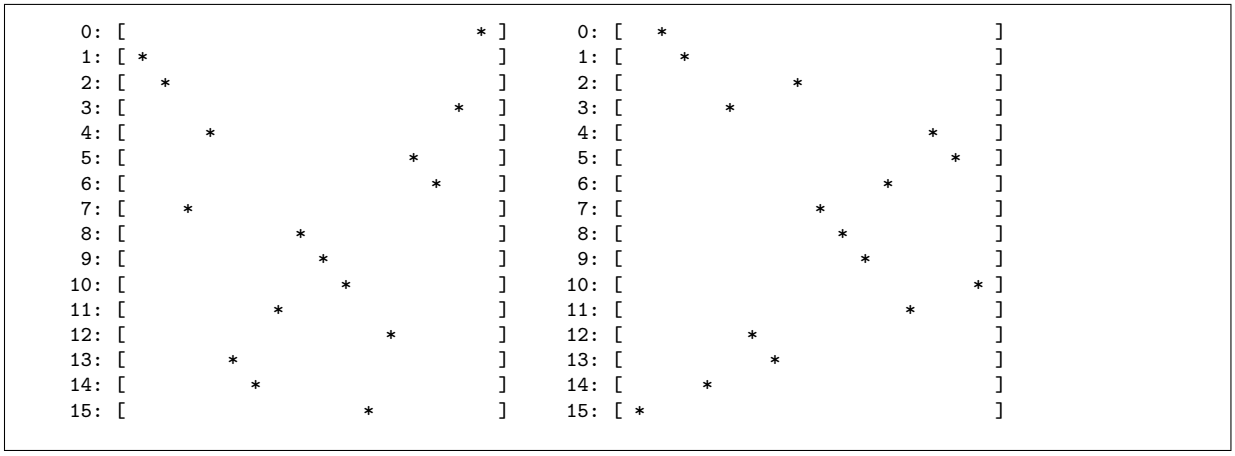


Figure 2.10: Permutation matrices of the reversed green code permutation (left) and its inverse (right).

```
template <typename Type>
inline void inverse_green_rev_permute(const Type *f, Type * restrict g, ulong n)
{
    for (ulong k=0, m=n-1;  k<n;  ++k, --m)
    {
        ulong r = green_code(m);
        r &= (n-1);
        g[k] = f[r];
    }
}
```

Let E denote the green code permutation, $\overline{E}$ the reversed green code permutation and R the revbin permutation. Further assume that the array length n is a power of two. Then

$$E = R\,G\,R \tag{2.18a}$$
$$E^{-1} = R\,G^{-1}\,R \tag{2.18b}$$
$$\overline{E} = R\,\overline{G}\,R \tag{2.18c}$$
$$\overline{E}^{-1} = R\,\overline{G}^{-1}\,R \tag{2.18d}$$

The relations between E and $\overline{E}$ are

$$\overline{E} = E\,r = X_1\,E \tag{2.19a}$$
$$\overline{E}^{-1} = r\,E^{-1} = E^{-1}\,X_1 \tag{2.19b}$$

and

$$\overline{E}^{-1}\,E = E^{-1}\,\overline{E} = r = X_{n-1} \tag{2.20a}$$
$$E\,\overline{E}^{-1} = \overline{E}\,E^{-1} = X_1 \tag{2.20b}$$

where X_x is the XOR permutation (with third argument equal to the subscript x, see section 2.7 on page 90). Now let $\overline{Z}$ be the reversed zip permutation described in section 2.6 on page 89. Then

$$\overline{Z} = G^{-1}\,E \tag{2.21a}$$
$$\overline{Z}^{-1} = E^{-1}\,G \tag{2.21b}$$

2.12 Decomposing permutations

In this section we will see some algorithms that use a certain type of decomposition (factorization in term of matrices) of some of the permutations we have studied so far. The resulting algorithms involve

proportional $n \cdot \log(n)$ computations for length-n arrays. This might seem to render the schemes worthless as one can always obtain a permutation with work proportional to n. There are, however, situations where one can use the algorithms advantageously. Firstly, with bit manipulations, where the whole binary words are modified in one or a few statements. The corresponding algorithms are therefore only proportional $\log(n)$. Secondly, when the algorithm can be used implicitly in order to integrate the permutation in a fast transform. The work can sometimes be reduced to zero in that case.

Array reversal

The most simple example might be the reversion via

```
template <typename Type>
void perm1(Type *f, ulong n)
// From shorter to longer sub-arrays.
{
    for (ulong k=2; k<=n; k*=2)
    {
        for (ulong j=0; j<n; j+=k)  func(f+j, k);
    }
}
```

where `func()` swaps the upper and lower half of an array:

```
template <typename Type>
void func(Type *f, ulong ldn)  { swap(f, f+n/2, n/2); }
```

This idea has been exploited in section 1.14 on page 28 in order to obtain a bit-reversal routine. The reversal is a self-inverse operation. Therefore one can alternatively execute the steps in reverse order and still get the same permutation:

```
template <typename Type>
void perm2(Type *f, ulong n)
// From longer to shorter sub-arrays.
{
    for (ulong k=n; k>=2; k/=2) // k == n, n/2, n/4, ... , 4, 2
    {
        for (ulong j=0; j<n; j+=k)  func(f+j, k);
    }
}
```

Note that `func()` in turn can be obtained via

```
    reverse(f, n);
    reverse(f, n/2);  reverse(f+n/2, n/2);
```

or the same statements in reversed order.

Gray code permutation

Let us try a less obvious example, use `perm1()` with `func()` defined to swap the third and fourth quarter:

```
    swap(f+n/2, f+n/2+n/4, n/4);  // quarters: [0,1,2,3]-->[0,1,3,2]
```

The resulting permutation is the Gray permutation. The other way round (using `perm2()`) one obtains the inverse Gray permutation.

Using `func()` defined as

```
    reverse(f+n/2, n/2);
```

one gets the Gray permutation through `perm2()`, its inverse through `perm1()`. This idea has been used in the core-routine for the sequency-ordered Walsh transform described in section 10.8 on page 320. The work for the Gray permutation has been completely vaporized there.

Note that the routine that swaps the halves of the upper half array could be obtained as either of

```
    inverse_gray_permute(f, n);
    gray_permute(f, n/2);     gray_permute(f+n/2, n/2);
```

or

```
inverse_gray_permute(f, n/2);  inverse_gray_permute(f+n/2, n/2);
gray_permute(f, n);
```

Similarly, the routine that reverses the upper half can be obtained as either of

```
gray_permute(f, n/2);     gray_permute(f+n/2, n/2);
inverse_gray_permute(f, n);
```

or

```
gray_permute(f, n);
inverse_gray_permute(f, n/2);  inverse_gray_permute(f+n/2, n/2);
```

The corresponding routines to Gray-permute the bits of a binary word are given in [FXT: file bits/bitgraypermute.h].

Zip and revbin permutation

Using `func()` defined to swap the second and third quarter

```
swap(f+n/4, f+n/2, n/4);  // quarters: [0,1,2,3]-->[0,2,1,3]
```

Then with `perm2()` one gets the zip permutation, `perm1()` gives the inverse. This idea has been used for the bit-wise zip shown in section 1.15 on page 32.

Using `func()` to cycle the second, third and fourth quarter:

```
// quarters: [0,1,2,3]-->[0,2,3,1]
```

which was obtained using

```
zip_rev(f, n);
unzip_rev(f, n/2);  unzip_rev(f+n/2, n/2);
```

both the reversed zip permutation and its inverse can be computed in the now hopefully obvious way.

The revbin permutation can be generated through the zip permutation or its inverse. However, the zip permutation is the more complicate one, so absorbing the revbin permutation into fast transforms does not seem to be easy. The other way round it makes more sense:

```
revbin_permute(f, n/2);  revbin_permute(f+n/2, n/2);
revbin_permute(f, n);
```

Is a convenient (though not the most effective) way to compute the zip permutation.

Clearly, the idea presented here is in analogy with the decomposition of linear transforms. Finally the permutations are (very simple forms of) linear transforms. See also section 1.34 on page 74.

2.13 General permutations

So far we treated special permutations that occurred as part of other algorithms. It is instructive to study permutations in general with the operations (as composition and inversion) on them.

2.13.1 Basic definitions and operations

A straightforward way to represent a permutation is to consider the array of indices that for the original (unpermuted) data would be the length-n *canonical sequence* 0, 1, 2, ..., $n-1$. The mentioned trivial sequence represents the 'do-nothing' permutation or *identity*. The concept is best described by the routine that *applies* a given permutation x on an array of data f: after the routine has finished the array g will contain the elements of f reordered according to x [FXT: `apply_permutation()` in perm/permapply.h]:

```
template <typename Type>
void apply_permutation(const ulong *x, const Type *f, Type * restrict g, ulong n)
// Apply the permutation x[] on f[]
// i.e. set g[k] <-- f[x[k]]  \forall k
{
    for (ulong k=0; k<n; ++k)  g[k] = f[x[k]];
}
```

An example using strings (arrays of characters): The permutation represented by

$x =$[7 6 3 2 5 1 0 4]

and the input data

$f =$[A B a d C a f e] would produce

$g =$[e f d a a B A C]

Routines that test various properties of permutations are given in [FXT: file perm/permq.h]. To check whether a given permutation is the identity is trivial:

```
bool is_identity(const ulong *f, ulong n)
// Return whether f[] is the identical permutation,
// i.e. whether f[k]==k for all k= 0...n-1
{
    for (ulong k=0; k<n; ++k)  if ( f[k] != k )  return false;
    return true;
}
```

A fixed point of a permutation is an index where the element is not moved:

```
ulong count_fixed_points(const ulong *f, ulong n)
// Return number of fixed points in f[]
{
    ulong ct = 0;
    for (ulong k=0; k<n; ++k)  if ( f[k] == k )  ++ct;
    return ct;
}
```

A *derangement* is a permutation that has no fixed points. To check whether a permutation is a derangement of identity use:

```
bool is_derangement(const ulong *f, ulong n)
// Return whether f[] is a derangement of identity,
// i.e. whether f[k]!=k for all k
{
    for (ulong k=0; k<n; ++k)  if ( f[k] == k )  return false;
    return true;
}
```

Whether two arrays are mutual derangements can be determined by:

```
bool is_derangement(const ulong *f, const ulong *g, ulong n)
// Return whether f[] is a derangement of g[],
// i.e. whether f[k]!=g[k] for all k
{
    for (ulong k=0; k<n; ++k)  if ( f[k] == g[k] )  return false;
    return true;
}
```

To check whether a given array really describes a valid permutation one has to verify that each index in the valid range appears exactly once. The `bitarray` class described in section 4.6 on page 145 allows us to do the job without modification of the input:

```
bool
is_valid_permutation(const ulong *f, ulong n, bitarray *bp/*=0*/)
// Return whether all values 0...n-1 appear exactly once,
// i.e. whether f represents a permutation of [0,1,...,n-1].
{
    // check whether any element is out of range:
    for (ulong k=0; k<n; ++k)  if ( f[k]>=n )  return false;

    // check whether values are unique:
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    ulong k;
    for (k=0; k<n; ++k)
```

```
    {
        if ( tp->test_set(f[k]) )  break;
    }
    if ( 0==bp )  delete tp;
    return  (k==n);
}
```

2.13.2 Compositions of permutations

One can apply several permutations to an array, one by one. The resulting permutation is called the *composition* of the applied permutations. The routines are given in [FXT: file perm/permq.cc]. As an example, the check whether some permutation g is equal to f applied twice, or f *squared*, use:

```
bool is_square(const ulong *f, const ulong *g, ulong n)
// Return whether f * f == g  as a permutation
{
    for (ulong k=0; k<n; ++k)  if ( g[k] != f[f[k]] )  return false;
    return true;
}
```

Note that in general $f \cdot g \neq g \cdot f$ for $f \neq g$, the operation of composition is not commutative.

A permutation f is said to be the *inverse* of another permutation g if it undoes its effect, that is $f \cdot g = \mathrm{id}$:

```
bool is_inverse(const ulong *f, const ulong *g, ulong n)
// Return whether f[] is the inverse of g[]
{
    for (ulong k=0; k<n; ++k)  if ( f[g[k]] != k )  return false;
    return true;
}
```

One has $g \cdot f = f \cdot g = \mathrm{id}$, in a group the left-inverse is equal to the right-inverse and we can simply call g 'the inverse' of f.

A permutation that is its own inverse (like the revbin permutation) is called an *involution*. Checking that is easy:

```
bool is_involution(const ulong *f, ulong n)
// Return whether max cycle length is <= 2,
// i.e. whether f * f = id.
{
    for (ulong k=0; k<n; ++k)  if ( f[f[k]] != k )  return false;
    return true;
}
```

The following routine computes the inverse of a given permutation [FXT: file perm/perminvert.cc]:

```
void make_inverse(const ulong *f, ulong * restrict g, ulong n)
// Set (as permutation) g to the inverse of f
{
    for (ulong k=0; k<n; ++k)  g[f[k]] = k;
}
```

2.13.3 Representation as disjoint cycles

If one wants to do the operation in-place a little bit of thought is required. The idea underlying all subsequent routines working in-place is that every permutation entirely consists of disjoint cycles. A *cycle* (of a permutation) is a subset of the indices that is rotated (by one) by the permutation. The term *disjoint* means that the cycles do not 'cross' each other. While this observation is pretty trivial it allows us to do many operations by following the cycles of the permutation, one by one, and doing the necessary operation on each of them. As an example consider the following permutation of an array originally consisting of the (canonical) sequence 0, 1, ..., 15. Extra spaces are inserted for readability:

```
[ 0, 1, 3, 2,   7, 6, 4, 5,   15, 14, 12, 13,   8, 9, 11, 10 ]
```

There are two fixed points (0 and 1) and these cycles:

```
( 2 <--  3 )
( 4 <--  7 <--  5 <--  6 )
( 8 <-- 15 <-- 10 <-- 12 )
( 9 <-- 14 <-- 11 <-- 13 )
```

The cycles do 'wrap around', for example, the initial 4 of the second cycle goes to position 6, the last element of the second cycle.

Note that the inverse permutation could formally be described by reversing every arrow in each cycle:

```
( 2 -->  3 )
( 4 -->  7 -->  5 -->  6 )
( 8 --> 15 --> 10 --> 12 )
( 9 --> 14 --> 11 --> 13 )
```

Equivalently, one can reverse the order of the elements in each cycle:

```
( 3 <--  2 )
( 6 <--  5 <--  7 <--  4 )
(12 <-- 10 <-- 15 <--  8 )
(13 <-- 11 <-- 14 <--  9 )
```

If we begin each cycle with its smallest element the inverse permutation looks like:

```
( 2 <--  3 )
( 4 <--  6 <--  5 <--  7 )
( 8 <-- 12 <-- 10 <-- 15 )
( 9 <-- 13 <-- 11 <-- 14 )
```

The last three sets of cycles all describe the same permutation:

```
[ 0, 1, 3, 2,   6, 7, 5, 4,   12, 13, 15, 14,   10, 11, 9, 8 ]
```

The code [FXT: `print_cycles()` in perm/printperm.cc] used to print the cycles in the above example is

```
ulong print_cycles(const ulong *f, ulong n, bitarray *bp=0)
// Print the cycles of the permutation.
// Return number of fixed points.
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    ulong ct = 0;  // # of fixed points
    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);

        // follow a cycle:
        ulong i = k;
        ulong g = f[i];  // next index
        if ( g==i )  // fixed point ?
        {
            ++ct;
            continue;
        }

        cout << "(" << setw(3) << i;
        while ( 0==(tp->test_set(g)) )
        {
            cout << " <-- " << setw(3) << g;
            g = f[g];
        }
        cout << " )" << endl;
    }

    if ( 0==bp )  delete tp;

    return  ct;
}
```

The bit-array (see section 4.6 on page 145 for the implementation) is used to keep track of the elements already processed.

A utility class to compute the decomposition of a permutation into cycles is [FXT: `class cycles` in perm/cycles.h]. A program that shows its usage is [FXT: demo perm/cycles-demo.cc], it prints the cycles and C-code for a permutation of given size:

```
Using: gray_permute(y, n)
Computing cycles:
  0:  (  2,  3) #=2
  1:  (  4,  7,  5,  6) #=4
  2:  (  8, 15, 10, 12) #=4
  3:  (  9, 14, 11, 13) #=4
  14 elements in   4 nontrivial cycles.
 cycle lengths:   2 ...   4
number of fixed points = 2
```

```
template <typename Type>
inline void foo_perm_16(Type *f)
// unrolled version for length 16
{
  swap2(f[2], f[3]);
  { Type t=f[4]; f[4]=f[7]; f[7]=f[5]; f[5]=f[6]; f[6]=t; }
  { Type t=f[8]; f[8]=f[15]; f[15]=f[10]; f[10]=f[12]; f[12]=t; }
  { Type t=f[9]; f[9]=f[14]; f[14]=f[11]; f[11]=f[13]; f[13]=t; }
}
```

2.13.4 Inverse and square, in-place

For the computation of the inverse we have to reverse each cycle [FXT: file perm/perminvert.cc]:

```
void make_inverse(ulong *f, ulong n, bitarray *bp/*=0*/)
// Set (as permutation) f to its own inverse.
// In-place version.
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);

        // invert a cycle:
        ulong i = k;
        ulong g = f[i];  // next index
        while ( 0==(tp->test_set(g)) )
        {
            ulong t = f[g];
            f[g] = i;
            i = g;
            g = t;
        }
        f[g] = i;
    }

    if ( 0==bp )  delete tp;
}
```

The extra array of tag bits can be avoided by using the highest bit of each word as tag bit. The scheme would fail if any word of the permutation array had the highest bit set. However, on byte-addressable machines such an array will not fit into memory at all (for word sizes of 16 or more bits). To keep the code similar to the version using the bit-array we define

```
static const ulong s1 = 1UL << (BITS_PER_LONG - 1);  // highest bit is tag bit
static const ulong s0 = ~s1;  // all bits but tag bit

static inline void SET(ulong *f, ulong k)  { f[k&s0] |= s1; }
static inline void CLEAR(ulong *f, ulong k)  { f[k&s0] &= s0; }
static inline bool TEST(ulong *f, ulong k)  { return (0!=(f[k&s0]&s1)); }
```

Note that we have to mask out the tag-bit when using the value 'k' as index. The routine can then be implemented as

```
void
make_inverse(ulong *f, ulong n)
// Set (as permutation) f to its own inverse.
// In-place version using highest bits of array as tag-bits.
{
    for (ulong k=0; k<n; ++k)
```

```
    {
        if ( TEST(f, k) ) { CLEAR(f, k);  continue; }  // already processed
        SET(f, k);

        // invert a cycle:
        ulong i = k;
        ulong g = f[i];  // next index
        while ( 0==TEST(f, g) )
        {
            ulong t = f[g];
            f[g] = i;
            SET(f, g);
            i = g;
            g = t;
        }
        f[g] = i;

        CLEAR(f, k);  // leave no tag bits set
    }
}
```

The extra `CLEAR()` statement at the end removes the tag-bit of the cycle minima. Its effect is that no tag-bits are set after routine has finished. It turns out that the routine has about the same performance as the bit-array version.

For the routine [FXT: file perm/permcompose.cc]

```
void make_square(const ulong *f, ulong * restrict g, ulong n)
// Set (as permutation) g = f * f
{
    for (ulong k=0; k<n; ++k)  g[k] = f[f[k]];
}
```

we obtain the following in-place version:

```
void make_square(ulong *f, ulong n, bitarray *bp/*=0*/)
// Set (as permutation) f = f * f
// In-place version.
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);

        // square a cycle:
        ulong i = k;
        ulong t = f[i];  // save
        ulong g = f[i];  // next index
        while ( 0==(tp->test_set(g)) )
        {
            f[i] = f[g];
            i = g;
            g = f[g];
        }
        f[i] = t;
    }

    if ( 0==bp )  delete tp;
}
```

2.13.5 Powers of a permutation

The e-th power of a permutation f is computed (and returned in g) by a version of the binary exponentiation algorithm described in section 14.6 on page 395. The routine is given in [FXT: file perm/permcompose.cc]:

```
void
power(const ulong *f, ulong * restrict g, ulong n, long e,
      ulong * restrict t/*=0*/)
```

```
// Set (as permutation) g = f ** e
{
    if ( e==0 )
    {
        for (ulong k=0; k<n; ++k)  g[k] = k;
        return;
    }

    if ( e==1 )
    {
        copy(f, g, n);
        return;
    }

    if ( e==-1 )
    {
        make_inverse(f, g, n);
        return;
    }

    // here:  abs(e) > 1
    ulong x = e>0 ? e : -e;

    if ( is_pow_of_2(x) )  // special case x==2^n
    {
        make_square(f, g, n);
        while ( x>2 )  { make_square(g, n);  x /= 2; }
    }
    else
    {
        ulong *tt = t;
        if ( 0==t )  { tt = new ulong[n]; }
        copy(f, tt, n);

        int firstq = 1;
        while ( 1 )
        {
            if ( x&1 )  // odd
            {
                if ( firstq )  // avoid multiplication by 1
                {
                    copy(tt, g, n);
                    firstq = 0;
                }
                else  compose(tt, g, n);

                if ( x==1 )   goto dort;
            }

            make_square(tt, n);
            x /= 2;
        }

    dort:
        if ( 0==t )  delete [] tt;
    }

    if ( e<0 )  make_inverse(g, n);
}
```

2.13.6 Applying permutations to data, in-place

The in-place analogue of the routine

```
template <typename Type>
void apply_permutation(const ulong *x, const Type *f, Type * restrict g, ulong n)
// Apply the permutation x[] to the array f[]
// i.e. set g[k] <-- f[x[k]]  \forall k
{
    for (ulong k=0; k<n; ++k)  g[k] = f[x[k]];
}
```

is [FXT: file perm/permapply.h]:

```
template <typename Type>
void apply_permutation(const ulong *x, Type * restrict f, ulong n, bitarray *bp=0)
// Apply the permutation x[] to the array f[]
// i.e. set f[k] <-- f[x[k]]  \forall k
```

```
// In-place version.
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);

        // --- do cycle: ---
        ulong i = k;  // start of cycle
        Type t = f[i];
        ulong g = x[i];
        while ( 0==(tp->test_set(g)) )  // cf. inverse_gray_permute()
        {
            f[i] = f[g];
            i = g;
            g = x[i];
        }
        f[i] = t;
        // --- end (do cycle) ---
    }

    if ( 0==bp )  delete tp;
}
```

Often one wants to apply the inverse of a permutation without actually inverting the permutation itself. This leads to

```
template <typename Type>
void apply_inverse_permutation(const ulong *x, const Type *f, Type * restrict g, ulong n)
// Apply the inverse permutation of x[] to the array f[],
// i.e. set g[x[k]] <-- f[k]  \forall k
{
    for (ulong k=0; k<n; ++k)  g[x[k]] = f[k];
}
```

whereas the in-place version is

```
template <typename Type>
void apply_inverse_permutation(const ulong *x, Type * restrict f, ulong n, bitarray *bp=0)
// Apply the inverse permutation of x[] to the array f[]
// i.e. set f[x[k]] <-- f[k]  \forall k
// In-place version.
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);

        // --- do cycle: ---
        ulong i = k;  // start of cycle
        Type t = f[i];
        ulong g = x[i];
        while ( 0==(tp->test_set(g)) )  // cf. gray_permute()
        {
            Type tt = f[g];
            f[g] = t;
            t = tt;
            g = x[g];
        }
        f[g] = t;
        // --- end (do cycle) ---
    }

    if ( 0==bp )  delete tp;
}
```

When a permutation of the set $S := \{0, 1, \ldots, n-1\}$ is given as a function $X(.)$ (where $X(k) \in S \; \forall \, k \in S$) the permutation can be applied on an array f via [FXT: `apply_permutation()` in perm/permapplyfunc.h]:

```
template <typename Type>
void apply_permutation(ulong (*x)(ulong), const Type *f, Type * restrict g, ulong n)
// Set g[k] <-- f[x(k)]  \forall k
// Must have:  0<=x(k)<n \forall k
{
    for (ulong k=0; k<n; ++k)  g[k] = f[x(k)];
}
```

For example, the statement `apply_permutation(inverse_gray_code, f, g, n)` is equivalent to `gray_permute(f, g, n)`. The inverse routine is

```
template <typename Type>
void apply_inverse_permutation(ulong (*x)(ulong), const Type *f, Type * restrict g, ulong n)
// Set g[x(k)] <-- f[k]  \forall k
// Must have:  0<=x(k)<n \forall k
{
    for (ulong k=0; k<n; ++k)  g[x(k)] = f[k];
}
```

The in-place versions of these routines are identical to the routines that apply permutations given as arrays. Only a tiny change must be made in the processing of the cycles. For example, the fragment

```
void apply_permutation(const ulong *x, Type * restrict f, ulong n, bitarray *bp=0)
  [-snip-]
        ulong i = k;  // start of cycle
        Type t = f[i];
        ulong g = x[i];  // <--=
        while ( 0==(tp->test_set(g)) )
        {
            f[i] = f[g];
            i = g;
            g = x[i];  // <--=
        }
        f[i] = t;
  [-snip-]
```

must be changed to

```
void apply_permutation(ulong (*x)(ulong), Type *f, ulong n, bitarray *bp=0)
  [-snip-]
        ulong i = k;  // start of cycle
        Type t = f[i];
        ulong g = x(i);  // <--=
        while ( 0==(tp->test_set(g)) )
        {
            f[i] = f[g];
            i = g;
            g = x(i);  // <--=
        }
        f[i] = t;
  [-snip-]
```

2.13.7 Random permutations

Routines for random permutations are given in [FXT: file perm/permrand.h]. The following routine randomly permutes an array with arbitrary elements:

```
template <typename Type>
void random_permute(Type *f, ulong n)
{
    for (ulong k=1; k<n; ++k)
    {
        ulong r = (ulong)rand();
        r ^= r>>16;  // avoid using low bits of rand alone
        ulong i = r % (k+1);
        swap2(f[k], f[i]);
    }
}
```

A random permutation can be obtained by applying the function of the canonical sequence:

```
void random_permutation(ulong *f, ulong n)
// Create a random permutation
{
    for (ulong k=0; k<n; ++k)  f[k] = k;
    random_permute(f, n);
}
```

A permutation consisting of exactly one cycle is called *cyclic*. Whether a given permutation has this property can be tested with [FXT: `is_cyclic()` in perm/permq.cc]:

```
bool
is_cyclic(const ulong *f, ulong n)
// Return whether permutation is exactly one cycle.
{
    if ( n<=1 )  return true;
    ulong k = 0,  e = 0;
    do  { e=f[e]; ++k; }  while ( e!=0 );
    return  (k==n);
}
```

The method is to follow the cycle starting at position zero and counting how long it is. The permutation is cyclic exactly if the length found equals the array length.

A routine to apply a random cyclic permutation to an array is

```
template <typename Type>
void random_permute_cyclic(Type *f, ulong n)
// Permute the elements of f by a random cyclic permutation.
{
    for (ulong k=n-1; k>0; --k)
    {
        ulong r = (ulong)rand();
        r ^= r>>16;  // avoid using low bits of rand alone
        ulong i = r % k;
        swap2(f[k], f[i]);
    }
}
```

Finally, a random cyclic permutation can be applying a random cyclic permutation to the canonical sequence:

```
inline void
random_cyclic_permutation(ulong *f, ulong n)
// Create a random permutation that is cyclic.
{
    for (ulong k=0; k<n; ++k)  f[k] = k;
    random_permute_cyclic(f, n);
}
```

Chapter 3

Sorting and searching

In this chapter some practical flavors of sorting algorithms are given. These include plain sorting, sorting index arrays, pointer sorting; all optionally with a supplied comparison function. Massive literature exist about the topic so we will not go into the algorithmic details. Very readable text are both [Int2Alg] and [Sedge], in-depth information can be found in [DEK3]. The sorting algorithms used in this chapter are selection sort, quicksort, counting sort and radix sort.

Some algorithms on sorted arrays like binary searching and determination of unique elements are included. Finally, some functions for scanning unsorted arrays are given.

3.1 Sorting

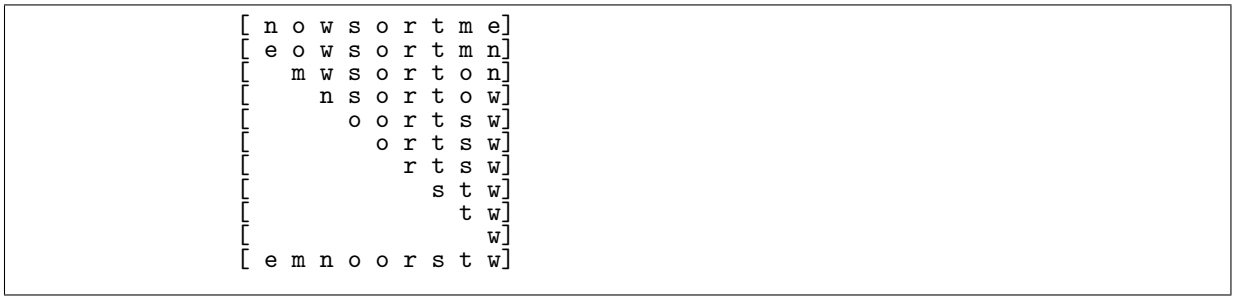


Figure 3.1: Sorting the string '`nowsortme`' with the selection sort algorithm.

There are a several algorithms for sorting that scale with $\sim n^2$ where n is the size of the array to be sorted. Here we use *selection sort* whose idea is to find the minimum of the array, swap it with the first element and repeat for all elements but the first. A demonstration of the algorithm is shown in figure 3.1, this is the output of [FXT: demo sort/selection-sort-demo.cc]. The implementation is straightforward [FXT: file sort/sort.h]:

```
template <typename Type>
void selection_sort(Type *f, ulong n)
// Sort f[] (ascending order).
// Algorithm is proportional to O(n*n), use for short arrays only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[i];
        ulong m = i; // position of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
```

```
            if ( f[j]<v )
            {
                m = j;
                v = f[m];
            }
        }
        swap2(f[i], f[m]);
    }
}
```

A verification routine is always handy:

```
template  <typename Type>
bool is_sorted(const Type *f, ulong n)
// Return whether the sequence
// f[0], f[1], ..., f[n-1]
// is sorted in ascending order.
{
    if ( 0==n )  return 1;

    while ( --n )  // n-1 ... 2
    {
        if ( f[n] < f[n-1] )  break;
    }

    return  !n;
}
```

While the quicksort-algorithm presented below scales $\sim n \log(n)$ (in the average case) it does not just obsolete the more simple schemes because (1) for arrays small enough the 'simple' algorithm is usually the fastest method because of its minimal bookkeeping overhead and (2) therefore it is used inside the quicksort for lengths below some threshold.

The main ingredient of quicksort is to *partition* the array: The corresponding routine reorders some elements where needed and returns some partition index k so that $\max(f_0, \ldots, f_{k-1}) \le \min(f_k, \ldots, f_{n-1})$:

```
template <typename Type>
ulong partition(Type *f, ulong n)
// Rearrange array, so that for some index p
// max(f[0], ..., f[p]) <= min(f[p+1], ..., f[n-1])
{
    // Avoid worst case with already sorted input:
    const Type v = median3(f[0], f[n/2], f[n-1]);

    ulong i = 0UL - 1;
    ulong j = n;
    while ( 1 )
    {
        do  { ++i; }  while ( f[i]<v );
        do  { --j; }  while ( f[j]>v );

        if ( i<j )  swap2(f[i], f[j]);
        else        return j;
    }
}
```

The function `median3()` is defined in [FXT: file sort/minmaxmed23.h]:

```
template <typename Type>
static inline Type  median3(const Type &x, const Type &y, const Type &z)
// Return median of the input values
{ return  x<y ? (y<z ? y : (x<z ? z : x)) : (z<y ? y : (z<x ? z : x)); }
```

The function does 2 or 3 comparisons, depending on the input.

Quicksort calls `partition` on the whole array, then on the two parts left and right from the partition index, then for the four, eight, ... parts, until the parts are of length one. Note that the sub-arrays are usually of different lengths.

```
template <typename Type>
void quick_sort(Type *f, ulong n)
{
    if ( n<=1 )  return;

    ulong p = partition(f, n);
    ulong ln = p + 1;
```

```
    ulong rn = n - ln;
    quick_sort(f, ln);  // f[0]  ... f[ln-1]  left
    quick_sort(f+ln, rn);  // f[ln] ... f[n-1]   right
}
```

The actual implementation uses two optimizations: Firstly, when the size of the subproblems is smaller than a certain threshold selection sort is used. Secondly, the recursive calls are made for the smaller of the two sub-arrays, thereby the stack size is bounded by $\lceil \log_2(n) \rceil$.

```
template <typename Type>
void quick_sort(Type *f, ulong n)
{
 start:
    if ( n<8 ) // parameter: threshold for nonrecursive algorithm
    {
        selection_sort(f, n);
        return;
    }

    ulong p = partition(f, n);
    ulong ln = p + 1;
    ulong rn = n - ln;

    if ( ln>rn )  // recursion for shorter sub-array
    {
        quick_sort(f+ln, rn);  // f[ln] ... f[n-1]   right
        n = ln;
    }
    else
    {
        quick_sort(f, ln);  // f[0]  ... f[ln-1]  left
        n = rn;
        f += ln;
    }

    goto start;
}
```

3.2 Binary search

The main reason for sorting may be that a fast search has to be performed repeatedly. The *binary search* algorithm works by the obvious subdivision of the data [FXT: `bsearch()` in sort/bsearch.h]:

```
template <typename Type>
ulong bsearch(const Type *f, ulong n, const Type v)
// Return index of first element in f[] that equals v
// Return ~0 if there is no such element.
// f[] must be sorted in ascending order.
// Must have  n!=0
{
    ulong nlo=0, nhi=n-1;
    while ( nlo != nhi )
    {
        ulong t = (nhi+nlo)/2;

        if ( f[t] < v )  nlo = t + 1;
        else             nhi = t;
    }

    if ( f[nhi]==v )  return nhi;
    else              return ~0UL;
}
```

The algorithm uses $\sim \log_2(n)$ operations. For very large arrays the algorithm can be improved by selecting the new index `t` different from midpoint `(nhi+nlo)/2`, dependent of the value sought and the distribution of the values in the array. As a simple example consider an array of floating point numbers that are equally distributed in the interval $[\min(v), \max(v)]$. If the sought value equals v one would want to use the relation

$$\frac{n - \min(n)}{\max(n) - \min(n)} = \frac{v - \min(v)}{\max(v) - \min(v)} \tag{3.1}$$

where n denotes an index, and $\min(n), \max(n)$ denote the minimal and maximal index of the current interval. Solving for n gives

$$n \quad = \quad \min(n) + \frac{\max(n) - \min(n)}{\max(v) - \min(v)} \; (v - \min(v)) \tag{3.2}$$

The corresponding *interpolation binary search* algorithm would select the new subdivision index `t` according to the given relation. One could even use quadratic interpolation schemes for the selection of `t`. For the majority of practical applications the midpoint version of the binary search will be good enough.

A simple modification of `bsearch` makes it search the first element greater than or equal to `v`: Replace the operator `==` in the above code by `>=` and you have it: [FXT: `bsearch_ge()` in sort/bsearch.h]. Similar for the '`<=`' relation: `bsearch_le()`.

Approximate matches are found by [FXT: `bsearch_approx()` in sort/bsearchapprox.h]:

```
template <typename Type>
ulong bsearch_approx(const Type *f, ulong n, const Type v, Type da)
// Return index of first element x in f[] for which  |(x-v)| <= da
// Return ~0 if there is no such element.
// f[] must be sorted in ascending order.
// da must be positive.
//
// Makes sense only with inexact types (float or double).
// Must have  n!=0
{
    ulong k = bsearch_ge(f, n, v-da);
    if ( k<n )  k = bsearch_le(f+k, n-k, v+da);
    return k;
}
```

3.3 Index sorting

While the 'plain' sorting reorders an array f so that, after it has finished, $f_k \leq f_{k+1}$ the following routines sort an array of indices without modifying the actual data. The index-sort routines reorder the indices in an array `x` such that `x` applied to `f` as a permutation (in the sense of section 2.13.6 on page 105) will render `f` a sorted array [FXT: file sort/sortidx.h]:

```
template <typename Type>
void idx_selection_sort(const Type *f, ulong n, ulong *x)
// Sort x[] so that the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order.
// Algorithm is proportional to O(n*n), use for short array only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[x[i]];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
            if ( f[x[j]]<v )
            {
                m = j;
                v = f[x[m]];
            }
        }
        swap2(x[i], x[m]);
    }
}
```

Apart from the 'read only'-feature the index-sort routines have the nice property to perfectly work on non-contiguous data.

The verification code looks like:

```
template  <typename Type>
bool is_idx_sorted(const Type *f, ulong n, const ulong *x)
// Return whether the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order.
{
    if ( 0==n )  return 1;

    while ( --n )  // n-1 ... 1
    {
        if ( f[x[n]] < f[x[n-1]] )  break;
    }

    return  !n;
}
```

The transformation of the `partition()` routine is straightforward:

```
template <typename Type>
ulong idx_partition(const Type *f, ulong n, ulong *x)
// rearrange index array, so that for some index p
// max(f[x[0]] ... f[x[p]]) <= min(f[x[p+1]] ... f[x[n-1]])
{
    // Avoid worst case with already sorted input:
    const Type v = median3(*x[0], *x[n/2], *x[n-1], cmp);

    ulong i = 0UL - 1;
    ulong j = n;
    while ( 1 )
    {
        do  ++i;
        while ( f[x[i]]<v );

        do  --j;
        while ( f[x[j]]>v );

        if ( i<j )  swap2(x[i], x[j]);
        else        return j;
    }
}
```

The index-quicksort itself deserves a minute of contemplation comparing it to the plain version:

```
template <typename Type>
void idx_quick_sort(const Type *f, ulong n, ulong *x)
// Sort x[] so that the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order.
{
 start:
    if ( n<8 ) // parameter: threshold for nonrecursive algorithm
    {
        idx_selection_sort(f, n, x);
        return;
    }

    ulong p = idx_partition(f, n, x);
    ulong ln = p + 1;
    ulong rn = n - ln;

    if ( ln>rn )  // recursion for shorter sub-array
    {
        idx_quick_sort(f, rn, x+ln);  // f[x[ln]] ... f[x[n-1]]  right
        n = ln;
    }
    else
    {
        idx_quick_sort(f, ln, x);  // f[x[0]]  ... f[x[ln-1]]  left

        n = rn;
        x += ln;
    }

    goto start;
}
```

The index-analogues of the binary search algorithms are again straightforward, they are given in [FXT: file sort/bsearchidx.h].

3.4 Pointer sorting

Pointer sorting is an idea similar to index sorting which is even less restricted than index sort: The data may be unaligned in memory. And overlapping. Or no data at all but port addresses controlling some highly dangerous machinery. Thereby pointer sort is the perfect way to highly cryptic and powerful programs that seg-fault when you least expect it.

Just to make the idea clear, the array of indices is replaced by an array of pointers:

```
template <typename Type>
void ptr_selection_sort(/*const Type *f,*/ ulong n, const Type **x)
// Sort x[] so that the sequence
// *x[0], *x[1], ..., *x[n-1]
// is sorted in ascending order.
// Algorithm is proportional to O(n*n), use for short array only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = *x[i];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
            if ( *x[j]<v )
            {
                m = j;
                v = *x[m];
            }
        }
        swap2(x[i], x[m]);
    }
}
```

The first argument (`const Type *f`) is not necessary with pointer sorting. It is indicated in commented out form to make the argument structure clear.

The verification routine is

```
template  <typename Type>
bool is_ptr_sorted(/*const Type *f,*/ ulong n, Type const*const*x)
// Return whether the sequence
// *x[0], *x[1], ..., *x[n-1]
// is sorted in ascending order.
{
    if ( 0==n )  return 1;

    while ( --n )  // n-1 ... 1
    {
        if ( *x[n] < *x[n-1] )  break;
    }

    return  !n;
}
```

Find the pointer sorting code in [FXT: file sort/sortptr.h]. The pointer versions of the search routines are given in [FXT: file sort/bsearchptr.h].

3.5 Sorting by a supplied comparison function

The routines in [FXT: file sort/sortfunc.h] are similar to the C-quicksort `qsort` that is part of the standard library. A comparison function `cmp` has to be supplied by the called so that compound data types can be sorted with respect to some key contained. Citing the manual page for `qsort`:

```
The comparison function must return an integer less than, equal to, or greater than
zero if the first argument is considered to be respectively less than, equal to, or
greater than the second.  If two members compare  as  equal,  their  order  in  the
sorted array is undefined.
```

Note that the numerous calls to `cmp` do have a negative impact on the performance. And then with C++ you can provide a comparison 'function' for compound data by overloading the operators `<`, `<`, `<=` and `>=` and use the plain version. That is, the comparisons are inlined an we are back in performance land. Isn't C++ nice?

As a prototypical example we give the selection sort routine:

```
template <typename Type>
void selection_sort(Type *f, ulong n, int (*cmp)(const Type &, const Type &))
// Sort f[] (ascending order)
// with respect to comparison function cmp().
// Algorithm is proportional to O(n*n), use for short array only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[i];
        ulong m = i; // position of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
            if ( cmp(f[j],v) < 0 )
            {
                m = j;
                v = f[m];
            }
        }
        swap2(f[i], f[m]);
    }
}
```

The rest of the supplied routines are rather straightforward translations of the (plain-) sort analogues: Replace the comparison operations as follows

```
    (a < b)      cmp(a,b) < 0
    (a > b)      cmp(a,b) > 0
    (a == b)     cmp(a,b) == 0
    (a <= b)     cmp(a,b) <= 0
    (a >= b)     cmp(a,b) >= 0
```

For example, the verification routine is

```
template  <typename Type>
bool is_sorted(const Type *f, ulong n, int (*cmp)(const Type &, const Type &))
// Return whether the sequence
// f[0], f[1], ..., f[n-1]
// is sorted in ascending order
// with respect to comparison function cmp().
{
    if ( 0==n )  return 1;
    while ( --n )  // n-1 ... 1
    {
        if ( cmp(f[n], f[n-1]) < 0 )  break;
    }
    return  !n;
}
```

3.5.1 Sorting complex numbers

You want to sort complex numbers? Fine for me, but *don't* tell your local mathematician. To see the mathematical problem we ask whether i is smaller or greater than zero. Assume $i > 0$: follows $i \cdot i > 0$ (we multiplied with a positive value) which is $-1 > 0$ and that is false. So, is $i < 0$? Then $i \cdot i > 0$ (multiplication with a negative value, as assumed). So $-1 > 0$, oops! The lesson is that there is no way to impose an arrangement on the complex numbers that would justify the usage of the symbols '$<$' and '$>$' in the mathematical sense.

Nevertheless we can invent a relation that allows us to sort: arranging (sorting) the complex numbers according to their absolute value (modulus) leaves infinitely many numbers in one 'bucket', namely all

those that have the same distance to zero. However, one could use the modulus as the *major* ordering parameter, the angle as the *minor*. Or the real part as the major and the imaginary part as the minor.

The latter is realized in

```
static inline int
cmp_complex(const Complex &f, const Complex &g)
{
    int  ret = 0;
    double fr = f.real();
    double gr = g.real();
    if ( fr==gr )
    {
        double fi = f.imag();
        double gi = g.imag();
        if ( fi!=gi )  ret = (fi>gi ? +1 : -1);
    }
    else    ret = (fr>gr ? +1 : -1);
    return  ret;
}
```

which, when used as comparison with the above function-sort as in

```
void complex_sort(Complex *f, ulong n)
// major order wrt. real part
// minor order wrt. imag part
{
    quick_sort(f, n, cmp_complex);
}
```

can indeed be the practical tool you had in mind.

3.5.2 Index and pointer sorting

The index sorting routines that use a supplied comparison function are given in [FXT: file sort/sortidxfunc.h]:

```
template <typename Type>
void idx_selection_sort(const Type *f, ulong n, ulong *x,
                        int (*cmp)(const Type &, const Type &))
// Sort x[] so that the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order
// with respect to comparison function cmp()
// Algorithm is proportional to O(n*n), use for short array only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[x[i]];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
            if ( cmp(f[x[j]], v) < 0 )
            {
                m = j;
                v = f[x[m]];
            }
        }
        swap2(x[i], x[m]);
    }
}
```

The verification routine is:

```
template  <typename Type>
bool is_idx_sorted(const Type *f, ulong n, const ulong *x,
                   int (*cmp)(const Type &, const Type &))
// Return whether the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order
```

```
// with respect to comparison function cmp()
{
    if ( 0==n )  return 1;
    while ( --n )  // n-1 ... 1
    {
        if ( cmp(f[x[n]], f[x[n-1]]) < 0 )  break;
    }
    return  !n;
}
```

The pointer sorting versions are given in [FXT: file sort/sortptrfunc.h]

```
template <typename Type>
void ptr_selection_sort(/*const Type *f,*/ ulong n, const Type **x,
                        int (*cmp)(const Type &, const Type &))
// Sort x[] so that the sequence
// *x[0], *x[1], ..., *x[n-1]
// is sorted in ascending order
// with respect to comparison function cmp().
// Algorithm is proportional to O(n*n), use for short array only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = *x[i];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
            if ( cmp(*x[j],v)<0 )
            {
                m = j;
                v = *x[m];
            }
        }
        swap2(x[i], x[m]);
    }
}
```

The verification routine is:

```
template  <typename Type>
bool is_ptr_sorted(/*const Type *f,*/ ulong n, Type const*const*x,
                   int (*cmp)(const Type &, const Type &))
// Return whether the sequence
// *x[0], *x[1], ..., *x[n-1]
// is sorted in ascending order
// with respect to comparison function cmp().
{
    if ( 0==n )  return 1;
    while ( --n )  // n-1 ... 1
    {
        if ( cmp(*x[n],*x[n-1])<0 )  break;
    }
    return  !n;
}
```

The corresponding versions of the binary search algorithm are given in [FXT: file sort/bsearchidxfunc.h] and [FXT: file sort/bsearchptrfunc.h].

3.6 Determination of unique elements

In this section we present utility functions that revolve around whether values in a sorted array are repeated or unique. All routines are taken from [FXT: file sort/unique.h].

Testing whether all values are unique:

```
template  <typename Type>
ulong test_unique(const Type *f, ulong n)
```

```
// For a sorted array test whether all values are unique
//   (i.e. whether no value is repeated)
//
// Return 0 if all values are unique
// else return index of the second element in the first pair found
{
    for (ulong k=1; k<n; ++k)
    {
        if ( f[k] == f[k-1] )  return  k;  // k != 0
    }
    return  0;
}

template  <typename Type>
ulong is_unique(const Type *f, ulong n)
// Return true if all values are unique, else return false.
{
    return ( 0==test_unique(f, n) );
}
```

Counting the elements that appear just once:

```
template  <typename Type>
int unique_count(const Type *f, ulong n)
// for a sorted array return the number of unique values
// the number of (not necessarily distinct) repeated
//   values is n - unique_count(f, n);
{
    if ( 1>=n )  return n;
    ulong ct = 1;
    for (ulong k=1; k<n; ++k)
    {
        if ( f[k] != f[k-1] )  ++ct;
    }
    return  ct;
}
```

Removing repeated elements:

```
template  <typename Type>
ulong unique(Type *f, ulong n)
// For a sorted array squeeze all repeated values
// and return the number of unique values.
// Example:  [1, 3, 3, 4, 5, 8, 8] --> [1, 3, 4, 5, 8]
// The routine also works for unsorted arrays as long
// as identical elements only appear in contiguous blocks.
// Example: [4, 4, 3, 7, 7] --> [4, 3, 7]
// The order is preserved.
{
    ulong u = unique_count(f, n);
    if ( u==n )  return n;  // nothing to do
    Type v = f[0];
    for (ulong j=1, k=1;  j<u;  ++j)
    {
        while ( f[k]==v )  ++k;  // search next different element
        v = f[j] = f[k];
    }
    return u;
}
```

The inner `while`-loop does never access an element out of bounds as it is executed only as long as there is at least one remaining change of value inside the array.

3.7 Unique elements with inexact types

Determination of unique elements with inexact types (floats) is a bit tricky as one cannot rely that elements that *should* be identical are exactly equal. A solution to the problem is to allow for a maximal

(absolute) difference within which two contiguous elements will still be considered equal can be provided as additional parameter. We replace equality conditions with call to

```
template  <typename Type>
inline bool approx_equal(Type x1, Type x2, Type da)
// Return whether abs(x2-x1) <= da
// Must have da>=0
{
    Type d = x2 - x1;
    if ( d<=0 )  d = -d;
    if ( d <= da )  return true;
    else            return false;
}
```

The verification routine is [FXT: file sort/uniqueapprox.h]:

```
template  <typename Type>
ulong test_unique_approx(const Type *f, ulong n, Type da)
// For a sorted array test whether all values are
// unique within some tolerance
//   (i.e. whether no value is repeated)
//
// Return 0 if all values are unique
// else return index of the second element in the first pair found
//
// Makes mostly sense with inexact types (float or double)
{
    if ( da<=0 )  da = -da;  // want positive tolerance

    for (ulong k=1; k<n; ++k)
    {
        if ( approx_equal(f[k], f[k-1], da) )  return k;  // k != 0
    }

    return  0;
}
```

One subtle point is that the values can slowly 'drift away' unnoticed by this implementation: Consider a long array where each difference computed has the same sign and is just smaller than `da`, say it is $d = 0.6 \cdot$`da`. The difference of the first and last value then is $0.6 \cdot (n-1) \cdot d$ which is greater than `da` for $n \geq 3$.

Counting unique elements:

```
template  <typename Type>
ulong unique_approx_count(const Type *f, ulong n, Type da)
// for a sorted array return the number of unique values
// the number of (not necessarily distinct) repeated
//   values is n - unique_approx_count(f, n, da);
{
    if ( 1>=n )  return n;
    if ( da<=0 )  da = -da;  // Must have positive tolerance
    ulong ct = 1;
    for (ulong k=1; k<n; ++k)
    {
        if ( approx_equal(f[k], f[k-1], da) )  ++ct;
    }
    return  ct;
}
```

The routine that squeezes out duplicates needs one additional line in the inner loop in order to be consistent with the other functions:

```
template  <typename Type>
ulong unique_approx(Type *f, ulong n, Type da)
// For a sorted array squeeze all repeated (within tolerance da) values
// and return the number of unique values.
// Example:  [1, 3, 3, 4, 5, 8, 8] --> [1, 3, 4, 5, 8]
// The routine also works for unsorted arrays as long
// as identical elements only appear in contiguous blocks.
// Example: [4, 4, 3, 7, 7] --> [4, 3, 7]
// The order is preserved.
{
    ulong u = unique_approx_count(f, n, da);

    if ( u==n )  return n;  // nothing to do
```

```
    if ( da<=0 )  da = -da;  // Must have positive tolerance
    Type v = f[0];
    for (ulong j=1, k=1;  j<u;  ++j)
    {
        // search next different element:
        while ( approx_equal(f[k], v, da) )
        {
            v = f[k];  // avoid problem with slowly drifting values
            ++k;
        }
        v = f[j] = f[k];
    }
    return u;
}
```

Quantization

An alternative way to deal with inexact types is to apply [FXT: `quantize()` in sort/quantize.h]

```
template  <typename Type>
void quantize(Type *f, ulong n, double q)
// In f[] set each element x to q*floor(1/q*(x+q/2))
// E.g.: q=1 ==> round to nearest integer
//       q=1/1000 ==> round to nearest multiple of 1/1000
// For inexact types (float or double).
{
    Type qh = q * 0.5;
    Type q1 = 1.0 / q;
    while ( n-- )
    {
        f[n] = q * floor( q1 * (f[n]+qh) );
    }
}
```

before using `test_unique_approx()`. One should use a quantization parameter `q` that is greater than the value used for `da`.

A minimalistic demo is given in [FXT: demo sort/unique-demo.cc]:

```
    Random values:
       0:  0.9727750243
       1:  0.2925167845
       2:  0.7713576982
       3:  0.5267449795
       4:  0.7699138366
       5:  0.4002286223

    Quantization with q=0.01

    Quantized & sorted :
       0:  0.2900000000
       1:  0.4000000000
       2:  0.5300000000
       3:  0.7700000000
       4:  0.7700000000
       5:  0.9700000000
    First REPEATED value at index 4  (and 3)

    Unique'd array:
       0:  0.2900000000
       1:  0.4000000000
       2:  0.5300000000
       3:  0.7700000000
       4:  0.9700000000
```

The routine `quantize()` turns out to be also useful for the conversion of imprecise data to symbols. For example, the array of floating point values on the left corresponds to the symbolic (numbers used as symbols) table on the right:

```
      1.3133     -1.0101     0.79412     -0.71544          9   2   6   3
     0.29064      0.99173   -1.4382       0.79412          5   7   0   6
     -1.1086      1.2521     0.99173     -1.0101           1   8   7   2
     -0.18003    -1.1086     0.29064      1.3133           4   1   5   9
```

In this example values were considered identical when their absolute difference is less than 10^{-3}. The symbolic representation can be helpful to recognize structure in imprecise data. The routine is [FXT: file sort/symbolify.h]:

```
template <typename Type>
ulong symbolify_by_size(const Type *f, Type * restrict g, ulong n,
                        Type eps=1e-6, ulong *ix=0)
// From f[] compute an array of 'symbols' g[] (i.e. numbers)
//   that represent the different values.
// Values are considered identical if their absolute difference
//   is less than eps.
// Symbols are given with respect to sort-order.
// Return number of different values found (after quantize).
// Optionally supply x[] (scratch space for permutations).
{
    ulong *x = ix;
    if ( 0==ix )  x = new ulong[n];

    set_seq(x, n);
    idx_quick_sort(f, n, x);

    apply_permutation(x, f, g, n);
    quantize(g, n, eps);

    eps *= 0.5;  // some val <1.0
    ulong nsym = 1;

    ulong z = 0;
    Type s = 0.0;
    Type el = g[z], lel;
    g[z] = s;
    for(ulong k=z+1; k<n; ++k)
    {
        lel = el;
        el = g[k];
        if ( fabs(el-lel) > eps )
        {
            ++nsym;
            s += 1.0;
        }
        g[k] = s;
    }

    apply_inverse_permutation(x, g, n);

    if ( 0==ix )  delete [] x;

    return  nsym;
}
```

The example shown was created with the program [FXT: demo sort/symbolify-demo.cc]. The routines `apply_permutation()` and `apply_inverse_permutation()` are given in section 2.13.6 on page 105.

3.8 Determination of equivalence classes

Let S be a set and $C := S \times S$ the set of all ordered pairs (x, y) with $x, y \in S$. A *binary relation* R on S is a subset of C. An *equivalence relation* is a binary relation that has three additional properties:

- *reflexive*: $x \equiv x \; \forall x$.
- *symmetric*: $x \equiv y \iff y \equiv x$.
- *transitive*: $x \equiv y,\ y \equiv z \implies x \equiv z$.

Here we wrote $x \equiv y$ for $(x, y) \in R$ where $x, y \in S$.

We want to determine the *equivalence classes*: An equivalence relation partitions a set into $1 \le q \le n$ subsets E_1, E_2, ..., E_q so that $x \equiv y$ whenever both x and y are in the same subset but $x \not\equiv y$ if x and y are in different subsets.

For example, the usual equality relation is an equivalence relation, with a set of (different) numbers each number is in its own class. With the equivalence relation that $x \equiv y$ whenever $x - y$ is a multiple of some fixed integer m and the set $\mathbb{Z}$ of all natural numbers we obtain m subsets and $x \equiv y$ if and only if $x \equiv y \bmod m$.

Let n be the number of elements in S and Q be a set so that, on termination of the algorithm, $Q_k = j$ if j is the least index so that $S_j \equiv S_k$ (note that we consider the sets to be in a fixed but arbitrary order here).

We proceed as follows:

1. Put each element in its own equivalence class: $Q_k := k$ for all $0 \le k < n$
2. Set $k := 1$ (index of the second element).
3. Search downwards for an equivalent element: For $j = k-1, \ldots, 0$ test whether $S_k \equiv S_j$. If so (at position j), set $Q_k = Q_j$ and goto step 4.
4. Set $k := k+1$ and if $k < n$ goto step 3, else terminate.

We can terminate the search with the first equivalent element found because, if j is the index of the equivalent, the Q_j is already minimal.

The lower and upper bounds for the computational cost are n and n^2, respectively: The algorithm needs proportional n operations when all elements are in the same equivalence class and n^2 operations when each element lies in its own class.

A C++ implementation is [FXT: `equivalence_classes()` in sort/equivclasses.h]. The equivalence relation must be supplied as a function `equiv_q()` that returns true when its arguments are equivalent.

```
template <typename Type>
ulong *
equivalence_classes(const Type *s, ulong n, bool (*equiv_q)(Type,Type), ulong *q=0)
// Given an equivalence relation '==' (as function equiv_q())
//  and a set s[] with n elements,
// write into q[k] (0<=k<n) the index j of the
//  first element s[j] so that s[k]==s[j].
{
    if ( 0==q )  q = new ulong[n];
    for (ulong k=0; k<n; ++k)  q[k] = k;  // each in own class
    for (ulong k=1; k<n; ++k)
    {
        ulong j = k;
        while ( j-- )
        {
            if ( equiv_q(s[j], s[k]) )
            {
                q[k] = q[j];
                break;
            }
        }
    }
    return q;
}
```

3.8.1 Examples

We now give concrete examples for equivalence classes.

Binary necklaces

The set S of n-bit binary words and the equivalence relation so that two words x and y are equivalent when there is a cyclic shift $h_k(x)$ by $0 \le k < b$ positions so that $h_k(x) = y$. The relation is supplied as the function [FXT: demo sort/equivclass-necklaces-demo.cc]:

```
static ulong b;  // number of bits
bool n_equiv_q(ulong x, ulong y)  // necklaces
{
    ulong d = bit_cyclic_dist(x, y, b);
    return  (0==d);
}
```

With $n = 4$ we obtain the following list of equivalence classes:

```
     0:   ....  [#=1]
     1:   1...  .1..  ...1  ..1.  [#=4]
     3:   1..1  11..  ..11  .11.  [#=4]
     5:   .1.1  1.1.  [#=2]
     7:   11.1  111.  1.11  .111  [#=4]
    15:   1111  [#=1]
  # of equivalence classes = 6
```

These correspond to the *binary necklaces* of length 4.

Unlabeled binary necklaces

Same set but the equivalence relation is defined to identify two words x and y when there is a cyclic shift $h_k(x)$ by $0 \le k < b$ positions so that either $h_k(x) = y$ or $h_k(x) = \overline{y}$ where $\overline{y}$ is the complement of y:

```
static ulong mm;  // mask to complement
bool nu_equiv_q(ulong x, ulong y)  // unlabeled necklaces
{
    ulong d = bit_cyclic_dist(x, y, b);
    if ( 0!=d )  d = bit_cyclic_dist(mm^x, y, b);
    return  (0==d);
}
```

With $n = 4$ we obtain:

```
    0:   1111  ....  [#=2]
    1:   111.  11.1  1.11  1...  .111  ...1  ..1.  .1..  [#=8]
    3:   .11.  1..1  11..  ..11  [#=4]
    5:   .1.1  1.1.  [#=2]
  # of equivalence classes = 4
```

These correspond to the *unlabeled binary necklaces* of length 4.

Binary bracelets

The *binary bracelets* are obtained by identifying two words that are identical up to rotation and possible reversion. The corresponding comparison function is

```
bool b_equiv_q(ulong x, ulong y)  // bracelets
{
    ulong d = bit_cyclic_dist(x, y, b);
    if ( 0!=d )  d = bit_cyclic_dist(revbin(x,b), y, b);
    return  (0==d);
}
```

There are six binary bracelets of length 4:

```
  0:   ....  [#=1]
  1:   1...  .1..  ...1  ..1.  [#=4]
  3:   1..1  11..  ..11  .11.  [#=4]
  5:   .1.1  1.1.  [#=2]
  7:   11.1  111.  1.11  .111  [#=4]
 15:   1111  [#=1]
```

The *unlabeled binary bracelets* are obtained by additionally allowing for bitwise complementation:

```
bool bu_equiv_q(ulong x, ulong y)  // unlabeled bracelets
{
    ulong d = bit_cyclic_dist(x, y, b);
    x ^= mm;
    if ( 0!=d )  d = bit_cyclic_dist(x, y, b);

    x = revbin(x,b);
    if ( 0!=d )  d = bit_cyclic_dist(x, y, b);
    x ^= mm;
    if ( 0!=d )  d = bit_cyclic_dist(x, y, b);

    return  (0==d);
}
```

There are four unlabeled binary bracelets of length 4:

```
0:   1111  ....  [#=2]
1:   111.  11.1  1.11  1...  .111  ...1  ..1.  .1..  [#=8]
3:   .11.  1..1  11..  ..11  [#=4]
5:   .1.1  1.1.  [#=2]
```

The shown functions are given in [FXT: demo sort/equivclass-bracelets-demo.cc] which can be used to produce listings of the equivalence classes.

The number of necklaces and bracelets

We give the number of binary necklaces '`N`', bracelets '`B`', unlabeled necklaces '`N/U`' and unlabeled bracelets '`B/U`'. The second row gives the sequence number of [OEIS].

n	N	B	N/U	B/U
[OEIS]#	A000031	A000029	A000013	A000011
1	2	2	1	1
2	3	3	2	2
3	4	4	2	2
4	6	6	4	4
5	8	8	4	4
6	14	13	8	8
7	20	18	10	9
8	36	30	20	18
9	60	46	30	23
10	108	78	56	44
11	188	126	94	63
12	352	224	180	122
13	632	380	316	190
14	1182	687	596	362
15	2192	1224	1096	612

An efficient algorithm for the generation of bracelets is given in [91].

Binary words with reversion and complement

The set S of n-bit binary words and the equivalence relation identifying two words x and y whenever the are mutual complements are bit-wise reversals.

For example, the equivalence classes with 3-, 4- and 5-bit words are:

```
3 classes with 3-bit words:
  0:   111  ...
  1:   .1.  .11  1..  11.
  2:   1.1  .1.

6 classes with 4-bit words:
  0:   1111  1...
  1:   111.  1...  .111  ...1
  2:   .1.1  .1..  1.11  11.1
  3:   11..  ..11
  5:   1.1.  .1.1
  6:   .11.  1..1
```

```
10 classes with 5-bit words:
  0:   11111  .....
  1:   1111.  1....  .1111  ....1
  2:   1.111  111.1  .1...  ...1.
  3:   111..  1..11  ..111  11...
  4:   1.1..  11.11
  5:   11.1.  1.1..  ..1.1  .1.11
  6:   ..11.  .11..  11..1  1..11
  9:   .11.1  1.11.  .1..1  1..1.
 10:   .1.1.  1.1.1
 14:   1...1  .111.
```

The sequence of numbers of equivalence classes for word-sizes n is

```
n:  1, 2, 3, 4,  5,  6,  7,  8,   9,  10,  11,   12,   13,   14,   15,    16, ...
#:  1, 2, 3, 6, 10, 20, 36, 72, 136, 272, 528, 1056, 2080, 4160, 8256, 16512, ...
```

It is entry A005418 of [OEIS].

We have chosen examples where the resulting equivalence classes can be verified by inspection. We could, for example, create the subsets of equivalent necklaces by simply rotating a given word and marking the so far visited words. Such an approach, however, is not possible in general when the equivalence relation does not have an obvious structure.

3.8.2 The number of equivalence relations for a set of n elements

The sequence $B(n)$ of the number of possible partitionings (and thereby equivalence relations) for the set $\{1, 2, \ldots, n\}$ starts as ($n \geq 1$):

```
1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597, ...
```

These are the *Bell numbers*, sequence A000110 of [OEIS]. They can be computed easily as indicated in the following table:

```
0:  [ 1]
1:  [ 1,  2]
2:  [ 2,  3,  5]
3:  [ 5,  7, 10,  15]
4:  [15, 20, 27,  37,  52]
5:  [52, 67, 87, 114, 151, 203]
n:  [B(n), ... ]
```

The first element in each row is the last element of the previous row, the remaining elements are the sum of their left and upper left neighbors. As pari/gp code:

```
N=7;  v=w=b=vector(N);  v[1]=1;
{ for(n=1,N-1,
    b[n] = v[1];
    print(n-1, ":  ", v); \\ print row
    w[1] = v[n];
    for(k=2,n+1, w[k]=w[k-1]+v[k-1]);
    v=w;
 ); }
```

Equivalent C++ code is given in [FXT: demo comb/bell-number-demo.cc]. An alternative way to compute the Bell Numbers is shown in section 5.7 on page 168.

3.9 Determination of monotonicity and convexity *

A sequence is called *monotone* if it is either purely ascending or purely descending. This includes the case where subsequent elements are equal. Whether a constant sequence is considered ascending or descending in this context is a matter of convention.

A routine to check for monotonicity is [FXT: file sort/monotone.h]:

```
template  <typename Type>
int is_monotone(const Type *f, ulong n)
// Return
//   +1 for ascending order
//   -1 for descending order
//   else 0
{
    if ( 1>=n )  return  +1;

    ulong k;
    for (k=1; k<n; ++k)  // skip constant start
    {
        if ( f[k] != f[k-1] )  break;
    }

    if ( k==n )  return +1;  // constant is considered ascending here

    int s = ( f[k] > f[k-1] ? +1 : -1 );

    if ( s>0 ) // was: ascending
    {
        // scan for descending pair:
        for (  ; k<n; ++k)  if ( f[k] < f[k-1] )  return 0;
    }
    else // was: descending
    {
        // scan for ascending pair:
        for (  ; k<n; ++k)  if ( f[k] > f[k-1] )  return 0;
    }

    return  s;
```

```
}
```

A *strictly monotone* sequence is a monotone sequence that has no identical pairs of elements. The test turns out to be slightly easier:

```
template  <typename Type>
int is_strictly_monotone(const Type *f, ulong n)
// return
//   +1 for strictly ascending order
//   -1 for strictly descending order
//   else 0
{
    if ( 1>=n )  return  +1;

    ulong k = 1;
    if ( f[k] == f[k-1] )  return 0;

    int s = ( f[k] > f[k-1] ? +1 : -1 );

    if ( s>0 ) // was: ascending
    {
        // scan for descending pair:
        for (  ; k<n; ++k)  if ( f[k] <= f[k-1] )  return 0;
    }
    else // was: descending
    {
        // scan for ascending pair:
        for (  ; k<n; ++k)  if ( f[k] >= f[k-1] )  return 0;
    }

    return  s;
}
```

A sequence is called *convex* if it starts with an ascending part and ends with a descending part. A *concave* sequence starts with a descending and ends with an ascending part. Whether a monotone sequence is considered convex or concave again is a matter of convention (you have the choice to consider the first or the last element as extremum). Lacking a term that contains both convex and concave the following routine is called `is_convex()` [FXT: file sort/convex.h]:

```
template <typename Type>
long is_convex(Type *f, ulong n)
// Return
//   +val for convex sequence (first rising then falling)
//   -val for concave sequence (first falling then rising)
//   else 0
//
// val is the (second) index of the first pair at the point
// where the ordering changes;  val>=n iff seq. is monotone.
//
// Note: a constant sequence is considered any of rising/falling
{
    if ( 1>=n )  return +1;

    ulong k = 1;
    for (k=1; k<n; ++k)  // skip constant start
    {
        if ( f[k] != f[k-1] )  break;
    }

    if ( k==n )  return  +n;  // constant is considered convex here

    int s = ( f[k] > f[k-1] ? +1 : -1 );

    if ( s>0 ) // was: ascending
    {
        // scan for strictly descending pair:
        for (  ; k<n; ++k)  if ( f[k] < f[k-1] )  break;
        s = +k;
    }
    else // was: descending
    {
        // scan for strictly ascending pair:
        for (  ; k<n; ++k)  if ( f[k] > f[k-1] )  break;
        s = -k;
    }

    if ( k==n )  return s;  // sequence is monotone

    // check that the ordering does not change again:
```

```
    if ( s>0 ) // was: ascending --> descending
    {
        // scan for strictly ascending pair:
        for (  ; k<n; ++k)  if ( f[k] > f[k-1] )  return 0;
    }
    else // was: descending
    {
        // scan for strictly descending pair:
        for (  ; k<n; ++k)  if ( f[k] < f[k-1] )  return 0;
    }
    return  s;
}
```

The test for *strictly convex* (or concave) sequences is:

```
template <typename Type>
long is_strictly_convex(Type *f, ulong n)
// Return
//   +val for strictly convex sequence
//        (i.e. first strictly rising then strictly falling)
//   -val for strictly concave sequence
//        (i.e. first strictly falling then strictly rising)
//   else 0
//
// val is the (second) index of the first pair at the point
// where the ordering changes;  val>=n iff seq. is strictly monotone.
{
    if ( 1>=n )  return +1;
    ulong k = 1;
    if ( f[k] == f[k-1] )  return 0;
    int s = ( f[k] > f[k-1] ? +1 : -1 );
    if ( s>0 ) // was: ascending
    {
        // scan for descending pair:
        for (  ; k<n; ++k)  if ( f[k] <= f[k-1] )  break;
        s = +k;
    }
    else // was: descending
    {
        // scan for ascending pair:
        for (  ; k<n; ++k)  if ( f[k] >= f[k-1] )  break;
        s = -k;
    }
    if ( k==n )  return s;  // sequence is monotone
    else  if ( f[k] == f[k-1] )  return 0;
    // check that the ordering does not change again:
    if ( s>0 ) // was: ascending --> descending
    {
        // scan for ascending pair:
        for (  ; k<n; ++k)  if ( f[k] >= f[k-1] )  return 0;
    }
    else // was: descending
    {
        // scan for descending pair:
        for (  ; k<n; ++k)  if ( f[k] <= f[k-1] )  return 0;
    }
    return  s;
}
```

3.10 Heap-sort

An alternative algorithm for sorting uses the heap data structure introduced in section 4.5 on page 142.

A heap can be sorted by swapping the first (and biggest) element with the last and 'repairing' the array of size $n-1$ by a call to `heapify1()`. Applying this idea recursively until there is nothing more to sort leads to the routine

```
template <typename Type>
```

```
void heap_sort_ascending(Type *x, ulong n)
// sort an array that has the heap-property into ascending order.
// On return x[] is _not_ a heap anymore.
{
    Type *p = x - 1;
    for (ulong k=n; k>1; --k)
    {
        swap2(p[1], p[k]);   // move largest to end of array
        --n;                 // remaining array is one element less
        heapify1(p, n, 1);  // restore heap-property
    }
}
```

that needs time $O(n \log(n))$. That is, a call to

```
template <typename Type>
void heap_sort(Type *x, ulong n)
{
    build_heap(x, n);
    heap_sort_ascending(x, n);
}
```

will sort the array `x[]` into ascending order. Note that sorting into descending order is not any harder [FXT: file sort/heapsort.h]:

```
template <typename Type>
void heap_sort_descending(Type *x, ulong n)
// sort an array that has the heap-property into descending order.
// On return x[] is _not_ a heap anymore.
{
    Type *p = x - 1;
    for (ulong k=n; k>1; --k)
    {
        ++p;  --n;           // remaining array is one element less
        heapify1(p, n, 1);  // restore heap-property
    }
}
```

A demo is given in [FXT: demo sort/heapsort-demo.cc].

3.11 Counting sort and radix sort

Imagine you want to sort an n-element array F of (unsigned) 8-bit values. An sorting algorithm that only uses 2 passes through the data proceeds as follows:

1. Allocate an array C of 256 integers and set all its elements to zero.
2. Count: For $k = 0, 1, \ldots, n-1$ do: Increment $C[F[k]]$.
   Now $C[x]$ contains how many bytes in F have the value x.
3. Set $r = 0$. For $j = 0, 1, \ldots, 255$ do:
   Set $k = C[j]$ and write j to the elements $F[r]$, $F[r+1]$, $\ldots$, $F[r+k-1]$ then add k to r.

For large values of n this method is significantly faster than any other sorting algorithm. Note that no comparisons are made between the elements of F. Instead they are counted, the algorithm is the *counting sort* algorithm.

It might seem that the idea applies only to very special cases but with a little care it can be used in more general situations. We modify the method so that we are able to sort also (unsigned) integer variables whose range of values would make the method impractical with respect to a subrange of the bits in each word. We need an array G that has as many elements as F:

1. Choose any consecutive run of b bits, these will be represented by a bit mask m. Allocate an array C of 2^b integers and set all its elements to zero.

2. Let $M(.)$ be a function that maps the (2^b) values of interest (the bits masked out by m) to the range 0, 1, ..., $2^b - 1$.

3. Count: For $k = 0, 1, \ldots, n-1$ do: Increment $C[M(F[k])]$.
   Now $C[x]$ contains how many values of $M(F[.])$ equal x.

4. Cumulate: For $j = 1, 2, \ldots, 2^b - 1$ (second to last) do: Add $C[j-1]$ to $C[j]$.
   Now $C[x]$ contains the the number of values $M(F[.])$ less or equal to x.

5. Copy: For $k = n-1, \ldots, 2, 1, 0$ (last to first) do: Set $x := M(F[k])$, decrement $C[x]$ then set $i = C[x]$, then set $G[i] := F[x]$.

A crucial property of the algorithm is that it is *stable*: When we use it to sort with respect to a certain m and there is more than one value being mapped to an identical value $M(.) = a$ then the relative order between these elements is preserved:

```
Input:              Counting sort wrt. two lowest bits:
                      m = ....,.11
  0:   11111.11<      0:   ...,i..
  1:   ...,i..        1:   .,iiii..
  2:   ..i.i.i        2:   .iii....
  3:   ..i..,i        3:   .,i.i.i
  4:   .,iiiii<       4:   .i..i..,i
  5:   .,iiii.        5:   .,i...i.
  6:   .i..i..i       6:   .,i.i.ii.
  7:   .i.i.ii.       7:   iiiii.ii<
  8:   .ii...ii<      8:   ..i.iiii<
  9:   .iii....       9:   .ii...ii<
```

Note that order of the three words ending with two set bits (marked with '<') is preserved.

A routine that verifies whether an array is sorted with respect to a bit range specified by the variable `b0` and `m` is [FXT: file sort/radixsort.cc]:

```
bool
is_counting_sorted(const ulong *f, ulong n, ulong b0, ulong m)
// Whether f[] is sorted wrt. bits b0,...,b0+z-1
//  where z is the number of bits set in m.
// m must contain a single run of bits starting at bit zero.
{
    m <<= b0;
    for (ulong k=1; k<n; ++k)
    {
        ulong xm = (f[k-1] & m ) >> b0;
        ulong xp = (f[k] & m ) >> b0;
        if ( xm>xp )  return false;
    }
    return true;
}
```

The function $M(.)$ is the combination of a mask-out and a shift operation. A routine that sorts according to `b0` and `m` is:

```
void
counting_sort_core(const ulong *f, ulong n, ulong * restrict g, ulong b0, ulong m)
// Write to g[] the array f[] sorted wrt. bits b0,...,b0+z-1
//  where z is the number of bits set in m.
// m must contain a single run of bits starting at bit zero.
{
    ulong nb = m + 1;
    m <<= b0;
    ALLOCA(ulong, cv, nb);
    for (ulong k=0; k<nb; ++k)  cv[k] = 0;

    // --- count:
    for (ulong k=0; k<n; ++k)
    {
        ulong x = (f[k] & m ) >> b0;
        ++cv[ x ];
    }

    // --- cumulative sums:
    for (ulong k=1; k<nb; ++k)  cv[k] += cv[k-1];

    // --- reorder:
    ulong k = n;
    while ( k-- )  // backwards ==> stable sort
    {
```

```
        ulong fk = f[k];
        ulong x = (fk & m) >> b0;
        --cv[x];
        ulong i = cv[x];
        g[i] = fk;
    }
}
```

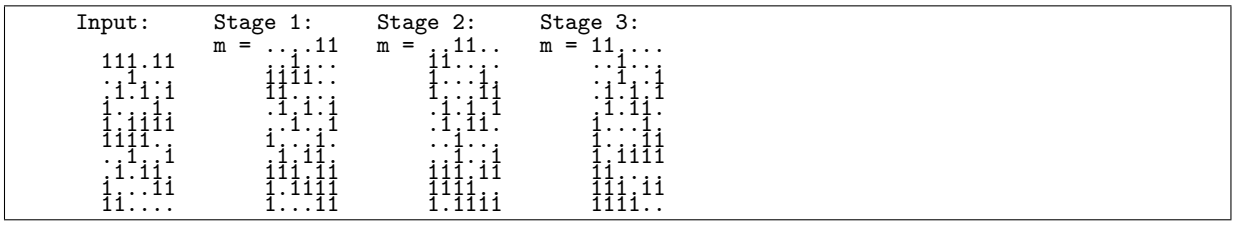


Figure 3.2: Radix sort of 10 six-bit values when using two-bit masks.

Now we can apply counting sort to a set of bit masks that cover the whole range. Figure 3.2 shows an example with 10 six-bit values and 3 two-bit masks, starting from the least significant bits. This is the output of [FXT: demo sort/radixsort-demo.cc] which allows to choose the number of bits sorted with each stage.

The routine [FXT: `radix_sort()` in sort/radixsort.cc] uses 8-bit masks to sort unsigned (long) integers:

```
void
radix_sort(ulong *f, ulong n)
{
    ulong nb = 8;  // Number of bits sorted with each step
    ulong tnb = BITS_PER_LONG; // Total number of bits

    ulong *fi = f;
    ulong *g = new ulong[n];

    ulong m = (1UL<<nb) - 1;
    for (ulong k=1, b0=0;  b0<tnb;  ++k, b0+=nb)
    {
        counting_sort_core(f, n, g, b0, m);
        swap2(f, g);
    }

    if ( f!=fi )  // result is actually in g[]
    {
        swap2(f, g);
        for (ulong k=0; k<n; ++k)  f[k] = g[k];
    }

    delete [] g;
}
```

There is room for optimization. Using two arrays for counting and combining copying with counting for the next pass where possible will reduce the number of passes almost by a factor of two.

3.12 Searching in unsorted arrays

In what follows an overview of the search functions for unordered arrays is given.

3.12.1 Minimal and maximal elements in unsorted arrays

Sorting an array only to determine the minimal (and/or maximal element) is obviously a bad idea because these can be found in linear time in the unsorted data.

Corresponding to all flavors of the sorting routines a `min()` and `max()` routine is supplied. We exemplify by giving the versions that determine the minimum.

Finding the minimum in an unsorted array corresponding to the 'plain' sorting routine: [FXT: file sort/minmax.h]

```
template <typename Type>
Type inline min(const Type *f, ulong n)
// Return minimum of array.
{
    Type v = f[0];
    while ( n-- )  if ( f[n]<v )  v = f[n];
    return v;
}
```

The index version: [FXT: file sort/minmaxidx.h]

```
template <typename Type>
Type idx_min(const Type *f, ulong n, const ulong *x)
// Return minimum (value) of array elements
//   {f[x[0]], f[x[1]], .. , f[x[n-1]]}
{
    Type v = f[x[0]];
    while ( n-- )  { if ( f[x[n]]<v )  v = f[x[n]]; }
    return  v;
}
```

The pointer version: [FXT: file sort/minmaxptr.h]

```
template <typename Type>
Type ptr_min(/*const Type *f,*/ ulong n, Type const*const*x)
// Return minimum (value) of array elements
//   {*x[0], *x[1], .. , *x[n-1]}
{
    Type v = *x[0];
    while ( n-- )  { if ( *x[n]<v )  v = *x[n]; }
    return  v;
}
```

The comparison function version: [FXT: file sort/minmaxfunc.h]

```
template <typename Type>
Type min(const Type *f, ulong n, int (*cmp)(const Type &, const Type &))
// Return minimum (value) of array elements
// wrt. to comparison function
{
    Type v = f[0];
    while ( n-- )  { if ( cmp(f[n],v) < 0 )  v = f[n]; }
    return  v;
}
```

The comparison function with index version: [FXT: file sort/minmaxidxfunc.h]

```
template <typename Type>
Type idx_min(const Type *f, ulong n, const ulong *x,
             int (*cmp)(const Type &, const Type &))
// Return minimum (value) of array elements
//   {f[x[0]], f[x[1]], .. , f[x[n-1]]}
// with respect to comparison function cmp()
{
    Type v = f[x[0]];
    while ( n-- )  { if ( cmp(f[x[n]], v) < 0 )  v = f[x[n]]; }
    return  v;
}
```

The comparison function with pointer version: [FXT: file sort/minmaxptrfunc.h]

```
template <typename Type>
Type ptr_min(/*const Type *f,*/ ulong n, Type const*const*x,
             int (*cmp)(const Type &, const Type &))
// Return minimum (value) of array elements
//   {*x[0], *x[1], .. , *x[n-1]}
// with respect to comparison function cmp().
{
    Type v = *x[0];
    while ( n-- )  { if ( cmp(*x[n],v)<0 )  v = *x[n]; }
    return  v;
}
```

3.12.2 Searching for values

To find the first occurrence of a certain value in an unsorted array one can use the routine [FXT: file sort/usearch.h]

```
template <typename Type>
inline ulong first_eq_idx(const Type *f, ulong n, Type v)
// Return index of first element == v
// Return n if all !=v
{
    ulong k = 0;
    while ( (k<n) && (f[k]!=v) )  k++;
    return k;
}
```

The functions `first_ne_idx()`, `first_ge_idx()` and `first_le_idx()` find the first occurrence of an element unequal (to `v`), greater or equal and less or equal, respectively.

If the last bit of speed matters, one could replace the shown code by

```
template <typename Type>
inline ulong first_eq_idx(const Type *f, ulong n, Type v)
// Return index of first element == v
// Return n if all !=v
{
    Type s = f[n-1];
    (Type *)f[n-1] = v;  // guarantee that the search stops
    ulong k = 0;
    while ( f[k]!=v )  k++;
    (Type *)f[n-1] = s;  // restore value
    if ( (k==n-1) && (v!=s) )  ++k;
    return k;
}
```

Replace the word `first` by last in the names to name the function that detect the last element satisfying the corresponding condition. For example,

```
template <typename Type>
inline ulong last_ge_idx(const Type *f, ulong n, Type v)
// Return index of last element >= v
// Return n if all <v
{
    ulong k = n-1;
    while ( f[k]<v )
    {
        k--;
        if ( 0==k )  return n;
    }
    return k;
}
```

3.12.3 Counting values

The functions in [FXT: file sort/ucount.h] count certain elements in unordered arrays.

```
template <typename Type>
inline ulong eq_count(const Type *f, ulong n, Type v)
// Return number of elements that are ==v
{
    ulong ct = 0;
    while ( n-- )  if ( f[n]==v )  ++ct;
    return ct;
}
```

As above, replace `eq_` in the function name with `ne_`, `ge_` or `le_` to count the number of occurrences of elements that are unequal (to `v`), greater or equal and less or equal, respectively.

3.12.4 Searching matches

The routines in [FXT: file sort/usearchfunc.h] generalize the functions form [FXT: file sort/usearch.h]: A supplied function implements the condition imposed. For example,

```
template <typename Type>
inline ulong first_idx(const Type *f, ulong n, bool (* func)(Type))
// Return index of first element for which func() returns true.
// Return n if there is no such element.
{
    ulong k = 0;
    while ( (k<n) && (!func(f[k])) )  k++;
    return k;
}
```

and `last_idx()` that scans beginning from the greatest index.

The functions

```
template <typename Type>
inline ulong next_idx(const Type *f, ulong n, bool (* func)(Type), ulong k0)
// Like first_idx() but start from k0.
{
    ulong k = k0;
    while ( (k<n) && (!func(f[k])) )  k++;
    return k;
}
```

and `previous_idx()` allow to determine all matching elements in forward or backward direction.

3.12.5 Selecting matches: grep

The routines from [FXT: file sort/grep.h] count or select elements from an unordered array for which a condition implemented by a supplied function is true.

The following function counts the matching elements:

```
template <typename Type>
inline ulong count(const Type *f, ulong n, bool (* func)(Type))
// Return number of elements for which func() returns true.
{
    ulong ct = 0;
    for (ulong k=0; k<n; ++k)  if ( func(f[k]) )  ct++;
    return  ct;
}
```

Discard all non-matches using

```
template <typename Type>
inline ulong grep(Type *f, ulong n, bool (* func)(Type))
// Delete elements for which func() returns false.
// Return number of elements kept.
{
    ulong k, j;
    for (k=0,j=0; j<n; ++k,++j)
    {
        f[k] = f[j];
        if ( func(f[j]) )  --k;
    }
    return k;
}
```

Record the values of the matches using

```
template <typename Type>
inline ulong grep(const Type *f, ulong n, bool (* func)(Type), Type *g)
// Make g[] the sequence of values for which func() returns true.
// Return number of 'matching' elements found.
{
    ulong ct = 0;
    for (ulong k=0; k<n; ++k)  if ( func(f[k]) )  g[ct++] = f[k];
    return  ct;
}
```

Finally, recording the indices of the matches can be done with

```
template <typename Type>
inline ulong grep_idx(const Type *f, ulong n, bool (* func)(Type), ulong *x)
// Make x[] the sequence of indices for which func() returns true.
// Return number of 'matching' elements found.
{
    ulong ct = 0;
    for (ulong k=0; k<n; ++k)  if ( func(f[k]) )  x[ct++] = k;
    return  ct;
}
```

Chapter 4

Data structures

This chapter presents implementations of the operations on selected data structures like LIFO, FIFO, deque and heap.

4.1 Stack (LIFO)

A *stack* (or LIFO for last-in, first-out) is a data structure that supports the operations: *push* to saves an entry and *pop* to retrieve and remove the entry that was entered last. The occasionally useful operation *peek* retrieves the same element as pop but does not remove it. Similarly, *poke* modifies the last entry.

An implementation with the option to let the stack grow when necessary is given in [FXT: file ds/stack.h]

```
template <typename Type>
class stack
{
public:
    Type  *x_;  // data
    ulong  s_;  // size
    ulong  p_;  // stack pointer (position of next write), top entry @ p-1
    ulong  gq_; // grow gq elements if necessary, 0 for "never grow"
public:
    stack(ulong n, ulong growq=0)
    {
        s_ = n;
        x_ = new Type[s_];
        p_ = 0;  // stack is empty
        gq_ = growq;
    }
    ~stack()  { delete [] x_; }
private:
    stack & operator = (const stack &);  // forbidden
public:
    ulong n()  const
    // return number of entries
    {
        return p_;
    }
    ulong push(Type z)
    // return size of stack, zero on stack overflow
    // if gq_ is nonzero the stack grows
    {
        if ( p_ >= s_ )
        {
            if ( 0==gq_ )  return 0;  // overflow
            grow();
        }
```

```
        x_[p_] = z;
        ++p_;
        return  s_;
    }

    ulong pop(Type &z)
    // read top entry and remove it
    // return number of entries at function start
    // if empty return zero and z is undefined
    {
        ulong ret = p_;
        if ( 0!=p_ )  { --p_;  z = x_[p_]; }
        return  ret;
    }

    ulong poke(Type z)
    // modify top entry
    // return number of entries
    // if empty return zero and nothing is done
    {
        if ( 0!=p_ )  x_[p_-1] = z;
        return p_;
    }

    ulong peek(Type &z)
    // read top entry (without removing it)
    // return number of entries
    // if empty return zero and z is undefined
    {
        if ( 0!=p_ )  z = x_[p_-1];
        return p_;
    }
   [-snip-]
```

The growth routine is implemented as

```
   [-snip-]
private:
    void grow()
    {
        ulong ns = s_ + gq_;  // new size
        x_ = ReAlloc<Type>(x_, ns);
        s_ = ns;
    }
};
```

here we used the function that imports the C function `realloc()`.

```
% man realloc
    #include <stdlib.h>

    void *realloc(void *ptr, size_t size);

    realloc() changes the size of the memory block pointed to  by  ptr  to  size
    bytes.   The  contents  will  be unchanged to the minimum of the old and new
    sizes; newly allocated memory will be uninitialized.  If ptr  is  NULL,  the
    call  is  equivalent  to malloc(size); if size is equal to zero, the call is
    equivalent to free(ptr).  Unless ptr is NULL, it must have been returned  by
    an earlier call to malloc(), calloc() or realloc().
```

Usage of the C realloc can be disabled globally in [FXT: file realloc.h]:

```
#define USE_C_REALLOC  // comment out to disable use of realloc()

#ifdef USE_C_REALLOC
#include <cstdlib>  // realloc()
#endif

#ifdef USE_C_REALLOC
template <typename Type>
inline Type *ReAlloc(Type *p, ulong n)
{
    return (Type *)realloc((void *)p, n*sizeof(Type));
}

#else

template <typename Type>
inline Type *ReAlloc(Type *p, ulong n, ulong nold)
```

```
{
    Type *np = new Type[n];
    ulong nc = (nold < n ? nold : n);
    for (ulong k=0; k<nc; ++k)  np[k] = p[k];
    delete [] p;
    return np;
}
#endif
```

A demo showing the working of the stack is demonstrated in [FXT: demo ds/stack-demo.cc]. An example output where the initial size is 4 and the growths-feature enabled (in steps of 4 elements) looks like:

```
    push( 1)  1  -  -  -   #=1
    push( 2)  1  2  -  -   #=2
    push( 3)  1  2  3  -   #=3
    push( 4)  1  2  3  4   #=4
    push( 5)  1  2  3  4  5  -  -  -   #=5
    push( 6)  1  2  3  4  5  6  -  -   #=6
    push( 7)  1  2  3  4  5  6  7  -   #=7
    pop== 7   1  2  3  4  5  6  -  -   #=6
    pop== 6   1  2  3  4  5  -  -  -   #=5
    push( 8)  1  2  3  4  5  8  -  -   #=6
    pop== 8   1  2  3  4  5  -  -  -   #=5
    pop== 5   1  2  3  4  -  -  -  -   #=4
    push( 9)  1  2  3  4  9  -  -  -   #=5
    pop== 9   1  2  3  4  -  -  -  -   #=4
    pop== 4   1  2  3  -  -  -  -  -   #=3
    push(10)  1  2  3 10  -  -  -  -   #=4
    pop==10   1  2  3  -  -  -  -  -   #=3
    pop== 3   1  2  -  -  -  -  -  -   #=2
    push(11)  1  2 11  -  -  -  -  -   #=3
    pop==11   1  2  -  -  -  -  -  -   #=2
    pop== 2   1  -  -  -  -  -  -  -   #=1
    push(12)  1 12  -  -  -  -  -  -   #=2
    pop==12   1  -  -  -  -  -  -  -   #=1
    pop== 1   -  -  -  -  -  -  -  -   #=0
    push(13) 13  -  -  -  -  -  -  -   #=1
    pop==13   -  -  -  -  -  -  -  -   #=0
    pop== 0   -  -  -  -  -  -  -  -   #=0
      (stack was empty)
    push(14) 14  -  -  -  -  -  -  -   #=1
    pop==14   -  -  -  -  -  -  -  -   #=0
    pop== 0   -  -  -  -  -  -  -  -   #=0
      (stack was empty)
    push(15) 15  -  -  -  -  -  -  -   #=1
```

4.2 Ring buffer

A *ring buffer* is a array plus read- and write operations that wrap around. That is, if the last position of the array is reached writing continues at the begin of the array, thereby erasing the oldest entries. The read operation should start at the oldest entry in the array.

The `ringbuffer` utility class in [FXT: `class ringbuffer` in ds/ringbuffer.h] is as simple as:

```
template <typename Type>
class ringbuffer
{
public:
    Type *x_;   // data (ring buffer)
    ulong s_;   // allocated size (# of elements)
    ulong n_;   // current number of entries in buffer
    ulong wpos_;  // next position to write in buffer
    ulong fpos_;  // first position to read in buffer
public:
    explicit ringbuffer(ulong n)
    {
        s_ = n;
        x_ = new Type[s_];
        n_ = 0;
```

```
        wpos_ = 0;
        fpos_ = 0;
    }
    ~ringbuffer()  { delete [] x_; }
    ulong n()  const  { return n_; }
    void insert(const Type &z)
    {
        x_[wpos_] = z;
        if ( ++wpos_>=s_ )  wpos_ = 0;
        if ( n_ < s_ )  ++n_;
        else  fpos_ =  wpos_;
    }
    ulong read(ulong n, Type &z)  const
    // read entry n (that is, [(fpos_ + n)%s_])
    // return 0 if entry #n is last in buffer
    // else return n+1
    {
        if ( n>=n_ )  return 0;
        ulong j = fpos_ + n;
        if ( j>=s_ )  j -= s_;
        z = x_[j];
        return  n + 1;
    }
};
```

Reading from the ring buffer goes like:

```
ulong k = 0;
Type z;  // type of entry
while ( (k = f.read(k, z)) )
{
    // do something with z
}
```

A demo is in [FXT: demo ds/ringbuffer-demo.cc], its output is

```
insert( 1)  1   #=1   r=1   w=0
insert( 2)  1  2   #=2   r=2   w=0
insert( 3)  1  2  3   #=3   r=3   w=0
insert( 4)  1  2  3  4   #=4   r=0   w=0
insert( 5)  2  3  4  5   #=4   r=1   w=1
insert( 6)  3  4  5  6   #=4   r=2   w=2
insert( 7)  4  5  6  7   #=4   r=3   w=3
insert( 8)  5  6  7  8   #=4   r=0   w=0
insert( 9)  6  7  8  9   #=4   r=1   w=1
```

Ring buffers can be useful for storing a constant amount of history-data such as for logging purposes. For that purpose one would enhance the `ringbuffer` class so that it uses an additional array of (fixed width) strings. The message to log would be copied into the array and the pointer set accordingly. Read should then just return the pointer to the string.

4.3 Queue (FIFO)

A *queue* (or FIFO for first-in, first-out) is a data structure that supports two operations: *push* saves an entry and *pop* retrieves (and removes) the entry that was entered the longest time ago. The occasionally useful operation *peek* retrieves the same element as pop but does not remove it.

A utility class with the optional feature of growing if necessary is [FXT: `class queue` in ds/queue.h]:

```
template <typename Type>
class queue
{
public:
    Type *x_;   // pointer to data
    ulong s_;   // allocated size (# of elements)
    ulong n_;   // current number of entries in buffer
    ulong wpos_;  // next position to write in buffer
    ulong rpos_;  // next position to read in buffer
```

```
    ulong gq_;  // grow gq elements if necessary, 0 for "never grow"
public:
    explicit queue(ulong n, ulong growq=0)
    {
        s_ = n;
        x_ = new Type[s_];
        n_ = 0;
        wpos_ = 0;
        rpos_ = 0;
        gq_ = growq;
    }

    ~queue()  { delete [] x_; }

    ulong n()  const  { return n_; }

    ulong push(const Type &z)
    // Return number of entries.
    // Zero is returned on failure
    //   (i.e. space exhausted and 0==gq_)
    {
        if ( n_ >= s_ )
        {
            if ( 0==gq_ )  return 0;  // growing disabled
            grow();
        }

        x_[wpos_] = z;
        ++wpos_;
        if ( wpos_>=s_ )  wpos_ = 0;

        ++n_;
        return n_;
    }

    ulong peek(Type &z)
    // Return number of entries.
    // if zero is returned the value of z is undefined.
    {
        z = x_[rpos_];
        return n_;
    }

    ulong pop(Type &z)
    // Return number of entries before pop
    // i.e. zero is returned if queue was empty.
    // If zero is returned the value of z is undefined.
    {
        ulong ret = n_;
        if ( 0!=n_ )
        {
            z = x_[rpos_];
            ++rpos_;
            if ( rpos_ >= s_ )  rpos_ = 0;
            --n_;
        }

        return ret;
    }
private:
    void grow()
    {
        ulong ns = s_ + gq_;  // new size
#ifdef USE_C_REALLOC
        rotate_left(x_, s_, rpos_);
        x_ = ReAlloc<Type>(x_, ns);
        wpos_ = s_;
        rpos_ = 0;
#else
        Type *nx = new Type[ns];
        // move read-position to zero:
        // rotate_left(x_, s_, rpos_);  copy(x_, nx, s_);
        copy_cyclic(x_, nx, s_, rpos_);
        wpos_ = s_;
        rpos_ = 0;
        delete [] x_;  x_ = nx;
#endif
    }

};
```

Its working is demonstrated in [FXT: demo ds/queue-demo.cc]. An example output where the initial size is 4 and the growths-feature enabled (in steps of 4 elements) looks like:

```
    push( 1)  1  -  -  -   #=1   r=0   w=1
    push( 2)  1  2  -  -   #=2   r=0   w=2
    push( 3)  1  2  3  -   #=3   r=0   w=3
    push( 4)  1  2  3  4   #=4   r=0   w=0
    push( 5)  1  2  3  4  5  -  -  -   #=5   r=0   w=5
    push( 6)  1  2  3  4  5  6  -  -   #=6   r=0   w=6
    push( 7)  1  2  3  4  5  6  7  -   #=7   r=0   w=7
    pop== 1   -  2  3  4  5  6  7  -   #=6   r=1   w=7
    pop== 2   -  -  3  4  5  6  7  -   #=5   r=2   w=7
    push( 8)  -  -  3  4  5  6  7  8   #=6   r=2   w=0
    pop== 3   -  -  -  4  5  6  7  8   #=5   r=3   w=0
    pop== 4   -  -  -  -  5  6  7  8   #=4   r=4   w=0
    push( 9)  9  -  -  -  5  6  7  8   #=5   r=4   w=1
    pop== 5   9  -  -  -  -  6  7  8   #=4   r=5   w=1
    pop== 6   9  -  -  -  -  -  7  8   #=3   r=6   w=1
    push(10)  9 10  -  -  -  -  7  8   #=4   r=6   w=2
    pop== 7   9 10  -  -  -  -  -  8   #=3   r=7   w=2
    pop== 8   9 10  -  -  -  -  -  -   #=2   r=0   w=2
    push(11)  9 10 11  -  -  -  -  -   #=3   r=0   w=3
    pop== 9   - 10 11  -  -  -  -  -   #=2   r=1   w=3
    pop==10   -  - 11  -  -  -  -  -   #=1   r=2   w=3
    push(12)  -  - 11 12  -  -  -  -   #=2   r=2   w=4
    pop==11   -  -  - 12  -  -  -  -   #=1   r=3   w=4
    pop==12   -  -  -  -  -  -  -  -   #=0   r=4   w=4
    push(13)  -  -  -  - 13  -  -  -   #=1   r=4   w=5
    pop==13   -  -  -  -  -  -  -  -   #=0   r=5   w=5
    pop== 0   -  -  -  -  -  -  -  -   #=0   r=5   w=5
      (queue was empty)
    push(14)  -  -  -  -  - 14  -  -   #=1   r=5   w=6
    pop==14   -  -  -  -  -  -  -  -   #=0   r=6   w=6
    pop== 0   -  -  -  -  -  -  -  -   #=0   r=6   w=6
      (queue was empty)
    push(15)  -  -  -  -  -  - 15  -   #=1   r=6   w=7
```

This is [FXT: demo ds/queue-out.txt]. You might want to compare this to the stack-demo at page 137.

4.4 Deque (double-ended queue)

A *deque* (for *double-ended queue*) combines the data structures stack and queue: insertion and deletion is possible both at the first- and the last position, all in time-$O(1)$. An implementation with the option to let the stack grow when necessary is given in [FXT: file ds/deque.h]

```
template <typename Type>
class deque
{
public:
    Type *x_;   // data (ring buffer)
    ulong s_;   // allocated size (# of elements)
    ulong n_;   // current number of entries in buffer
    ulong fpos_;  // position of first element in buffer
    // insert_first() will write to (fpos-1)%n
    ulong lpos_;  // position of last element in buffer plus one
    // insert_last() will write to  lpos,  n==(lpos-fpos) (mod s)
    // entries are at [fpos, ... , lpos-1]  (range may be empty)

    ulong  gq_; // grow gq elements if necessary, 0 for "never grow"
public:
    explicit deque(ulong n, ulong growq=0)
    {
        s_ = n;
        x_ = new Type[s_];
        n_ = 0;
        fpos_ = 0;
        lpos_ = 0;
        gq_ = growq;
    }
```

```
~deque()  { delete [] x_; }

ulong n()  const  { return n_; }

ulong insert_first(const Type &z)
// returns number of entries after insertion
// zero is returned on failure
//   (i.e. space exhausted and 0==gq_)
{
    if ( n_ >= s_ )
    {
        if ( 0==gq_ )  return 0;  // growing disabled
        grow();
    }

    --fpos_;
    if ( fpos_ == -1UL )  fpos_ = s_ - 1;
    x_[fpos_] = z;
    ++n_;
    return  n_;
}

ulong insert_last(const Type &z)
// returns number of entries after insertion
// zero is returned on failure
//   (i.e. space exhausted and 0==gq_)
{
    if ( n_ >= s_ )
    {
        if ( 0==gq_ )  return 0;  // growing disabled
        grow();
    }

    x_[lpos_] = z;
    ++lpos_;
    if ( lpos_>=s_ )  lpos_ = 0;
    ++n_;
    return  n_;
}

ulong extract_first(Type & z)
// return number of elements before extract
// return 0 if extract on empty deque was attempted
{
    if ( 0==n_ )  return 0;
    z = x_[fpos_];
    ++fpos_;
    if ( fpos_ >= s_ )  fpos_ = 0;
    --n_;
    return  n_ + 1;
}

ulong extract_last(Type & z)
// return number of elements before extract
// return 0 if extract on empty deque was attempted
{
    if ( 0==n_ )  return 0;
    --lpos_;
    if ( lpos_ == -1UL )  lpos_ = s_ - 1;
    z = x_[lpos_];
    --n_;
    return  n_ + 1;
}

ulong read_first(Type & z)  const
// read (but don't remove) first entry
// return number of elements (i.e. on error return zero)
{
    if ( 0==n_ )  return 0;
    z = x_[fpos_];
    return n_;
}

ulong read_last(Type & z)  const
// read (but don't remove) last entry
// return number of elements (i.e. on error return zero)
{
```

```
        return  read(n_-1, z);  // ok for n_==0
    }

    ulong read(ulong k, Type & z)  const
    // read entry k (that is, [(fpos_ + k)%s_])
    // return 0 if k>=n_
    // else return k+1
    {
        if ( k>=n_ )  return 0;
        ulong j = fpos_ + k;
        if ( j>=s_ )  j -= s_;
        z = x_[j];
        return  k + 1;
    }
private:
    void grow()
    {
        ulong ns = s_ + gq_;  // new size
        // Move read-position to zero:
        rotate_left(x_, s_, fpos_);
        x_ = ReAlloc<Type>(x_, ns);
        fpos_ = 0;
        lpos_ = n_;
        s_ = ns;
    }
};
```

Its working is demonstrated in [FXT: demo ds/deque-demo.cc]:

```
    insert_first( 1)        1
    insert_last(51)         1 51
    insert_first( 2)        2  1 51
    insert_last(52)         2  1 51 52
    insert_first( 3)        3  2  1 51 52
    insert_last(53)         3  2  1 51 52 53
    extract_first()= 3      2  1 51 52 53
    extract_last()= 53      2  1 51 52
    insert_first( 4)        4  2  1 51 52
    insert_last(54)         4  2  1 51 52 54
    extract_first()= 4      2  1 51 52 54
    extract_last()= 54      2  1 51 52
    extract_first()= 2      1 51 52
    extract_last()= 52      1 51
    extract_first()= 1     51
    extract_last()= 51
    insert_first( 5)        5
    insert_last(55)         5 55
    extract_first()= 5     55
    extract_last()= 55
    extract_first()= (deque is empty)
    extract_last()=  (deque is empty)
    insert_first( 7)        7
    insert_last(57)         7 57
```

4.5 Heap and priority queue

A *heap* is a binary tree where the left and right children are smaller than or equal to their parent node. The function [FXT: file ds/heap.h]

```
template <typename Type>
ulong test_heap(const Type *x, ulong n)
// return 0 if x[] has heap property
// else index of node found to be bigger than its parent
{
    const Type *p = x - 1;
    for (ulong k=n; k>1; --k)
    {
        ulong t = (k>>1); // parent(k)
        if ( p[t]<p[k] )  return k;
    }
    return 0;  // has heap property
}
```

finds out whether a given array has the described heap-property.

It turns out that a unordered array of size n can be reordered to a heap in $O(n)$ time[1]. The routine

```
template <typename Type>
void build_heap(Type *x, ulong n)
// reorder data to a heap
{
    for (ulong j=(n>>1); j>0; --j)  heapify1(x-1, n, j);
}
```

does the trick. It uses the subroutine

```
template <typename Type>
void heapify1(Type *z, ulong n, ulong k)
// subject to the condition that the trees below the children of node
// k are heaps, move the element z[k] (down) until the tree below node
// k is a heap.
//
// data expected in z[1..n] (!)
{
    ulong m = k;
    ulong l = (k<<1); // left(k);
    if ( (l <= n) && (z[l] > z[k]) )  m = l;
    ulong r = (k<<1) + 1; // right(k);
    if ( (r <= n) && (z[r] > z[m]) )  m = r;
    if ( m != k )
    {
        swap2(z[k], z[m]);
        heapify1(z, n, m);
    }
}
```

That runs in time $O(\log(n))$.

Heaps are useful to build a so-called *priority queue*. This is a data structure that supports insertion of an element and extraction of the maximal element it contains both in an efficient manner. A priority queue can be used to 'schedule' a certain event for a given point in time and return the next pending event.

A new element can be inserted into a heap in $O(\log(n))$ time by appending it and moving it towards the root (that is, the first element) as necessary:

```
bool heap_insert(Type *x, ulong n, ulong s, Type t)
// with x[] a heap of current size n
// and max size s (i.e. space for s elements allocated)
// insert t and restore heap-property.
//
// Return true if successful
// else (i.e. space exhausted) false
//
// Takes time \log(n)
{
    if ( n > s )  return false;
    ++n;
    Type *x1 = x - 1;
    ulong j = n;
    while ( j > 1 )
    {
        ulong k = (j>>1);  // k==parent(j)
        if ( x1[k] >= t )  break;
        x1[j] = x1[k];
        j = k;
    }
    x1[j] = t;
    return true;
}
```

Similarly, the maximal element can be removed in time $O(\log(n))$:

```
template <typename Type>
Type heap_extract_max(Type *x, ulong n)
// Return maximal element of heap and
```

[1] This is slightly non-obvious, see [Int2Alg] (p.145) for an explanation.

```
// restore heap structure.
// Return value is undefined for 0==n
{
    Type m = x[0];
    if ( 0 != n )
    {
        Type *x1 = x - 1;
        x1[1] = x1[n];
        --n;
        heapify1(x1, n, 1);
    }
    // else error
    return m;
}
```

Two modifications seem appropriate: First, replacement of `extract_max()` by a `extract_next()`, leaving it as an (compile time) option whether to extract the minimal or the maximal element. This is achieved by changing the comparison operators at a few strategic places so that the heap is built either as described above or with its minimum as first element:

```
#if  1
// next() is the one with the smallest key
// i.e.  extract_next()  is  extract_min()
#define _CMP_ <
#define _CMPEQ_ <=
#else
// next() is the one with the biggest key
// i.e.  extract_next()  is  extract_max()
#define _CMP_ >
#define _CMPEQ_ >=
#endif
```

Second, augmenting the elements used by a event-description that can be freely defined. The resulting utility class is [FXT: `class priority_queue` in ds/priorityqueue.h]:

```
template <typename Type1, typename Type2>
class priority_queue
{
public:
    Type1 *t_;  // time:   t[0..s-1]
    Type2 *e_;  // events: e[0..s-1]
    ulong s_;   // allocated size (# of elements)
    ulong n_;   // current number of events
    ulong gq_;  // grow gq elements if necessary, 0 for "never grow"
public:
    priority_queue(ulong n, ulong growq=0)
    {
        s_ = n;
        t_ = new Type1[s_];
        e_ = new Type2[s_];
        n_ = 0;
        gq_ = growq;
    }

    ~priority_queue()  { delete [] t_;  delete [] e_; }

    ulong n()  const  { return n_; }

    Type1 get_next(Type2 &e)  const
    // No check if empty
    {
        e = e_[0];
        return t_[0];
    }

    void heapify1(Type1 *t1, ulong n, ulong k, Type2 *e1)
    // events in         e1[1..n]
    // time of events in t1[1..n]
    {
        ulong m = k;
        ulong l = (k<<1); // left(k);
        if ( (l <= n) && (t1[l] _CMP_ t1[k]) )  m = l;
        ulong r = (k<<1) + 1; // right(k);
        if ( (r <= n) && (t1[r] _CMP_ t1[m]) )  m = r;
        if ( m != k )
        {
            swap2(t1[k], t1[m]);  swap2(e1[k], e1[m]);
            heapify1(t1, n, m, e1);
```

```
        }
    }
    Type1 extract_next(Type2 &e)
    {
        Type1 m = get_next(e);
        if ( 0 != n_ )
        {
            Type1 *t1 = t_ - 1;
            Type2 *e1 = e_ - 1;
            t1[1] = t1[n_];  e1[1] = e1[n_];
            --n_;
            heapify1(t1, n_, 1, e1);
        }
        // else error
        return m;
    }
    bool insert(const Type1 &t, const Type2 &e)
    // Insert event e at time t
    // Return true if successful
    // else (i.e. space exhausted and 0==gq_) false
    {
        if ( n_ >= s_ )
        {
            if ( 0==gq_ )  return false;  // growing disabled
            grow();
        }
        ++n_;
        Type1 *t1 = t_ - 1;
        Type2 *e1 = e_ - 1;
        ulong j = n_;
        while ( j > 1 )
        {
            ulong k = (j>>1);  // k==parent(j)
            if ( t1[k] _CMPEQ_ t )  break;
            t1[j] = t1[k];  e1[j] = e1[k];
            j = k;
        }
        t1[j] = t;
        e1[j] = e;
        return true;
    }
   [-snip-]
```

The second argument of the constructor determines the number of elements added in case of growth, it is disabled (equals zero) by default.

```
   [-snip-]
private:
    void grow()
    {
        ulong ns = s_ + gq_;  // new size
        t_ = ReAlloc<Type1>(t_, ns);
        e_ = ReAlloc<Type2>(e_, ns);
        s_ = ns;
    }
};
```

The `ReAlloc()` routine call the library function `realloc()` when available, else it just copies to newly allocated memory. It is defined in [FXT: file realloc.h].

The program [FXT: demo ds/priorityqueue-demo.cc] inserts events at random times $0 \le t < 1$, then extract all of them. It gives the output shown in figure 4.1 on the next page. A more typical usage would intermix the insertions and extractions.

4.6 Bit-array

The use of *bit-arrays* should be obvious: an array of tag values (like 'seen' versus 'unseen') where all standard data types would be a waste of space. Besides reading and writing individual bits one should

```
      Inserting into piority_queue:          Extracting from piority_queue:
       # :  event  @          time            # :  event  @          time
        0:      A  @      0.840188            10:      F  @      0.197551
        1:      B  @      0.394383             9:      I  @      0.277775
        2:      C  @      0.783099             8:      G  @      0.335223
        3:      D  @       0.79844             7:      B  @      0.394383
        4:      E  @      0.911647             6:      J  @       0.55397
        5:      F  @      0.197551             5:      H  @       0.76823
        6:      G  @      0.335223             4:      C  @      0.783099
        7:      H  @       0.76823             3:      D  @       0.79844
        8:      I  @      0.277775             2:      A  @      0.840188
        9:      J  @       0.55397             1:      E  @      0.911647
```

Figure 4.1: Insertion of events labeled 'A', 'B', ..., 'J' scheduled for random times into a priority queue (left) and subsequent extraction (right).

implement a convenient search for the next set (or cleared) bit.

The class [FXT: `class bitarray` in ds/bitarray.h] is used, for example, for lists of small primes [FXT: file mod/primes.cc], for in-place transposition routines [FXT: file aux2/transpose.h] (see section 2.3 on page 84) and several operations on permutations (see section 2.13.6 on page 105). The constructor allocates memory by default. If the second argument is nonzero it must point to an accessible memory range:

```
    bitarray(ulong nbits, ulong *f=0)
    // nbits must be nonzero
    {
        ulong nw = ctor_core(nbits);
        if ( f!=0 )
        {
            f_ = (ulong *)f;
            myfq_ = false;
        }
        else
        {
            f_ = new ulong[nw];
            myfq_ = true;
        }
    }
```

The public methods are

```
// operations on bit n:
    ulong test(ulong n)  const // Test whether n-th bit set
    void set(ulong n)          // Set n-th bit
    void clear(ulong n)        // Clear n-th bit
    void change(ulong n)       // Toggle n-th bit
    ulong test_set(ulong n)    // Test whether n-th bit is set and set it
    ulong test_clear(ulong n)  // Test whether n-th bit is set and clear it
    ulong test_change(ulong n) // Test whether n-th bit is set and toggle it

// Operations on all bits:
    void clear_all()           // Clear all bits
    void set_all()             // Set all bits
    int all_set_q()  const;    // Return whether all bits are set
    int all_clear_q()  const;  // Return whether all bits are clear

// Scanning the array:
    // Note: the given index n is included in the search
    ulong next_set_idx(ulong n)  const   // Return index of next set or value beyond end
    ulong next_clear_idx(ulong n)  const // Return index of next clear or value beyond end
```

Combined operations like 'test-and-set-bit', 'test-and-clear-bit', 'test-and-change-bit' are often needed in applications that use bit-arrays. This is underlined by the fact that modern CPUs typically have instructions implementing these operations. On the x86 architecture the corresponding CPU instructions are used [FXT: file bits/bitasm-i386.h]. For example,

```
static inline ulong asm_bts(ulong *f, ulong i)
// Bit Test and Set
{
    ulong ret;
    asm ( "btsl %2, %1 \n"
          "sbbl %0, %0"
          : "=r" (ret)
```

```
          : "m" (*f), "r" (i) );
    return ret;
}
```

The equivalent code for the AMD64 architecture is obtained by changing '`btsl`' to '`btsq`' and '`sbbl`' to '`sbbq`', see [FXT: file bits/bitasm-amd64.h]. If no specialized CPU instructions are available macros as the following are used:

```
#define  DIVMOD_TEST(n, d, bm) \
ulong d = n / BITS_PER_LONG; \
ulong bm = 1UL << (n % BITS_PER_LONG); \
ulong t = bm & f_[d];
```

Performance is still good in that case as the modulo operation and division by `BITS_PER_LONG` (a power of two) are replaced with cheap (bit-and and shift) operations. The class does not supply overloading of the array-index operator `[]` because the writing variant would cause a performance penalty. One might want to add 'sparse'-versions of the scan functions for large bit-arrays with only few bits set or unset.

4.7 Finite-state machines

A *finite-state machine* (FSM) (or *state-engine*) in its simplest form can be described as a program that has finite set of valid states and for each state a certain action is taken. In C-syntax:

```
void state_engine(int state)
{
    while ( state != end )
    {
        // valid states are: st1 ... stn (and end)
        switch ( state )
        {
        case st1:  state = func1();  break;
        case st2:  state = func2();  break;
        case st3:  state = func3();  break;
        [--snip--]
        case stn:  state = funcn();  break;
        default:   blue_smoke();  // invalid state
        }
    }
}

int main()
{
    // initialize:
    int state = start;
    state_engine( state );
    return 0;
}
```

As an example we show a state engine that transforms a linear coordinate t into the corresponding pair (x and y) of coordinates of Hilbert's space-filling curve.

Apart from two bits of internal state the FSM processes at each step two bits of input. The array `htab[]` serves as lookup table for the next state plus two bits of the result.

The function ([FXT: `lin2hilbert()` in bits/hilbert.cc]) implements a FSM as suggested in [HAKMEM], item 115:

```
void
lin2hilbert(ulong t, ulong &x, ulong &y)
// Transform linear coordinate t to Hilbert x and y
{
    ulong xv = 0, yv = 0;
    ulong c01 = (0<<2);  // (2<<2) for transposed output (swapped x, y)
    for (ulong i=0; i<(BITS_PER_LONG/2); ++i)
    {
        ulong abi = t >> (BITS_PER_LONG-2);
        t <<= 2;

        ulong st = htab[ (c01<<2) | abi ];
        c01 = st & 3;

        yv <<= 1;
```

```
        yv |= ((st>>2) & 1);
        xv <<= 1;
        xv |= (st>>3);
    }
    x = xv;  y = yv;
}
```

The table used is defined as

```
static const ulong htab[] = {
#define HT(xi,yi,c0,c1) ((xi<<3)+(yi<<2)+(c0<<1)+(c1))
    // index == HT(c0,c1,ai,bi)
    HT( 0, 0,  1, 0 ),
    HT( 0, 1,  0, 0 ),
    HT( 1, 1,  0, 0 ),
    HT( 1, 0,  0, 1 ),
    HT( 1, 1,  1, 1 ),
    HT( 0, 1,  0, 1 ),
    HT( 0, 0,  0, 1 ),
    HT( 1, 0,  0, 0 ),
    HT( 0, 0,  0, 0 ),
    HT( 1, 0,  1, 0 ),
    HT( 1, 1,  1, 0 ),
    HT( 0, 1,  1, 1 ),
    HT( 1, 1,  0, 1 ),
    HT( 1, 0,  1, 1 ),
    HT( 0, 0,  1, 1 ),
    HT( 0, 1,  1, 0 )
#undef HT
};
```

As indicated in the code the table maps every four bits `c0,c1,ai,bi` to four bits `xi,yi,c0,c1`. The original table from [HAKMEM] is (verbatim):

```
    OLD                 NEW
    C   C    A   B      X   Y    C   C
     0   1    I   I      I   I    0   1

    0   0    0   0      0   0    1   0
    0   0    0   1      0   1    0   0
    0   0    1   0      1   1    0   0
    0   0    1   1      1   0    0   1
    0   1    0   0      1   1    1   1
    0   1    0   1      0   1    0   1
    0   1    1   0      0   0    0   1
    0   1    1   1      1   0    0   0
    1   0    0   0      0   0    0   0
    1   0    0   1      1   0    1   0
    1   0    1   0      1   1    1   0
    1   0    1   1      0   1    1   1
    1   1    0   0      1   1    0   1
    1   1    0   1      1   0    1   1
    1   1    1   0      0   0    1   1
    1   1    1   1      0   1    1   0
```

All 16 4-bit words appear in both the left ('OLD') and the right ('NEW') columns. So the algorithm is invertible. Swap the columns and sort numerically to get

```
    NEW                 OLD
    C   C    X   Y      A   B    C   C
     0   1    I   I      I   I    0   1

    0   0    0   0      0   0    1   0
    0   0    0   1      0   1    0   0
    0   0    1   0      1   1    0   1
    0   0    1   1      1   0    0   0
    0   1    0   0      1   0    0   1
    0   1    0   1      0   1    0   1
    0   1    1   0      1   1    0   0
    0   1    1   1      0   0    1   1
    1   0    0   0      0   0    0   0
    1   0    0   1      1   1    1   1
    1   0    1   0      0   1    1   0
    1   0    1   1      1   0    1   0
    1   1    0   0      1   0    1   1
    1   1    0   1      1   1    1   0
    1   1    1   0      0   1    1   1
    1   1    1   1      0   0    0   1
```

Turn that into a table for the inverse function:

```
static const ulong ihtab[] = {
#define IHT(ai,bi,c0,c1) ((ai<<3)+(bi<<2)+(c0<<1)+(c1))
    // index == HT(c0,c1,xi,yi)
    IHT( 0, 0,  1, 0 ),
    IHT( 0, 1,  0, 0 ),
    IHT( 1, 1,  0, 1 ),
    IHT( 1, 0,  0, 0 ),
    IHT( 1, 0,  0, 1 ),
```

```
    IHT( 0, 1,  0, 1 ),
    IHT( 1, 1,  0, 0 ),
    IHT( 0, 0,  1, 1 ),
    IHT( 0, 0,  0, 0 ),
    IHT( 1, 1,  1, 1 ),
    IHT( 0, 1,  1, 0 ),
    IHT( 1, 0,  1, 0 ),
    IHT( 1, 0,  1, 1 ),
    IHT( 1, 1,  1, 0 ),
    IHT( 0, 1,  1, 1 ),
    IHT( 0, 0,  0, 1 )
#undef IHT
};
```

And process the words backwards:

```
ulong
hilbert2lin(ulong x, ulong y)
// Transform Hilbert x and y to linear coordinate t
{
    ulong t = 0;
    ulong c01 = 0;
    for (ulong i=0; i<(BITS_PER_LONG/2); ++i)
    {
        t <<= 2;
        ulong xi = x >> (BITS_PER_LONG/2-1);
        xi &= 1;
        ulong yi = y >> (BITS_PER_LONG/2-1);
        yi &= 1;
        ulong xyi = (xi<<1) | yi;
        x <<= 1;
        y <<= 1;

        ulong st = ihtab[ (c01<<2) | xyi ];
        c01 = st & 3;

        t |= (st>>2);
    }
    return t;
}
```

This function is the inverse of `lin2hilbert()`.

A method to compute the direction (left, right, up or down) at the n-th move of the Hilbert curve is given in section 1.20 on page 47.

4.8 Emulation of coroutines

Sometimes it is possible to find an recursive algorithm for solving some problem that is not easily solved iteratively. However the recursive implementations might produce the results in midst of its calling graph. When a utility class providing the results one by one with some `next` call is required there is an apparent problem: There is only one stack available for function calls[2]. We do not have 'offline' functions or *coroutines*.

As an example consider the following recursive code[3]

```
int n = 4;
int a[n+1], q[2*n+1];

int main()
{
    paren(0, 0);
    return 0;
}

void paren(long i, long s)
{
    long k, t;
    if ( i<n )
```

[2] True for the majority of the programming languages.

[3] given by Glenn Rhoads

```
    {
        for (k=0; k<=i-s;  ++k)
        {
            a[i-1] = k;
            t = s + a[i-1];
            q[t + i] = '(';
            paren(i + 1, t);  // recursion
            q[t + i] = ')';
        }
    }
    else
    {
        a[i-1] = n - s;
        Visit();  // next set of parens available
    }
}
```

that generates following output (the different ways to group four pairs of parenthesis):

```
(((())))
(((())))
((())())
((()))()
(()(()))
(()()())
(()())()
(())(())
(())()()
()((()))
()(()())
()(())()
()()(())
()()()()
```

A reasonable way to create coroutines is to rewrite the function as a state engine and use a class [FXT: `class coroutine` in ds/coroutine.h] that provides two stacks, one for local variables (including arguments) and one for the state of the function:

```
#define CHECK_STACKS  // detect stack overflow
#ifdef CHECK_STACKS
#include "jjassert.h"
#endif // CHECK_STACKS

template <typename Type>
class coroutine
{
public:
    ulong *t_;  // sTate stack
    ulong tp_;  // sTate stack Pointer
    Type  *d_;  // Data stack
    ulong dp_;  // Data stack Pointer
    ulong  n_;  // size of stacks

public:
    coroutine(ulong maxdepth)
    {
        n_ = maxdepth;
        t_ = new ulong[n_];
        d_ = new Type[n_];
        init();
    }

    ~coroutine()
    {
        delete [] d_;
        delete [] t_;
    }

    void init()  { dp_=0; tp_=0; }

#ifdef CHECK_STACKS
    void chk_tp(ulong j=0)  const
    {
        ulong t = tp_ - j;
        jjassert( (long)t>=0 );
        jjassert( t<n_ );
    }

    void chk_dp(ulong j=0)  const
```

```
    {
        ulong t = dp_ - j;
        jjassert( (long)t>=0 );
        jjassert( t<n_ );
    }
#else  // do not check
#define chk_tp(j) {;}
#define chk_dp(j) {;}
#endif // CHECK_STACKS

    // --- state stack:
    void stpush(ulong x)   { chk_tp();  t_[tp_] = x;  ++tp_; }
    ulong stpeek()  const  { chk_tp(1);  return  t_[tp_-1]; }
    void stpoke(ulong x)   { chk_tp(1);  t_[tp_-1] = x; }
    void stpop(ulong ct=1)  { tp_-=ct; }
    void stnext()  { chk_tp(1);  ++t_[tp_-1]; }
    void stnext(ulong x)  { chk_tp(1);  t_[tp_-1] = x; }
    bool more()  const  { return (tp_!=0); }

    // --- stack for variables and args:
    void push(Type x)  {  chk_dp();  d_[dp_] = x;  ++dp_; }
    Type &peek()  { chk_dp(1); return d_[dp_-1]; }
    void pop(ulong ct=1) { dp_-=ct; }

private:
    coroutine & operator = (const coroutine &);  // forbidden
};
#undef CHECK_STACKS
```

Rewriting the function in question (as part of a utility class, given in [FXT: demo ds/coroutine-paren-demo.cc]) only requires the understanding of the language, not of the algorithm. The process is straightforward but needs a bit of concentration:

```
int
paren::next_recursion()
{
 redo:
    vars &v = cr_->peek();
    int &i=v.i,  &s=v.s; // args
    int &k=v.k,  &t=v.t; // locals
 loop:
    switch ( cr_->stpeek() )
    {
    case 0:
        if ( i>=n )
        {
            x[i-1] = n - s;
            cr_->stnext( ST_RETURN );  return 1;
        }
        cr_->stnext();

    case 1:
        // loop end ?
        if ( k>i-s )  { break; }  // shortcut: nothing to do at end

        cr_->stnext();

    case 2: // start of loop body
        {
            long ii = i - 1;
            x[ii] = k;
            t = s + k;

            str[t+i] = '('; // OPEN_CHAR;

            cr_->stnext();
            CORO_CALL( i+1, t, 0, 0 );
            goto redo;
        }

    case 3:
        str[t+i] = ')'; // CLOSE_CHAR;
        ++k;

        // loop end ?
        if ( k>i-s )  { break; }  // shortcut: nothing to do at end

        cr_->stpoke(2);  goto loop; // shortcut: back to loop body

    default: ;
```

```
    }
    CORO_RETURN();
    if ( cr_->more() )  goto redo;
    return 0;
}
```

The constructor pushes the needed variables and parameters on the data stack and the initial state on the state stack:

```
paren::paren(int nn)
{
    n = (nn>0 ? nn : 1);
    x = new int[n+1];
    ++x;
    str = new char[2*n+1];
    int i = 0;
    for (  ; i<n  ; ++i)  str[i] = '('; // OPEN_CHAR;
    for (  ; i<2*n; ++i)  str[i] = ')'; // CLOSE_CHAR;
    str[2*n] = 0;
    cr_ = new coroutine<vars>(n+1);
    //        i, s, k, t
    CORO_CALL( 0, 0, 0, 0 );
    idx = 0;
    q = next_recursion();
}
```

The `CORO_CALL` actually only initializes the data for the state engine, the following call to `next_recursion` then lets the thing run.

The method `next()` of the paren class lets the offline function advance until the next result is available:

```
int paren::next()
{
    if ( 0==q )  return 0;
    else
    {
        q = next_recursion();
        return  ( q ? ++idx : 0 );
    }
}
```

Performance-wise the `coroutine`-rewritten functions are close to the original: state engines are fast and the stack operations are cheap.

The shown method can also be applied when the recursive algorithm consists of more than one function by merging the functions into one state engine.

Further, investigating the contents of the data stack can be of help in the search of a iterative solution.

The code of the converted functions is admittedly ugly but without a generic support of coroutines it seems hard to du much better. One could also use threads for the emulation of coroutines. This approach, however, adds portability problems.

An iterative algorithm for the generation of valid pairings or parenthesis is given in section 5.5 on page 159.

Chapter 5

Combinatorial algorithms

This chapter presents selected combinatorial algorithms. The generation of (all) combinations, compositions, permutations, subsets, integer partitions, set partitions, valid pairings of parentheses and necklaces is treated. For some of these an algorithm for the generation of the objects in a minimal-change (Gray code) order is given.

The routines are useful in situations where an exhaustive search over all configurations of a certain kind is needed. Combinatorial algorithms are also fundamental with many programming problems and finally they can simply be fun!

Several (more esoteric) combinatorial objects that are found via searching in directed graphs are presented in chapter 7.

5.1 Combinations in lexicographic order

```
[ 0  1  2 ]    111...    #  0
[ 0  1  3 ]    11.1..    #  1
[ 0  1  4 ]    11..1.    #  2
[ 0  1  5 ]    11...1    #  3
[ 0  2  3 ]    1.11..    #  4
[ 0  2  4 ]    1.1.1.    #  5
[ 0  2  5 ]    1.1..1    #  6
[ 0  3  4 ]    1..11.    #  7
[ 0  3  5 ]    1..1.1    #  8
[ 0  4  5 ]    1...11    #  9
[ 1  2  3 ]    .111..    # 10
[ 1  2  4 ]    .11.1.    # 11
[ 1  2  5 ]    .11..1    # 12
[ 1  3  4 ]    .1.11.    # 13
[ 1  3  5 ]    .1.1.1    # 14
[ 1  4  5 ]    .1..11    # 15
[ 2  3  4 ]    ..111.    # 16
[ 2  3  5 ]    ..11.1    # 17
[ 2  4  5 ]    ..1.11    # 18
[ 3  4  5 ]    ...111    # 19
```

Figure 5.1: All combinations of three elements out of six in lexicographic order as arry of positions (left column) and delta set (middle column).

The combinations of three elements out of six in *lexicographic* order are shown in figure 5.1. The C++ class [FXT: `class combination_lex` in comb/combination-lex.h] supplies the methods `first()`, `last()`, `next()` and `prev()`. The data is stored as an array $x[\,]$ of length k of values $0 \le x[j] < n$ (leftmost column in figure 5.1).

```
class combination_lex
{
public:
    ulong n_;
    ulong k_;
    ulong *x_;
public:
    combination_lex(ulong n, ulong k)
    {
        n_ = (n ? n : 1);  // not zero
        k_ = (k ? k : 1);  // not zero
        x_ = new ulong[k_];
        first();
    }
    ~combination_lex()  { delete [] x_; }
    void first()
    {
        for (ulong k=0; k<k_; ++k)  x_[k] = k;
    }
    void last()
    {
        for (ulong i=0; i<k_; ++i)  x_[i] = n_ - k_ + i;
    }
```

Going to the next combination is achieved via the method `next()`:

```
    ulong next()  // return zero if previous combination was the last
    {
        if ( x_[0] == n_ - k_ )  { first();  return 0; }
        ulong j = k_ - 1;
        // trivial if highest element != highest possible value:
        if ( x_[j] < (n_-1) )  { ++x_[j];  return 1; }
        // find highest falling edge:
        while ( 1 == (x_[j] - x_[j-1]) )  { --j; }
        // move lowest element of highest block up:
        ulong z = ++x_[j-1];
        // ... and attach rest of block:
        while ( j < k_ )  { x_[j] = ++z;  ++j; }
        return  1;
    }
```

The method `prev()` allows to go to the previous combination in the list:

```
    ulong prev()  // return zero if current combination is the first
    {
        if ( x_[k_-1] == k_-1 )  { last();  return 0; }
        // find highest falling edge:
        ulong j = k_ - 1;
        while ( 1 == (x_[j] - x_[j-1]) )  { --j; }
        --x_[j];  // move down edge element
        // ... and move rest of block to high end:
        while ( ++j < k_ )  x_[j] = n_ - k_ + j;
        return  1;
    }
    const ulong * data()  { return x_; }
    friend ostream & operator << (ostream &os, const combination_lex &x);
};
```

The listing shown in figure 5.1 on the previous page can then be produced by a simple fragment like [FXT: demo comb/combination-lex-demo.cc]:

```
    ulong ct = 0, n = 6,  k = 3;
    combination_lex comb(n, k);
    do
    {
        cout << endl;
        cout << "   [ " << comb << " ]  ";
        print_set_as_bitset("", comb.data(), k, n );
```

```
        cout << "  #" << setw(3) << ct;
        ++ct;
    }
    while ( comb.next() );
```

The routine generates the $225,792,840$ combinations $\binom{32}{20}$ at a rate of about 95 million objects per second. The combinations $\binom{32}{12}$ (same number) are generated at a rate of 160 million objects per second.

5.2 Combinations in co-lexicographic order

```
[ 0  1  2 ]    111...    #  0
[ 0  1  3 ]    11.1..    #  1
[ 0  2  3 ]    1.11..    #  2
[ 1  2  3 ]    .111..    #  3
[ 0  1  4 ]    11..1.    #  4
[ 0  2  4 ]    1.1.1.    #  5
[ 1  2  4 ]    .11.1.    #  6
[ 0  3  4 ]    1..11.    #  7
[ 1  3  4 ]    .1.11.    #  8
[ 2  3  4 ]    ..111.    #  9
[ 0  1  5 ]    11...1    # 10
[ 0  2  5 ]    1.1..1    # 11
[ 1  2  5 ]    .11..1    # 12
[ 0  3  5 ]    1..1.1    # 13
[ 1  3  5 ]    .1.1.1    # 14
[ 2  3  5 ]    ..11.1    # 15
[ 0  4  5 ]    1...11    # 16
[ 1  4  5 ]    .1..11    # 17
[ 2  4  5 ]    ..1.11    # 18
[ 3  4  5 ]    ...111    # 19
```

Figure 5.2: All combinations of three elements out of six in co-lexicographic (colex) order.

The combinations of three elements out of six in *co-lexicographic* ('colex') order are shown in figure 5.2.

The interesting parts of the C++ class [FXT: `class combination_colex` in comb/combination-colex.h] are the methods `next()` and `prev()`:

```
class combination_colex
{
    [-snip-]

    ulong next()  // return zero if previous combination was the last
    {
        if ( x_[0] == n_ - k_ )  { first();  return 0; }

        ulong j = 0;
        // until lowest rising edge ...
        while ( 1 == (x_[j+1] - x_[j]) )
        {
            x_[j] = j;  // attach block at low end
            ++j;
        }

        ++x_[j];  // move edge element up

        return  1;
    }

    ulong prev()  // return zero if current combination is the first
    {
        if ( x_[k_-1] == k_-1 )  { last();  return 0; }

        // find lowest falling edge:
        ulong j = 0;
        while ( j == x_[j] )  ++j;

        --x_[j];  // move edge element down

        // attach rest of low block:
        while ( 0!=j-- )  x_[j] = x_[j+1] - 1;

        return  1;
```

```
    }
    const ulong * data()  { return x_; }
    friend ostream & operator << (ostream &os, const combination_colex &x);
};
```

For the connection between lex-order and colex-order see section 1.27 on page 56. Usage is completely identical to that of the `class combination_lex`, see [FXT: demo comb/combination-colex-demo.cc].

The routine generates the $225,792,840$ combinations $\binom{32}{20}$ at a rate of about 145 million objects per second. The combinations $\binom{32}{12}$ (same number) are generated at a rate of 195 million objects per second.

5.3 Combinations in minimal-change order

```
          ...111   [  0  1  2 ]    swap: (0, 0)   #  0
          ..11.1   [  0  2  3 ]    swap: (3, 1)   #  1
          ..111.   [  1  2  3 ]    swap: (1, 0)   #  2
          ..1.11   [  0  1  3 ]    swap: (2, 0)   #  3
          .11..1   [  0  3  4 ]    swap: (4, 1)   #  4
          .11.1.   [  1  3  4 ]    swap: (1, 0)   #  5
          .111..   [  2  3  4 ]    swap: (2, 1)   #  6
          .1.1.1   [  0  2  4 ]    swap: (3, 0)   #  7
          .1.11.   [  1  2  4 ]    swap: (1, 0)   #  8
          .1..11   [  0  1  4 ]    swap: (2, 0)   #  9
          11...1   [  0  4  5 ]    swap: (5, 1)   # 10
          11..1.   [  1  4  5 ]    swap: (1, 0)   # 11
          11.1..   [  2  4  5 ]    swap: (2, 1)   # 12
          111...   [  3  4  5 ]    swap: (3, 2)   # 13
          1.1..1   [  0  3  5 ]    swap: (4, 0)   # 14
          1.1.1.   [  1  3  5 ]    swap: (1, 0)   # 15
          1.11..   [  2  3  5 ]    swap: (2, 1)   # 16
          1..1.1   [  0  2  5 ]    swap: (3, 0)   # 17
          1..11.   [  1  2  5 ]    swap: (1, 0)   # 18
          1...11   [  0  1  5 ]    swap: (0, 2)   # 19
```

Figure 5.3: All combinations of three elements out of six in a minimal-change order.

The combinations of three elements out of six in a *minimal-change* order are shown in figure 5.3. With each transition exactly one element changes its position. The algorithm used in the utility class [FXT: `class combination_minchange` in comb/combination-minchange.h] is based on inlined versions of the routines that were explained in section 1.30 on page 63. The listing at the beginning of this section can be generated via [FXT: demo comb/combination-minchange-demo.cc].

Knuth [DEK4] gives a more efficient (loopless) algorithm which is used in [FXT: `class combination_revdoor` in comb/combination-revdoor.h]. A demo is [FXT: demo comb/combination-revdoor-demo.cc].

An alternative minimal-change order

An alternative minimal-change order for the combinations $\binom{n}{k}$ can be obtained by using the negated and bit-reversed sequence for the minimal-change combinations $\binom{n-k}{n}$ (see figure 5.4 on the facing page). The interesting feature is that the last combination is identical to the first shifted left by one. This makes it easy to generate the subsets of a set with n elements in monotonic minimal-change order by concatenating the sequences for $k = 1, 2, \ldots, n$.

The usage of the utility class [FXT: `class combination_altminchange` in comb/combination-altminchange.h] is identical to that of the 'standard' minimal-change order. The listing shown in figure 5.4

```
          ...111   [  0  1  2 ]    swap: (3, 0)   #  0
          .1..11   [  0  1  4 ]    swap: (4, 2)   #  1
          1...11   [  0  1  5 ]    swap: (5, 4)   #  2
          ..1.11   [  0  1  3 ]    swap: (5, 3)   #  3
          .11..1   [  0  3  4 ]    swap: (4, 1)   #  4
          1.1..1   [  0  3  5 ]    swap: (5, 4)   #  5
          11...1   [  0  4  5 ]    swap: (4, 3)   #  6
          .1.1.1   [  0  2  4 ]    swap: (5, 2)   #  7
          1..1.1   [  0  2  5 ]    swap: (5, 4)   #  8
          ..11.1   [  0  2  3 ]    swap: (5, 3)   #  9
          .111..   [  2  3  4 ]    swap: (4, 0)   # 10
          1.11..   [  2  3  5 ]    swap: (5, 4)   # 11
          11.1..   [  2  4  5 ]    swap: (4, 3)   # 12
          111...   [  3  4  5 ]    swap: (3, 2)   # 13
          .11.1.   [  1  3  4 ]    swap: (5, 1)   # 14
          1.1.1.   [  1  3  5 ]    swap: (5, 4)   # 15
          11..1.   [  1  4  5 ]    swap: (4, 3)   # 16
          .1.11.   [  1  2  4 ]    swap: (5, 2)   # 17
          1..11.   [  1  2  5 ]    swap: (5, 4)   # 18
          ..111.   [  1  2  3 ]    swap: (5, 3)   # 19
```

Figure 5.4: All combinations of three elements out of six in an alternative minimal-change order.

can be produced via [FXT: demo comb/combination-altminchange-demo.cc]:

5.4 A recursive routine for combinations

Consider the directed graph with $n+1$ nodes labeled -1, 0, 1, ..., $n-1$. Each node x has edges to every node y with $y > x$. The paths of length $k+1$ starting at node -1 correspond to the combinations $\binom{n}{k}$.

We develop a recursive algorithm that finds all paths of length $k+1$ whose second, third, ..., last component are the representation of a combination. A first attempt might be as follows:

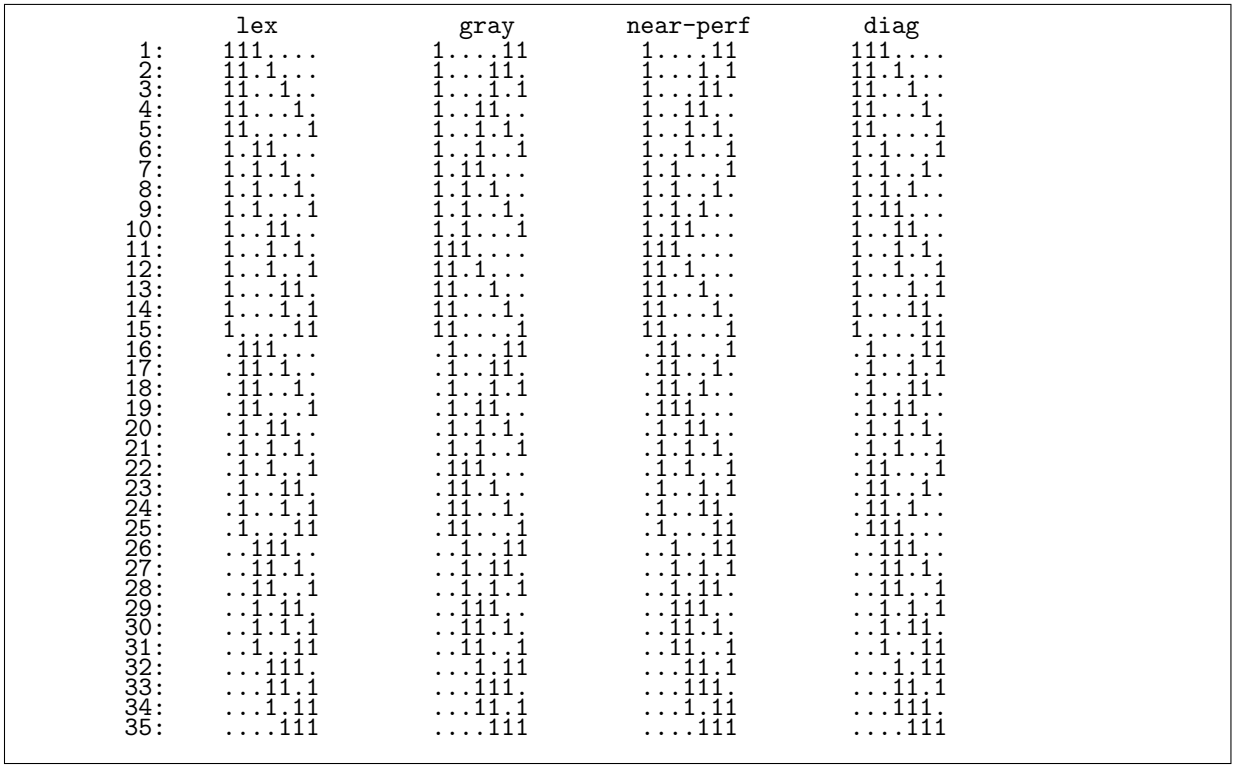
```
ulong n, k; // (n choose k)
ulong *rv;  // record of visits in graph

void next_rec(ulong d)
{
    ulong r = k - d;  // number of elements remaining
    ulong rv1 = rv[d-1];  // left neighbor

    if ( 0==r )
    {
        visit();
    }
    else
    {
        for (ulong x=rv1+1; x<n; ++x)
        {
            rv[d] = x;
            next_rec(d+1);
        }
    }
}
```

The routine `visit()` shall process the combination, for example, print it. We need to initialize the node -1:

```
int main()
{
    n = 7;  k = 3;
    rv = new ulong[k+1];
    ++rv;
    rv[-1] = -1UL;
    next_rec(0);
    return 0;
}
```



	lex	gray	near-perf	diag
1:	111....	1...11	1...11	111....
2:	11.1...	1...11.	1...1.1	11.1...
3:	11..1..	1...1.1	1...11.	11..1..
4:	11...1.	1..11..	1..11..	11...1.
5:	11....1	1..1.1.	1..1.1.	11....1
6:	1.11...	1..1..1	1..1..1	1.1...1
7:	1.1.1..	1.11...	1.1...1	1.1..1.
8:	1.1..1.	1.1.1..	1.1..1.	1.1.1..
9:	1.1...1	1.1..1.	1.1.1..	1.11...
10:	1..11..	1.1...1	1.11...	1..11..
11:	1..1.1.	111....	111....	1..1.1.
12:	1..1..1	11.1...	11.1...	1..1..1
13:	1...11.	11..1..	11..1..	1...1.1
14:	1...1.1	11...1.	11...1.	1...11.
15:	1....11	11....1	11....1	1....11
16:	.111...	.1...11	.11...1	.1...11
17:	.11.1..	.1..11.	.11..1.	.1..1.1
18:	.11..1.	.1..1.1	.11.1..	.1..11.
19:	.11...1	.1.11..	.111...	.1.11..
20:	.1.11..	.1.1.1.	.1.11..	.1.1.1.
21:	.1.1.1.	.1.1..1	.1.1.1.	.1.1..1
22:	.1.1..1	.111...	.1.1..1	.11...1
23:	.1..11.	.11.1..	.1..1.1	.11..1.
24:	.1..1.1	.11..1.	.1..11.	.11.1..
25:	.1...11	.11...1	.1...11	.111...
26:	..111..	..1..11	..1..11	..111..
27:	..11.1.	..1.11.	..1.1.1	..11.1.
28:	..11..1	..1.1.1	..1.11.	..11..1
29:	..1.11.	..111..	..111..	..1.1.1
30:	..1.1.1	..11.1.	..11.1.	..1.11.
31:	..1..11	..11..1	..11..1	..1..11
32:	...111.	...1.11	...11.1	...1.11
33:	...11.1	...111.	...111.	...11.1
34:	...1.11	...11.1	...1.11	...111.
35:	111	111	111	111

Figure 5.5: All combinations of three out of seven in lexicographic, minimal-change, near-perfect, and diagonal minimal-change order (from left to right).

The output of the program are the combinations in lexicographic order (left column in figure 5.5). However, as given, the routine is quite ineffective. We do not need to examine paths where the d-th element is greater than $n-r$, none of them gives a combination. Therefore we change the loop as follows

```
for (ulong x=rv1+1; x<n; ++x)     // works
for (ulong x=rv1+1; x<=n-r; ++x)  // works better!
```

We further have to initialize the sentinel element in the array `rv[]`:

```
rv = new ulong[k+1];
++rv;
rv[-1] = -1UL;
```

A final modification in the `next_rec()` routine is to replace the loop of recursive calls by

```
bool q;
switch ( rq )
{
case 0:  q = 1;  break;   // lex order
case 1:  q = !(d&1);  break;   // gray order
case 2:  q = rv1&1;  break;  // near-perfect order
case 3:  q = (d^rv1)&1;  break;  // diagonal gray order
}

if ( nq )  q = !q;

if ( q )
{
    for (ulong x=rv1+1; x<=n-r; ++x)  // forward
    {
        rv[d] = x;
        next_rec(d+1);
    }
}
else
{
    for (ulong x=n-r; (long)x>=(long)rv1+1; --x)  // backward
    {
```

```
            rv[d] = x;
            next_rec(d+1);
        }
    }
```

The global parameter `rq` determines which condition is used to select the forward- or backward loop. Using the values 0, 1, 2, 3 we obtain the orders shown in figure 5.5 on the facing page. Setting the global parameter `nq` to 'true' reverses the order. All orders that can be obtained with the routine have in common that combinations with identical prefixes appear in succession.

The quantity $\rho(n,k)$ does not depend from the particular order so the algorithm remains CAT. Given the efficiency of the algorithm it is hardly worth to create an iterative version. Tail recursion can be obtained by unifying the two loops in the recursion as follows:

```
    long x0=rv1+1, x1=n-r, i=1;  // forward
    if ( !q )  { swap2(x0, x1);  i=-i; }  // backward
    x1 += i;
    for (long x=x0; (long)x!=x1; x+=i)
    {
        rv[d] = x;
        next_rec(d+1);
    }
```

The C++ class [FXT: `class comb_rec` in comb/combination-rec.h] implements the shown ideas and is easy to use. The data shown in figure 5.5 on the preceding page was produced via [FXT: demo comb/combination-rec-demo.cc]:

```
void visit(const comb_rec &C)
{
    ulong *rv = C.rv;
    cout << setw(4) << C.ct << ":";
    print_set_as_bitset("    ", rv, C.k, C.n);
    cout << "   ";
    for (ulong j=0; j<C.k; ++j)  cout << " " << rv[j];
    cout << endl;
}

int
main(int argc, char **argv)
{
    ulong n = 7;
    NXARG(n, "Combination (n choose k)");
    ulong k = 3;
    NXARG(k, " 0 < k <= n");
    ulong rq = 2;
    NXARG(rq, "Order: 0==>lex,  1==>Gray,  2==>near-perfect,  3==>diagonal Gray");
    ulong nq = 0;
    NXARG(nq, "Whether to reverse order.");

    comb_rec C(n,k);
    C.generate(visit, rq, nq);
    return 0;
}
```

The routine generates the $225,792,840$ combinations $\binom{32}{20}$ at a rate of about 32 million objects per second. The combinations $\binom{32}{12}$ (same number) are generated at a rate of 45 million objects per second.

5.5 Parenthesis strings and restricted growth strings

An iterative scheme to generate all valid ways to group parenthesis can be obtained from a modified version of the combinations in co-lexicographic order (see section 5.2 on page 155). For $n = 5$ pairs the possible combinations are shown in figure 5.6 on the next page. This is the output of [FXT: demo comb/paren-demo.cc].

Now consider the sequences to the right of the paren strings as binary words. Whenever the leftmost block has more than one bit then its rightmost bit is moved one position to the right. If the leftmost block consists of a single bit then the bits of the longest run of the repeated pattern '`1.`' at the left are gathered at the left end and further, the rightmost bit in next block of ones (which contains at least two

```
(((((())))) 11111.....  #  0      (()())(())  11..11.1..  # 21
((((()))()) 1111.1....  #  1      (()()()())  1.1.11.1..  # 22
((((())())) 111.11....  #  2      (())((()))  111...11..  # 23
((((()))()) 11.111....  #  3      (())(()())  11.1..11..  # 24
((((())))() 1.1111....  #  4      (())(())()  1.11..11..  # 25
(((()(()))) 1111..1...  #  5      (())()(())  11..1.11..  # 26
(((()()())) 111.1.1...  #  6      (())()()()  1.1.1.11..  # 27
(((()))()() 11.11.1...  #  7      ()((((())))  1111....1.  # 28
(((()()))() 1.111.1...  #  8      ()(((()()))  111.1...1.  # 29
(((())(()))  111..11...  #  9      ()(((())())  11.11...1.  # 30
(((())()()) 11.1.11...  # 10      ()(((()))()  1.111...1.  # 31
(((())()))() 1.11.11...  # 11      ()(()(()))  111..1..1.  # 32
(((()))(())  11..111...  # 12      ()(()()())  11.1.1..1.  # 33
(((()))()() 1.1.111...  # 13      ()(())(())  1.11.1..1.  # 34
(()((()))) 1111...1..  # 14      ()(())()()  11..11..1.  # 35
(()(()())) 111.1..1..  # 15      ()(()())()  1.1.11..1.  # 36
(()(()))() 11.11..1..  # 16      ()()((()))  111...1.1.  # 37
(()(()))()  1.111..1..  # 17      ()()(()())  11.1..1.1.  # 38
(()()(())) 111..1.1..  # 18      ()()(())()  1.11..1.1.  # 39
(()()()()) 11.1.1.1..  # 19      ()()()(())  11..1.1.1.  # 40
(()()())()  1.11.1.1..  # 20      ()()()()()  1.1.1.1.1.  # 41
```

Figure 5.6: All (42) valid strings of 5 pairs of parenthesis in colex order.

ones) is moved by one position to the right and the rest of the block is gathered at the left end (see the transitions from #13 to #14 or #36 to #37).

The code of the resulting utility class [FXT: `class paren` in comb/paren.h] is quite concise:

```
class paren
// parentheses by a modified comb_colex procedure
{
public:
    ulong k_;  // Number of paren pairs
    ulong n_;  // ==2*k
    ulong *x_; // Positions where a opening paren occurs
    char *str_;  // String representation,  e.g. "(((())))()"
public:
    paren(ulong k)
    {
        k_ = (k>1 ? k : 2);  // not zero (empty) or one (trivial: "()")
        n_ = 2 * k_;
        x_ = new ulong[k_ + 1];
        x_[k_] = 999;  // sentinel
        str_ = new char[n_ + 1];
        str_[n_] = 0;
        first();
    }

    ~paren()
    {
        delete [] x_;
        delete [] str_;
    }

    void first()  { for (ulong i=0; i<k_; ++i)  x_[i] = i; }

    void last()  { for (ulong i=0; i<k_; ++i)  x_[i] = 2*i; }

    ulong next()  // return zero if current paren is the last
    {
        // if ( k_==1 )  return 0;  // uncomment to make algorithm work for k_==1

        ulong j = 0;
        if ( x_[1] == 2 )
        {
            // scan for low end == 010101:
            j = 2;
            while ( (j<=k_) && (x_[j]==2*j) )  ++j;  // can touch sentinel
            if ( j==k_ )  {  first();  return 0; }
        }

        // scan block:
        while ( 1 == (x_[j+1] - x_[j]) )  { ++j; }
```

```
        ++x_[j];  // move edge element up
        for (ulong i=0; i<j; ++i)  x_[i] = i; // attach block at low end
        return 1;
    }
    ulong prev()  // return zero if current paren is the first
    {
        // if ( k_==1 )  return 0;  // uncomment to make algorithm work for k_==1
        ulong j = 0;
        // scan for first gap:
        while ( x_[j]==j )  ++j;
        if ( j==k_ )  {  last();  return 0; }
        if ( x_[j]-x_[j-1] == 2 )  --x_[j];  // gap of length one
        else
        {
            ulong i = --x_[j];
            --j;
            --i;
            // j items to go, distribute as 1.1.1.11111
            for (  ;  2*i>j;  --i,--j)  x_[j] = i;
            for (  ; i; --i)  x_[i] = 2*i;
            x_[0] = 0;
        }
        return 1;
    }
    const ulong * data()  { return x_; }
    [-snip-]
    const char * string()  // generate on demand
    {
        for (ulong j=0; j<n_; ++j)  str_[j] = ')';
        for (ulong j=0; j<k_; ++j)  str_[x_[j]] = '(';
        return str_;
    }
};
```

The $477,638,700$ paren words for $n = 18$ are generated in about 7.5 seconds, corresponding to a rate of more than 60 million objects per second.

The number of parenthesis strings: Catalan numbers

The number of valid combinations of n parenthesis pairs is

$$C_n \;=\; \frac{\binom{2n}{n}}{n+1} \;=\; \frac{\binom{2n+1}{n}}{2n+1} \;=\; \frac{\binom{2n-2}{n-1}}{n} \;=\; \binom{2n}{n} - \binom{2n}{n-1} \tag{5.1}$$

as nicely explained in (chapter 7.5, example 4 on page 343 of) [ConcMath]. These are the Catalan numbers, sequence A000108 of [OEIS]:

n	C_n	n	C_n	n	C_n
1	1	11	58786	21	24466267020
2	2	12	208012	22	91482563640
3	5	13	742900	23	343059613650
4	14	14	2674440	24	1289904147324
5	42	15	9694845	25	4861946401452
6	132	16	35357670	26	18367353072152
7	429	17	129644790	27	69533550916004
8	1430	18	477638700	28	263747951750360
9	4862	19	1767263190	29	1002242216651368
10	16796	20	6564120420	30	3814986502092304

The Catalan numbers are generated most easily with the relation

$$C_{n+1} \;=\; \frac{2\,(2\,n+1)}{n+2}\,C_n \tag{5.2}$$

Using pari/gp:

```
? N=10; C=vector(N); c=1; for(n=1, N, C[n]=c; c*=2*(2*n+1)/(n+2)); C
  [1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796]
```

The generating function is

$$\frac{1-\sqrt{1-4\,x}}{2\,x} \;=\; \sum_{n=0}^{\infty} C_n\,x^n \;=\; 1 + x + 2\,x^2 + 5\,x^3 + 14\,x^4 + 42\,x^5 + \ldots \tag{5.3}$$

One further has the convolutional property

$$C_n \;=\; \sum_{k=0}^{n-1} C_k\,C_{n-1-k} \tag{5.4}$$

Section 1.32 on page 67 gives a bit-level algorithm for the generation of the paren words in colex order.

Restricted growth strings

```
         1:  as[ 0, 0, 0, 0, ]    ()()()()    1.1.1.1.
         2:  as[ 0, 0, 0, 1, ]    ()()(())    1.1.11..
         3:  as[ 0, 0, 1, 0, ]    ()(())()    1.11..1.
         4:  as[ 0, 0, 1, 1, ]    ()(()())    1.11.1..
         5:  as[ 0, 0, 1, 2, ]    ()((()))    1.111...
         6:  as[ 0, 1, 0, 0, ]    (())()()    11..1.1.
         7:  as[ 0, 1, 0, 1, ]    (())(())    11..11..
         8:  as[ 0, 1, 1, 0, ]    (()())()    11.1..1.
         9:  as[ 0, 1, 1, 1, ]    (()()())    11.1.1..
        10:  as[ 0, 1, 1, 2, ]    (()(()))    11.11...
        11:  as[ 0, 1, 2, 0, ]    ((()))()    111...1.
        12:  as[ 0, 1, 2, 1, ]    ((())())    111..1..
        13:  as[ 0, 1, 2, 2, ]    ((()()))    111.1...
        14:  as[ 0, 1, 2, 3, ]    ((((?)))    1111....
```

Figure 5.7: Restricted growth strings (array '`as[]`') and the corresponding paren pairs and delta sets.

Objects that are counted by the Catalan numbers can be represented by sequences a_0, a_1, …, a_n where $a_0 = 0$ and $a_k \le a_{k-1} + 1$. These sequences are examples of so-called *restricted growth string* (RGS). Some sources use the term *restricted growth function*. The RGSs for $n = 4$ are shown in figure 5.7. A RGS can be incremented by incrementing the highest (rightmost in figure 5.7) digit a_j where $a_j \le a_{j-1}$ and setting $a_i = 0$ for all $i > j$. A decrement is obtained by decrementing the highest digit $a_j \ne 0$ and setting $a_i = a_{i-1} + 1$ for all $i > j$.

Using [FXT: `class catalan` in comb/catalan.h] one can generate all RGSs for a given n, by default in minimal-change order:

```
class catalan
// Catalan restricted growth strings (RGS)
// By default in near-perfect minimal change order, i.e.
//  exactly two symbols in paren string change with each step
{
public:
    int *as_;   // digits of the RGS: as_[k] <= as[k-1] + 1
    int *d_;    // direction with recursion (+1 or -1)
    ulong n_;   // Number of digits (paren pairs)
    char *str_; // paren string
```

```
    bool xdr_;  // whether to change direction in recursion (==> minimal change order)
    int dr0_;   // dr0: starting direction in each recursive step:
    //   dr0=+1  ==> start with as[]=[0,0,0,...,0]   == "()()()...()"
    //   dr0=-1  ==> start with as[]=[0,1,2,...,n-1] == "((( ... )))"
public:
    catalan(ulong n, bool xdr=true, int dr0=+1)
        : n_(n)
    {
        as_ = new int[n_];
        d_ = new int[n_];
        str_ = new char[2*n_+1];  str_[2*n_] = 0;
        init(xdr, dr0);
    }

    ~catalan()
    {
        delete [] as_;
        delete [] d_;
        delete [] str_;
    }

    void init(bool xdr, int dr0)
    {
        dr0_ = ( (dr0>=0) ? +1 : -1 );
        xdr_ = xdr;

        ulong n = n_;
        if ( dr0_>0 )  for (ulong k=0; k<n; ++k)  as_[k] = 0;
        else           for (ulong k=0; k<n; ++k)  as_[k] = k;

        for (ulong k=0; k<n; ++k)  d_[k] = dr0_;
    }

    bool next()  { return next_rec(n_-1); }

    const int *get()  const  { return as_; }

    const char* str()  { make_str();  return (const char*)str_; }

   [-snip-]
```

The minimal change order is obtained by changing the 'direction' in the recursion, an essentially identical mechanism (for the generation of set partitions) is presented in section 5.7 on page 168. The function is given in [FXT: file comb/catalan.cc]:

```
int
catalan::next_rec(ulong k)
{
    if ( k<1 )  return 0;  // current is last

    int d = d_[k];
    int as = as_[k] + d;
    bool ovq = ( (d>0) ? (as>as_[k-1]+1) : (as<0) );
    if ( ovq )  // have to recurse
    {
        ulong ns1 = next_rec(k-1);
        if ( 0==ns1 )  return 0;

        d = ( xdr_ ? -d : dr0_ );
        d_[k] = d;

        as = ( (d>0) ? 0 : as_[k-1]+1 );
    }
    as_[k] = as;

    return 1;
}
```

The program [FXT: demo comb/catalan-demo.cc] demonstrates the usage:

```
    ulong n = 4;
    bool xdr = true;
    int dr0 = -1;

    catalan cat(n, xdr, dr0);

    ulong ct = 0;
    do
    {
        ++ct;
        cout << setw(3) << ct << ":  ";
        cat.print_internal();
```

```
    // print paren string:
    const char *str = cat.str();
    cout << "   " << str;

    // print delta set:
    cout << "   ";
    for (ulong k=0; k<2*n; ++k)  cout << ( str[k]=='(' ? '1' : '.' );

    cout << endl;
}
while ( cat.next() );

cout << " C=" << ct;  // catalan number n
cout << endl;
```

Figure 5.8 shows two minimal-change orders for $n = 4$ that were produced with `dr0=+1` (top) and `dr0=-1` (bottom).

```
 1:  as[ 0, 0, 0, 0, ]    d[ +, +, +, +, ]    ()()()()    1.1.1.1.
 2:  as[ 0, 0, 0, 1, ]    d[ +, +, +, +, ]    ()()(())    1.1.11..
 3:  as[ 0, 0, 1, 2, ]    d[ +, +, +, -, ]    ()((()))    1.111...
 4:  as[ 0, 0, 1, 1, ]    d[ +, +, +, -, ]    ()(()())    1.11.1..
 5:  as[ 0, 0, 1, 0, ]    d[ +, +, +, -, ]    ()(())()    1.11..1.
 6:  as[ 0, 1, 2, 0, ]    d[ +, +, -, +, ]    ((()))()    111...1.
 7:  as[ 0, 1, 2, 1, ]    d[ +, +, -, +, ]    ((())())    111..1..
 8:  as[ 0, 1, 2, 2, ]    d[ +, +, -, +, ]    (((()))    111.1...
 9:  as[ 0, 1, 2, 3, ]    d[ +, +, -, +, ]    ((((0)))    1111....
10:  as[ 0, 1, 1, 2, ]    d[ +, +, -, -, ]    (()((0))    11.11...
11:  as[ 0, 1, 1, 1, ]    d[ +, +, -, -, ]    (()()())    11.1.1..
12:  as[ 0, 1, 1, 0, ]    d[ +, +, -, -, ]    (()())()    11.1..1.
13:  as[ 0, 1, 0, 0, ]    d[ +, +, -, +, ]    (())()()    11..1.1.
14:  as[ 0, 1, 0, 1, ]    d[ +, +, -, +, ]    (())(())    11..11..

 1:  as[ 0, 1, 2, 3, ]    d[ -, -, -, -, ]    ((((0)))    1111....
 2:  as[ 0, 1, 2, 2, ]    d[ -, -, -, -, ]    (((()))    111.1...
 3:  as[ 0, 1, 2, 1, ]    d[ -, -, -, -, ]    ((())())    111..1..
 4:  as[ 0, 1, 2, 0, ]    d[ -, -, -, -, ]    ((()))()    111...1.
 5:  as[ 0, 1, 1, 0, ]    d[ -, -, -, +, ]    (()())()    11.1..1.
 6:  as[ 0, 1, 1, 1, ]    d[ -, -, -, +, ]    (()()())    11.1.1..
 7:  as[ 0, 1, 1, 2, ]    d[ -, -, -, +, ]    (()((0))    11.11...
 8:  as[ 0, 1, 0, 1, ]    d[ -, -, -, -, ]    (())(())    11..11..
 9:  as[ 0, 1, 0, 0, ]    d[ -, -, -, -, ]    (())()()    11..1.1.
10:  as[ 0, 0, 0, 0, ]    d[ -, -, +, +, ]    ()()()()    1.1.1.1.
11:  as[ 0, 0, 0, 1, ]    d[ -, -, +, +, ]    ()()(())    1.1.11..
12:  as[ 0, 0, 1, 2, ]    d[ -, -, +, -, ]    ()((()))    1.111...
13:  as[ 0, 0, 1, 1, ]    d[ -, -, +, -, ]    ()(()())    1.11.1..
14:  as[ 0, 0, 1, 0, ]    d[ -, -, +, -, ]    ()(())()    1.11..1.
```

Figure 5.8: Two minimal-change orders for the paren strings of 4 pairs corresponding to `dr0=+1` (top) and `dr0=-1` (bottom). Both are near-perfect orders: the changes occur either in adjacent positions or with distance two.

For even values of n it is possible to generate paren strings in *strong minimal-change order* where changes occur only in adjacent positions. Figure 5.9 on the facing page shows an example for four pairs of parens. The listing was generated with [FXT: demo graph/graph-parengray-demo.cc] that uses directed graphs and the search algorithms described in chapter 7 on page 233.

5.6 Integer partitions

An integer x is the sum of the positive integers less or equal to itself in various ways. The decompositions into sums of integers are called the *integer partitions* of the number x. Figure 5.10 on the facing page shows all integer partitions of $x = 6$. We want to attack a slightly more general problem and find all partitions of a number x with respect to a set $V = \{v_0, v_1, \ldots, v_{n-1}\}$ (where $v_i > 0$), that is all decompositions of the form $x = \sum_{k=0}^{n-1} c_k \cdot v_k$ (where $c_i \geq 0$).

```
 0:   ....1111 == (((())))
 1:   ...1.111 == (((()))   ^= ...11...
 2:   ...11.11 == ((()())   ^= ....11..
 3:   ...111.1 == ()((()))  ^= .....11.
 4:   ..1.11.1 == ()(()())  ^= ..11....
 5:   ..1.1.11 == (()()())  ^= .....11.
 6:   ..1..111 == (()(()))  ^= ....11..
 7:   .1...111 == ((()))()  ^= .11.....
 8:   .1..1.11 == (()())()  ^= ....11..
 9:   .1..11.1 == ()(())()  ^= .....11.
10:   .1.1.1.1 == ()()()()  ^= ...11...
11:   ..11.1.1 == ()()(())  ^= .11.....
12:   ..11..11 == ()(())()  ^= .....11.
13:   .1.1..11 == ()()()()  ^= .11.....
```

Figure 5.9: Strong minimal-change order for the paren strings of 4 pairs: all changes occur in adjacent positions.

```
 # 0:    6 ==  6* 1 + 0    + 0    + 0    + 0    + 0     ==   1 + 1 + 1 + 1 + 1 + 1
 # 1:    6 ==  4* 1 + 1* 2 + 0    + 0    + 0    + 0     ==   1 + 1 + 1 + 1 + 2
 # 2:    6 ==  2* 1 + 2* 2 + 0    + 0    + 0    + 0     ==   1 + 1 + 2 + 2
 # 3:    6 ==  0    + 3* 2 + 0    + 0    + 0    + 0     ==   2 + 2 + 2
 # 4:    6 ==  3* 1 + 0    + 1* 3 + 0    + 0    + 0     ==   1 + 1 + 1 + 3
 # 5:    6 ==  1* 1 + 1* 2 + 1* 3 + 0    + 0    + 0     ==   1 + 2 + 3
 # 6:    6 ==  0    + 0    + 2* 3 + 0    + 0    + 0     ==   3 + 3
 # 7:    6 ==  2* 1 + 0    + 0    + 1* 4 + 0    + 0     ==   1 + 1 + 4
 # 8:    6 ==  0    + 1* 2 + 0    + 1* 4 + 0    + 0     ==   2 + 4
 # 9:    6 ==  1* 1 + 0    + 0    + 0    + 1* 5 + 0     ==   1 + 5
 #10:    6 ==  0    + 0    + 0    + 0    + 0    + 1* 6  ==   6
```

Figure 5.10: All (eleven) integer partitions of 6.

A C++ class for the generation of all partitions is [FXT: `class partition` in comb/partition.h]:

```
class partition
// Integer partitions of x into supplied values pv[0],...,pv[n-1].
// pv[] defaults to [1,2,3,...,x]
{
public:
    ulong ct_;  // Number of of partitions found so far
    ulong n_;   // Number of values
    ulong i_;   // level in iterative search

    long *pv_;  // values into which to partition
    ulong *pc_; // multipliers for values
    ulong pci_; // temporary for pc_[i_]
    long *r_;   // rest
    long ri_;   // temporary for r_[i_]
    long x_;    // value to partition

public:
    partition(ulong x, ulong n=0, const ulong *pv=0)
    {
        if ( 0==n )  n = x;
        n_ = n;
        pv_ = new long[n_+1];
        if ( pv )  for (ulong j=0; j<n_; ++j)  pv_[j] = pv[j];
        else       for (ulong j=0; j<n_; ++j)  pv_[j] = j + 1;
        pc_ = new ulong[n_+1];
        r_  = new long[n_+1];
        init(x);
    }

    void init(ulong x)
    {
        x_ = x;
        ct_ = 0;
        for (ulong k=0; k<n_; ++k)  pc_[k] = 0;
        for (ulong k=0; k<n_; ++k)  r_[k] = 0;
        r_[n_-1] = x_;
        r_[n_] = x_;
        i_ = n_ - 1;
        pci_ = 0;
```

```
        ri_ = x_;
    }

    ~partition()
    {
        delete [] pv_;
        delete [] pc_;
        delete [] r_;
    }

    ulong next();  // generate next partition
    ulong next_func(ulong i);  // aux

    ulong count(ulong x);  // count number of partitions
    ulong count_func(ulong i);  // aux

    [-snip-]
};
```

The algorithm to generate the partitions is to assign to the first bucket r_0 an integer multiple of the first set element v_0: $r_0 = c \cdot v_0$ (this has to be done for all $c \geq 0$ for which $r_0 \leq x$). Now set $c_0 = c$. If $r_0 = x$ we already found a partition (consisting of c_0 only), else (if $r_0 < x$) solve the remaining problem where $x' := x - c_0 \cdot v_0$ and $V' := \{v_1, v_2, \ldots, v_{n-1}\}$.

The algorithm is implemented in [FXT: file comb/partition.cc]. A recursive method is used to count the partitions:

```
ulong
partition::count(ulong x)
{
    init(x);
    count_func(n_-1);
    return ct_;
}

ulong
partition::count_func(ulong i)
{
    if ( 0!=i )
    {
        while ( r_[i]>0 )
        {
            pc_[i-1] = 0;
            r_[i-1] = r_[i];
            count_func(i-1);  // recursion
            r_[i] -= pv_[i];
            ++pc_[i];
        }
    }
    else  // recursion end
    {
        if ( 0!=r_[i] )
        {
            long d = r_[i] / pv_[i];
            r_[i] -= d * pv_[i];
            pc_[i] = d;
        }
    }

    if ( 0==r_[i] )  // valid partition found
    {
        // if ( whatever )  ++ct_;  // restricted count
        ++ct_;
        return 1;
    }
    else  return 0;
}
```

An iterative version of the algorithm can supply the partitions one by one:

```
ulong
partition::next()
{
    if ( i_>=n_ )  return  n_;

    r_[i_] = ri_;
    pc_[i_] = pci_;
    i_ = next_func(i_);
```

```
    for (ulong j=0; j<i_; ++j)  pc_[j] = r_[j] = 0;
    ++i_;
    ri_ = r_[i_] - pv_[i_];
    pci_ = pc_[i_] + 1;
    return  i_ - 1;  // >=0
}

ulong
partition::next_func(ulong i)
{
 start:
    if ( 0!=i )
    {
        while ( r_[i]>0 )
        {
            pc_[i-1] = 0;
            r_[i-1] = r_[i];
            --i;  goto start;   // iteration
        }
    }
    else  // iteration end
    {
        if ( 0!=r_[i] )
        {
            long d = r_[i] / pv_[i];
            r_[i] -= d * pv_[i];
            pc_[i] = d;
        }
    }
    if ( 0==r_[i] )  // valid partition found
    {
        ++ct_;
        return i;
    }
    ++i;
    if ( i>=n_ )  return n_;  // search finished
    r_[i] -= pv_[i];
    ++pc_[i];
    goto start;  // iteration
}
```

The routines can easily adapted to the generation of partitions satisfying certain restrictions, for example, partitions into unequal parts (that is, $c_i \leq 1$).

The listing shown in figure 5.10 on page 165 can be generated with [FXT: demo comb/partition-demo.cc]:

```
void
print_part(ulong n)
{
        partition pp(n);
        ulong ct = 0;
        while ( pp.next() < n )
        {
            cout << "    #" << setw(2) << ct << ": ";
            cout << setw(4) << pp.x_ << " == ";
            pp.print2();
            cout << "  ==  ";
            pp.print();
            cout << endl;
            ++ct;
        }
        cout << " " << n << ":  ct=" << ct << endl;
        cout << endl;
}
```

The 190,569,292 partitions of 100 are generated in less than 11 seconds, corresponding to a rate of about 18 million partitions per second.

The number of integer partitions

The total number of integer partitions of n is sequence A000041 of [OEIS]. For $1 \le x \le 50$:

n	P_n	n	P_n	n	P_n	n	P_n	n	P_n
1	1	11	56	21	792	31	6842	41	44583
2	2	12	77	22	1002	32	8349	42	53174
3	3	13	101	23	1255	33	10143	43	63261
4	5	14	135	24	1575	34	12310	44	75175
5	7	15	176	25	1958	35	14883	45	89134
6	11	16	231	26	2436	36	17977	46	105558
7	15	17	297	27	3010	37	21637	47	124754
8	22	18	385	28	3718	38	26015	48	147273
9	30	19	490	29	4565	39	31185	49	173525
10	42	20	627	30	5604	40	37338	50	204226

The numbers P_n can be computed efficiently with [FXT: demo comb/numof-partitions-demo.cc].

The generating function for the number of integer partitions is

$$\frac{1}{\prod_{k=1}^{\infty}(1-x^k)} \quad = \quad \sum_{n=0}^{\infty} P_n\, x^n \tag{5.5}$$

```
? N=10;default(seriesprecision,N+1);taylor(1/prod(k=1,N,1-x^k),x)
  1 + x + 2*x^2 + 3*x^3 + 5*x^4 + 7*x^5 + \
    + 11*x^6 + 15*x^7 + 22*x^8 + 30*x^9 + 42*x^10 + O(x^11)
```

5.7 Set partitions

For a set of n elements, say $S_n := \{1, 2, \ldots, n\}$, a *set partition* is a set $P = \{s_1, s_2, \ldots, s_k\}$ of non-empty subsets s_i of S_n whose intersection is empty and whose union equals S.

For example, there are 5 set partitions of the set $S_3 = \{1, 2, 3\}$:

```
1:  { {1, 2, 3} }
2:  { {1, 2}, {3} }
3:  { {1, 3}, {2} }
4:  { {1}, {2, 3} }
5:  { {1}, {2}, {3} }
```

The following sets are not set partitions of S_3:

```
  { {1, 2, 3}, {1} }   // intersection not empty
  { {1}, {3} }         // union does not contain 2
```

As the order of elements in a set does not matter we sort them in ascending order. For a set of sets we order the sets in ascending order of the first elements.

We write Z_n for the list of all set partitions of the n-element set S_n. In order to generate Z_n we observe that with a complete list Z_{n-1} of partitions of the set S_{n-1} we can generate the elements of Z_n in the following way: For each element (set partition) $P \in Z_{n-1}$, create set partitions of S_n by appending the element n to the first, second, ..., last subset and one more by appending the set n as the last subset.

For example, the partition $\{\{1, 2\}, \{3, 4\}\} \in Z_4$ leads to 3 partitions of S_5:

```
P = { {1, 2}, {3, 4} }
      -->  { {1, 2, 5}, {3, 4} }
      -->  { {1, 2}, {3, 4, 5} }
      -->  { {1, 2}, {3, 4}, {5} }
```

```
 ------------------
p1={1}
      -->  p={1, 2}
      -->  p={1}, {2}
 ------------------
p1={1, 2}
      -->  p={1, 2, 3}
      -->  p={1, 2}, {3}
p1={1}, {2}
      -->  p={1, 3}, {2}
      -->  p={1}, {2, 3}
      -->  p={1}, {2}, {3}
 ------------------
p1={1, 2, 3}
      -->  p={1, 2, 3, 4}
      -->  p={1, 2, 3}, {4}
p1={1, 2}, {3}
      -->  p={1, 2, 4}, {3}
      -->  p={1, 2}, {3, 4}
      -->  p={1, 2}, {3}, {4}
p1={1, 3}, {2}
      -->  p={1, 3, 4}, {2}
      -->  p={1, 3}, {2, 4}
      -->  p={1, 3}, {2}, {4}
p1={1}, {2, 3}
      -->  p={1, 4}, {2, 3}
      -->  p={1}, {2, 3, 4}
      -->  p={1}, {2, 3}, {4}
p1={1}, {2}, {3}
      -->  p={1, 4}, {2}, {3}
      -->  p={1}, {2, 4}, {3}
      -->  p={1}, {2}, {3, 4}
      -->  p={1}, {2}, {3}, {4}
 ------------------
```

```
 1:    {1, 2, 3, 4}
 2:    {1, 2, 3}, {4}
 3:    {1, 2, 4}, {3}
 4:    {1, 2}, {3, 4}
 5:    {1, 2}, {3}, {4}
 6:    {1, 3, 4}, {2}
 7:    {1, 3}, {2, 4}
 8:    {1, 3}, {2}, {4}
 9:    {1, 4}, {2, 3}
10:    {1}, {2, 3, 4}
11:    {1}, {2, 3}, {4}
12:    {1, 4}, {2}, {3}
13:    {1}, {2, 4}, {3}
14:    {1}, {2}, {3, 4}
15:    {1}, {2}, {3}, {4}
```

Figure 5.11: Recursive construction of the set partitions of the 4-element set $S_4 = \{1, 2, 3, 4\}$ (left). The resulting list of all set partitions of is shown on the right.

```
   n   b=   B(n)    k: 1     2     3     4     5     6     7     8     9    10
  1:  b=      1        1
  2:  b=      2        1     1
  3:  b=      5        1     3     1
  4:  b=     15        1     7     6     1
  5:  b=     52        1    15    25    10     1
  6:  b=    203        1    31    90    65    15     1
  7:  b=    877        1    63   301   350   140    21     1
  8:  b=   4140        1   127   966  1701  1050   266    28     1
  9:  b=  21147        1   255  3025  7770  6951  2646   462    36     1
 10:  b= 115975        1   511  9330 34105 42525 22827  5880   750    45     1
```

Figure 5.12: Stirling numbers of the second kind and Bell numbers.

Now we start with the only partition of the one-element set, $\{\{1\}\}$, and apply the described step $n-1$ times. The construction is shown in the left column of figure 5.11 on the preceding page, the right column shows all set partitions for $n = 5$.

The sequence of numbers of set partitions of S_n for $n \geq 1$ is 1, 2, 5, 15, 52, 203, 877, ... These are the *Bell numbers*, sequence A000110 of [OEIS]. The Bell numbers can be computed using the observation that in every step of the computation a partition of S_{x-1} into k subsets leads to $k+1$ partitions of S_x. Of these partitions k contain k subsets and one contains $k+1$ subsets.

If we write the number of partitions of S_n into k subsets as a triangular array, then the entry can be computed as the sum of its upper left neighbor plus k times its upper neighbor. We start with a single one in the row for $n = 1$. The scheme is shown in figure 5.12 on the previous page. The numbers are the *Stirling numbers of the second kind* (or *Stirling set numbers*), see sequence A008277 of [OEIS].

The sum over all elements of row n gives the n-th Bell number. Another way of computing the Bell numbers is given in section 3.8 on page 121. The array shown can be generated with the demo [FXT: demo comb/stirling2-demo.cc].

We further have the recursion

$$B_{n+1} \quad = \quad \sum_{k=0}^{n} \binom{n}{k} B_k \tag{5.6}$$

As pari/gp code:

```
? N=11; v=vector(N); v[1]=1;
? for (j=2, N, v[j]=sum(k=1, j-1, binomial(j-2,k-1)*v[k])); v
  [1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975]
```

A exponential generating function for the Bell numbers is

$$\exp(\exp(x)-1) \quad = \quad \sum_{n=0}^{\infty} B_n \frac{x^n}{n!} \tag{5.7}$$

Which can be cast into the one-liner

```
? v=Vec(taylor(exp(exp(x)-1),x));vector(length(v),j,v[j]*(j-1)!)
  [1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597, 27644437, 190899322]
```

```
 1:  {1, 2, 3, 4}              1:  {1}, {2}, {3}, {4}
 2:  {1, 2, 3}, {4}            2:  {1}, {2}, {3, 4}
 3:  {1, 2, 4}, {3}            3:  {1}, {2, 4}, {3}
 4:  {1, 2}, {3, 4}            4:  {1, 4}, {2}, {3}
 5:  {1, 2}, {3}, {4}          5:  {1, 4}, {2, 3}
 6:  {1, 3, 4}, {2}            6:  {1}, {2, 3, 4}
 7:  {1, 3}, {2, 4}            7:  {1}, {2, 3}, {4}
 8:  {1, 3}, {2}, {4}          8:  {1, 3}, {2}, {4}
 9:  {1, 4}, {2, 3}            9:  {1, 3}, {2, 4}
10:  {1}, {2, 3, 4}           10:  {1, 3, 4}, {2}
11:  {1}, {2, 3}, {4}         11:  {1, 2, 3, 4}
12:  {1, 4}, {2}, {3}         12:  {1, 2, 3}, {4}
13:  {1}, {2, 4}, {3}         13:  {1, 2}, {3}, {4}
14:  {1}, {2}, {3, 4}         14:  {1, 2}, {3, 4}
15:  {1}, {2}, {3}, {4}       15:  {1, 2, 4}, {3}
```

Figure 5.13: Set partitions of $S_4 = \{1, 2, 3, 4\}$ in two different minimal-change orders.

A slight modification of the construction allows us to generate the set partitions in a minimal-change order. Firstly, we can generate the 'incremented' partitions in two orders, forward (left to right)

```
P = { {1, 2}, {3, 4} }
      -->  { {1, 2, 5}, {3, 4} }
      -->  { {1, 2}, {3, 4, 5} }
      -->  { {1, 2}, {3, 4}, {5} }
```

or backward (right to left)

```
    P = { {1, 2}, {3, 4} }
          -->  { {1, 2}, {3, 4}, {5} }
          -->  { {1, 2}, {3, 4, 5} }
          -->  { {1, 2, 5}, {3, 4} }
```

With the forward direction we obtain the partitions ordered as shown for $n = 5$ in the right column of figure 5.11 on page 169, the backward direction gives the reversed list. If we, secondly, change the direction with every subset that is to be incremented, we obtain the minimal-change order shown in figure 5.13 on the facing page for $n = 4$. The left column is obtained by starting with the forward direction in each step of the recursion, the right by starting with the backward direction. The list can be computed with [FXT: demo comb/setpartition-demo.cc].

The C++ class [FXT: `class set_partition` in comb/setpartition.h] stores the list in an array of signed characters. The stored value is negated if the element is the last in the subset. The work involved with the creation of Z_n is proportional to $\sum_{k=1}^{n} k\, B_k$ where B_k is the k-th Bell number.

The constructor allows to determine the order in which the partitions are being created:

```
class set_partition
// Set partitions of the set {1,2,3,...,n}
// By default in minimal change order
{
public:
    ulong n_;   // Number of elements of set (set = {1,2,3,...,n})
    int *p_;    // p[] contains set partitions of length 1,2,3,...,n
    int **pp_;  // pp[k] points to start of set partition k
    int *ns_;   // ns[k] Number of Sets in set partition k
    int *as_;   // element k attached At Set (0<=as[k]<=k) of set(k-1)
    int *d_;    // direction with recursion (+1 or -1)
    int *x_;    // current set partition (==pp[n])
    bool xdr_;  // whether to change direction in recursion (==> minimal change order)
    int dr0_;   // dr0: starting direction in each recursive step:
    //   dr0=+1  ==> start with partition  {{1,2,3,...,n}}
    //   dr0=-1  ==> start with partition  {{1},{2},{3},...,{n}}}
public:
    set_partition(ulong n, bool xdr=true, int dr0=+1)
        : n_(n)
    {
        ulong np = (n_*(n_+1))/2;  // == \sum_{k=1}^{n}{k}
        p_ = new int[np];

        pp_ = new int *[n_+1];
        pp_[0] = 0;  // unused
        pp_[1] = p_;
        for (ulong k=2; k<=n_; ++k)  pp_[k] = pp_[k-1] + (k-1);

        ns_ = new int[n_+1];
        as_ = new int[n_+1];
        d_ = new int[n_+1];
        x_ = pp_[n_];

        init(xdr, dr0);
    }
   [-snip-]  // destructor

    bool next()
    { return next_rec(n_); }

    const int* get()  const
    { return x_; }

    ulong print()  const
    // Print current set partition
    // Return number of chars printed
    { return print_p(n_); }

    ulong print_p(ulong k)  const;
    void print_internal()  const;  // print internal state
protected:
   [-snip-]  // internal methods
};
```

The actual work is done by the methods `next_rec()` and `cp_append()` [FXT: file comb/setpartition.cc]:

```
int
set_partition::cp_append(const int *src, int *dst, ulong k, ulong a)
// Copy partition in src[0,...,k-2] to dst[0,...,k-1]
// append element k at subset a (a>=0)
// Return number of sets in created partition.
{
    ulong ct = 0;
    for (ulong j=0; j<k-1; ++j)
    {
        int e = src[j];
        if ( e > 0 )  dst[j] = e;
        else
        {
            if ( a==ct )  { dst[j]=-e; ++dst; dst[j]=-k; }
            else  dst[j] = e;
            ++ct;
        }
    }
    if ( a>=ct )  { dst[k-1] = -k; ++ct; }
    return ct;
}

int
set_partition::next_rec(ulong k)
// Update partition in level k from partition in level k-1  (k<=n)
// Return number of sets in created partition
{
    if ( k<=1 )  return 0;  // current is last

    int d = d_[k];
    int as = as_[k] + d;
    bool ovq = ( (d>0) ? (as>ns_[k-1]) : (as<0) );
    if ( ovq )  // have to recurse
    {
        ulong ns1 = next_rec(k-1);
        if ( 0==ns1 )  return 0;

        d = ( xdr_ ? -d : dr0_ );
        d_[k] = d;

        as = ( (d>0) ? 0 : ns_[k-1] );
    }
    as_[k] = as;

    ulong ns = cp_append(pp_[k-1], pp_[k], k, as);
    ns_[k] = ns;
    return ns;
}
```

The partitions are stored as an array of signed integers that are greater than zero and smaller or equal to n. A negative value indicates that it is the last of the subset. For example, the set partitions of S_4 together with their 'signed value' representations are shown in figure 5.14 on the next page. The array `as[]` contains a restricted growth string (RGS) with the condition $a_j \leq 1 + \max_{i<j}(a_i)$. A different sort of RGSs is described in section 5.5 on page 159.

The copying is the performance bottleneck of the algorithm. Therefore 'only' about 11 million partitions are generated per second.

5.8 Compositions in co-lexicographic order

The compositions of n into at most k parts ('k-compositions of n') are the ordered tuples $(x_0, x_1, \ldots, x_{k-1})$ where $x_0 + x_1 + \ldots + x_{k-1} = n$ and $0 \leq x_i \leq n$. Order matters: one 3-composition of 4 is $(0, 1, 2, 1)$, a different one is $(2, 0, 1, 1)$.

The 3-compositions of 5 in co-lexicographic (colex) order are shown in figure 5.15 on the next page. This is the output of [FXT: demo comb/composition-colex-demo.cc]:

```
int
```

```
arg 1: 4 == n  [Partition set of n elements]
arg 2: 0 == xdr  [Change direction in recursion ==> minimal change order]
arg 3: 1 == dr0  [Starting direction in recursion (+-1)]
arg 4: 0 == maxct  [Stop after maxct partitions (0: never stop)]
arg 5: 1 == priq  [Option: print internal state with each partition]
  1:  as[ , 0, 0, 0, 0, ]     x[ +1, +2, +3, -4, ]    {1, 2, 3, 4}
  2:  as[ , 0, 0, 0, 1, ]     x[ +1, +2, -3, -4, ]    {1, 2, 3}, {4}
  3:  as[ , 0, 0, 1, 0, ]     x[ +1, +2, -4, -3, ]    {1, 2, 4}, {3}
  4:  as[ , 0, 0, 1, 1, ]     x[ +1, -2, +3, -4, ]    {1, 2}, {3, 4}
  5:  as[ , 0, 0, 1, 2, ]     x[ +1, -2, -3, -4, ]    {1, 2}, {3}, {4}
  6:  as[ , 0, 1, 0, 0, ]     x[ +1, +3, -4, -2, ]    {1, 3, 4}, {2}
  7:  as[ , 0, 1, 0, 1, ]     x[ +1, -3, +2, -4, ]    {1, 3}, {2, 4}
  8:  as[ , 0, 1, 0, 2, ]     x[ +1, -3, -2, -4, ]    {1, 3}, {2}, {4}
  9:  as[ , 0, 1, 1, 0, ]     x[ +1, -4, +2, -3, ]    {1, 4}, {2, 3}
 10:  as[ , 0, 1, 1, 1, ]     x[ -1, +2, +3, -4, ]    {1}, {2, 3, 4}
 11:  as[ , 0, 1, 1, 2, ]     x[ -1, +2, -3, -4, ]    {1}, {2, 3}, {4}
 12:  as[ , 0, 1, 2, 0, ]     x[ +1, -4, -2, -3, ]    {1, 4}, {2}, {3}
 13:  as[ , 0, 1, 2, 1, ]     x[ -1, +2, -4, -3, ]    {1}, {2, 4}, {3}
 14:  as[ , 0, 1, 2, 2, ]     x[ -1, -2, +3, -4, ]    {1}, {2}, {3, 4}
 15:  as[ , 0, 1, 2, 3, ]     x[ -1, -2, -3, -4, ]    {1}, {2}, {3}, {4}
```

Figure 5.14: The partitions of the set $S_4 = \{1, 2, 3, 4\}$ together with the internal representations: the 'signed value' array `x[]` and the 'attachment' array `as[]`.

```
#  0:   3 . . . .
#  1:   2 1 . . .
#  2:   1 2 . . .
#  3:   . 3 . . .
#  4:   2 . 1 . .
#  5:   1 1 1 . .
#  6:   . 2 1 . .
#  7:   1 . 2 . .
#  8:   . 1 2 . .
#  9:   . . 3 . .
# 10:   2 . . 1 .
# 11:   1 1 . 1 .
# 12:   . 2 . 1 .
# 13:   1 . 1 1 .
# 14:   . 1 1 1 .
# 15:   . . 2 1 .
# 16:   1 . . 2 .
# 17:   . 1 . 2 .
# 18:   . . 1 2 .
# 19:   . . . 3 .
# 20:   2 . . . 1
# 21:   1 1 . . 1
# 22:   . 2 . . 1
# 23:   1 . 1 . 1
# 24:   . 1 1 . 1
# 25:   . . 2 . 1
# 26:   1 . . 1 1
# 27:   . 1 . 1 1
# 28:   . . 1 1 1
# 29:   . . . 2 1
# 30:   1 . . . 2
# 31:   . 1 . . 2
# 32:   . . 1 . 2
# 33:   . . . 1 2
# 34:   . . . . 3
```

Figure 5.15: The 3-compositions of 5 in Co-lexicographic order. Dots are used for zeros.

```
main(int argc, char **argv)
{
    ulong k = 5, n = 3;
    composition_colex p(n, k);
    const ulong *x = p.data();
    ulong ct = 0;
    do
    {
        cout << " #" << setw(3) << ct << ":   ";
        for (ulong i=0; i<k; ++i)  cout << x[i] << " ";
        cout << endl;
        ++ct;
    }
    while ( p.next() );
    return 0;
}
```

The C++ class used is [FXT: `class composition_colex` in comb/composition-colex.h]:

```
class composition_colex
// k-compositions of n, co-lexicographic (colex) order
{
public:
    ulong n_, k_;  // n is the sum of k elements
    ulong *x_;  // data
public:
    composition_colex(ulong n, ulong k)
    {
        n_ = (n ? n : 1);  // not zero
        k_ = (k ? k : 1);  // not zero
        x_ = new ulong[k_];
        first();
    }

    ~composition_colex()  { delete [] x_; }

    const ulong *data()  const  { return x_; }

    void first()
    {
        x_[0] = n_;
        for (ulong k=1; k<k_; ++k)  x_[k] = 0;
    }

    void last()
    {
        for (ulong k=0; k<k_; ++k)  x_[k] = 0;
        x_[k_-1] = n_;
    }
```

The methods `next()` and `prev()` can be implemented concisely as:

```
    ulong next()
    {
        // Return zero if current comp. is last:
        if ( n_==x_[k_-1] )  return 0;

        ulong j = 0;
        while ( 0==x_[j] )  ++j;  // find first nonzero
        ulong v = x_[j];  // value of first nonzero
        x_[j] = 0;        // set to zero
        x_[0] = v - 1;    // value-1 to first position
        ++x_[j+1];        // increment next position

        return  1;
    }

    ulong prev()
    {
        // Return zero if current comp. is first:
        if ( n_==x_[0] )  return 0;

        ulong v0 = x_[0];   // vlaue at first position
        x_[0] = 0;          // set first position to zero
        ulong j = 1;
        while ( 0==x_[j] )  ++j;  // find next nonzero
        --x_[j];                  // decrement value
        x_[j-1] = 1 + v0;         // set previous position
```

```
        return  1;
    }
};
```

More than 100 million combinations are generated per second.

5.9 Compositions in minimal-change order

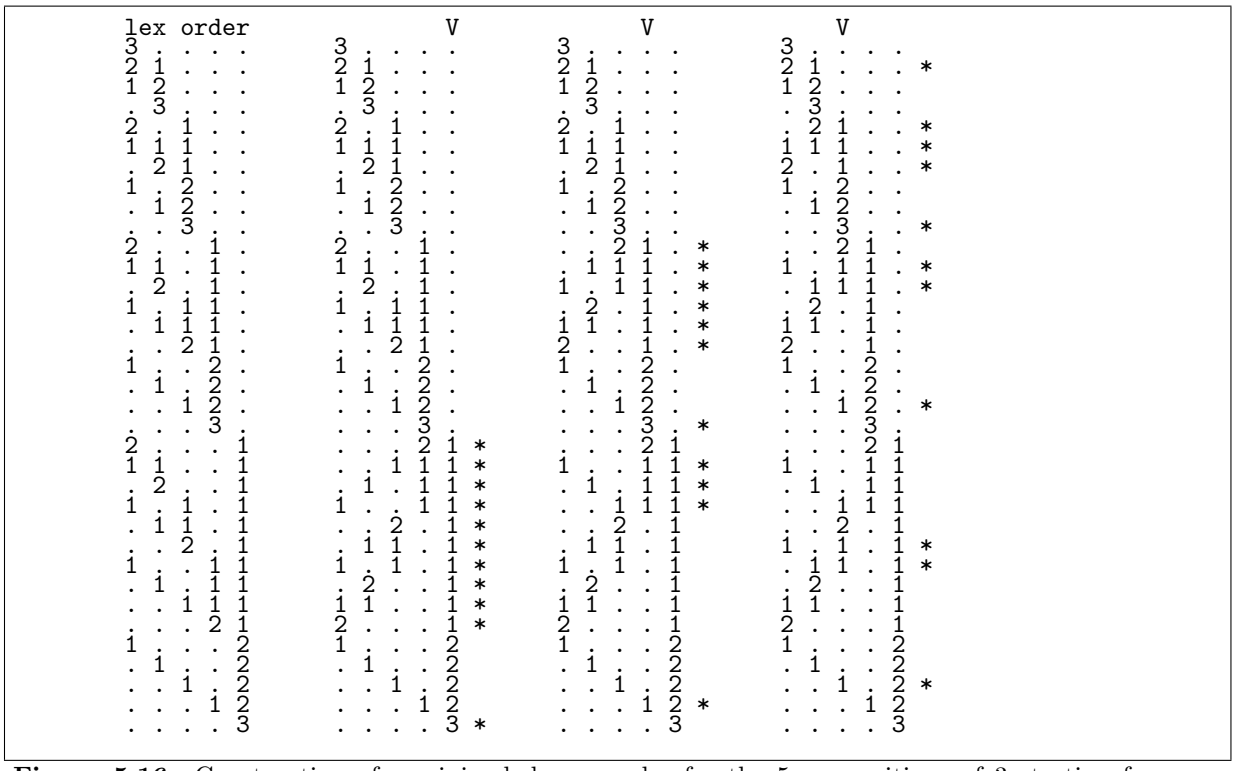


Figure 5.16: Construction of a minimal-change order for the 5-compositions of 3 starting from co-lexicographic order. At each stage the regions of the marked (symbol '`V`') colum that contain an odd entry (symbol '`*`') are reversed. One starts with the rightmost column and works to the left. The two leftmost columns can be omitted as a run of two or more odd numbers never occurs

A recursive construction for a minimal-change order for compositions is shown in figure 5.16. The algorithm starts with a list in co-lexicographic order and works on the complete list. Hence it is not efficient. It is implemented as [FXT: `class all_compositions_minchange` in comb/all-compositions-minchange.h].

In the list in minimal-change oder (rightmost column in figure 5.16) each transition either changes two adjacent elements by one or (if k odd) the lowest nonzero set is decremented and the lowest element is set to one or (if k even) the lowest element (which equals one) is set to zero and the second lowest nonzero element is incremented.

Figure 5.17 on the following page shows a minimal-change order of 4-compositions of 4 which is *near-perfect*: a one in the delta set moves by at most two positions. The list was generated using [FXT: `class composition_chase` in comb/composition-chase.h]. The class is based on a generator for a near-perfect order for combinations that is given in [DEK4]. The list shown in figure 5.17 on the next page is the output of [FXT: demo comb/composition-chase-demo.cc].

```
 0:    3 4 5 6     . . . 1 1 1 1     0 0 0 4
 1:    2 4 5 6     . . 1 . 1 1 1     0 0 1 3
 2:    0 4 5 6     1 . . . 1 1 1     1 0 0 3
 3:    1 4 5 6     . 1 . . 1 1 1     0 1 0 3
 4:    1 2 5 6     . 1 1 . . 1 1     0 2 0 2
 5:    0 2 5 6     1 . 1 . . 1 1     1 1 0 2
 6:    0 1 5 6     1 1 . . . 1 1     2 0 0 2
 7:    0 3 5 6     1 . . 1 . 1 1     1 0 1 2
 8:    1 3 5 6     . 1 . 1 . 1 1     0 1 1 2
 9:    2 3 5 6     . . 1 1 . 1 1     0 0 2 2
10:    2 3 4 6     . . 1 1 1 . 1     0 0 3 1
11:    0 3 4 6     1 . . 1 1 . 1     1 0 2 1
12:    1 3 4 6     . 1 . 1 1 . 1     0 1 2 1
13:    1 2 4 6     . 1 1 . 1 . 1     0 2 1 1
14:    0 2 4 6     1 . 1 . 1 . 1     1 1 1 1
15:    0 1 4 6     1 1 . . 1 . 1     2 0 1 1
16:    0 1 2 6     1 1 1 . . . 1     3 0 0 1
17:    0 1 3 6     1 1 . 1 . . 1     2 1 0 1
18:    0 2 3 6     1 . 1 1 . . 1     1 2 0 1
19:    1 2 3 6     . 1 1 1 . . 1     0 3 0 1
20:    1 2 3 4     . 1 1 1 1 . .     0 4 0 0
21:    0 2 3 4     1 . 1 1 1 . .     1 3 0 0
22:    0 1 3 4     1 1 . 1 1 . .     2 2 0 0
23:    0 1 2 4     1 1 1 . 1 . .     3 1 0 0
24:    0 1 2 3     1 1 1 1 . . .     4 0 0 0
25:    0 1 2 5     1 1 1 . . 1 .     3 0 1 0
26:    0 1 3 5     1 1 . 1 . 1 .     2 1 1 0
27:    0 2 3 5     1 . 1 1 . 1 .     1 2 1 0
28:    1 2 3 5     . 1 1 1 . 1 .     0 3 1 0
29:    1 2 4 5     . 1 1 . 1 1 .     0 2 2 0
30:    0 2 4 5     1 . 1 . 1 1 .     1 1 2 0
31:    0 1 4 5     1 1 . . 1 1 .     2 0 2 0
32:    0 3 4 5     1 . . 1 1 1 .     1 0 3 0
33:    1 3 4 5     . 1 . 1 1 1 .     0 1 3 0
34:    2 3 4 5     . . 1 1 1 1 .     0 0 4 0
```

Figure 5.17: 4-compositions of 4 in a minimal-change order. The left column shows the corresponding combinations.

5.10 Compositions and combinations

Figure 5.18 on the facing page shows the correspondence between compositions and combinations. The listing was generated using [FXT: demo comb/comb2comp-demo.cc]. Entries in the left column are combinations of 5 parts out of 7. The middle column is a binary representation of the combination in the obvious way. It also is a binary representation of a composition: A run of r consecutive ones corresponds to an r in the 3-composition of 5 in the right column.

As can be seen, a k-composition of n corresponds to a combination of $K = n$ parts from $N = n + k - 1$ elements and a combination of K elements out of N corresponds to a k-composition of n where $k = N - K + 1$ and $n = K$.

Routines for the conversion between combinations and compositions are given in [FXT: file comb/comp2comb.h]. Conversion of a composition into the corresponding combination:

```
inline void
comp2comb(const ulong *c, ulong k, ulong *m)
// Convert k-composition in c[] to combination in m[]
{
    for (ulong j=0,i=0,z=0; j<k; ++j)
    {
        ulong cj = c[j];
        for (ulong w=0; w<cj; ++w)   m[i++] = z++;
        ++z;
    }
}
```

Conversion of a combination into the corresponding composition:

```
inline void
comb2comp(const ulong *m, ulong N, ulong K, ulong *c)
// Convert combination (N choose K) in m[] to k-composition in c[]
// Must have: K>0
{
    ulong k = N-K+1;
```

```
0 1 2 3 4    11111..    5 0 0
0 1 2 4 5    111.11.    3 2 0
0 2 3 4 5    1.1111.    1 4 0
1 2 3 4 5    .11111.    0 5 0
0 1 3 4 5    11.111.    2 3 0
0 1 2 3 5    1111.1.    4 1 0
0 1 2 5 6    111..11    3 0 2
0 2 3 5 6    1.11.11    1 2 2
1 2 3 5 6    .111.11    0 3 2
0 1 3 5 6    11.1.11    2 1 2
0 3 4 5 6    1..1111    1 0 4
1 3 4 5 6    .1.1111    0 1 4
2 3 4 5 6    ..11111    0 0 5
0 2 4 5 6    1.1.111    1 1 3
1 2 4 5 6    .11.111    0 2 3
0 1 4 5 6    11..111    2 0 3
0 1 2 4 6    111.1.1    3 1 1
0 2 3 4 6    1.111.1    1 3 1
1 2 3 4 6    .1111.1    0 4 1
0 1 3 4 6    11.11.1    2 2 1
0 1 2 3 6    1111..1    4 0 1
```

Figure 5.18: Combinations 5 out of seven (left) and the corresponding 3-compositions of 5 (right). Note that while the sequence of combinations has a minimal-change property the corresponding sequence of compositions does not.

```
    for (ulong z=0; z<k; ++z)  c[z] = 0;
    --k;
    ulong m1 = N;
    while ( K-- )
    {
        ulong m0 = m[K];
        ulong d = m1 - m0;
        k -= (d-1);
        ++c[k];
        m1 = m0;
    }
}
```

5.11 Subsets in lexicographic order

The (nonempty) subsets of a set of five elements in *lexicographic* order shown in figure 5.19 on the next page. Clearly there are 2^n subsets (including the empty set) of an n-element set.

The corresponding utility class is not too complicated [FXT: `class subset_lex` in comb/subset-lex.h]:

```
class subset_lex
{
protected:
    ulong *x;  // subset data
    ulong n;   // number of elements in set
    ulong k;   // index of last element in subset
    // Number of elements in subset == k+1

    [-snip-]
```

The crucial methods are `next()` and `prev()`:

```
    ulong next()
    // Generate next subset
    // Return number of elements in subset
    // Return zero if current == last
    {
        if ( x[k] == n-1 )  // last element is max ?
        {
            if ( 0==k )  { return 0; }  // note: user has to call first() again

            --k;     // remove last element
            x[k]++;  // increase last element
        }
        else  // add next element from set:
        {
```

```
 0  #= 1:    ....1  {0}
 1  #= 2:    ...11  {0, 1}
 2  #= 3:    ..111  {0, 1, 2}
 3  #= 4:    .1111  {0, 1, 2, 3}
 4  #= 5:    11111  {0, 1, 2, 3, 4}
 5  #= 4:    1.111  {0, 1, 2, 4}
 6  #= 3:    .1.11  {0, 1, 3}
 7  #= 4:    11.11  {0, 1, 3, 4}
 8  #= 3:    1..11  {0, 1, 4}
 9  #= 2:    ..1.1  {0, 2}
10  #= 3:    .11.1  {0, 2, 3}
11  #= 4:    111.1  {0, 2, 3, 4}
12  #= 3:    1.1.1  {0, 2, 4}
13  #= 2:    .1..1  {0, 3}
14  #= 3:    11..1  {0, 3, 4}
15  #= 2:    1...1  {0, 4}
16  #= 1:    ...1.  {1}
17  #= 2:    ..11.  {1, 2}
18  #= 3:    .111.  {1, 2, 3}
19  #= 4:    1111.  {1, 2, 3, 4}
20  #= 3:    1.11.  {1, 2, 4}
21  #= 2:    .1.1.  {1, 3}
22  #= 3:    11.1.  {1, 3, 4}
23  #= 2:    1..1.  {1, 4}
24  #= 1:    ..1..  {2}
25  #= 2:    .11..  {2, 3}
26  #= 3:    111..  {2, 3, 4}
27  #= 2:    1.1..  {2, 4}
28  #= 1:    .1...  {3}
29  #= 2:    11...  {3, 4}
30  #= 1:    1....  {4}
```

Figure 5.19: Subsets of a five element set in lexicographic order.

```
            ++k;
            x[k] = x[k-1] + 1;
        }
        return  k + 1;
    }
    ulong prev()
    // Generate previous subset
    // Return number of elements in subset
    // Return zero if current == first
    {
        if ( k == 0 )  // only one lement ?
        {
            if ( x[0]==0 )  { return 0; }  // note: user has to call last() again
            x[0]--;  // decr first element
            x[++k] = n - 1;     // add element
        }
        else  // remove last element:
        {
            if ( x[k] == x[k-1]+1 )  --k;
            else
            {
                x[k]--;  // decr last element
                x[++k] = n - 1;     // add element
            }
        }
        return  k + 1;
    }
    const ulong * data()  { return x; }
};
```

One can generate the list at the beginning of this sections by a code fragment like [FXT: demo comb/subset-lex-demo.cc]:

```
    ulong n = 5;
```

```
subset_lex sl(n);
ulong idx = 0;
ulong num = sl.first();
do
{
    cout << setw(2) << idx;
    ++idx;

    cout << "  #=" << setw(2) << num << ":  ";
    print_set_as_bitset("  ", sl.data(), num, n);
    print_set("  ", sl.data(), num);
    cout << endl;
}
while ( (num = sl.next()) );
```

More than 230 million subsets per second can be generated with the given code.

5.12 Subsets in minimal-change order

```
 0:  .....   chg @ 4   #=0   set={}
 1:  1....   chg @ 0   #=1   set={0}
 2:  11...   chg @ 1   #=2   set={0, 1}
 3:  .1...   chg @ 0   #=1   set={1}
 4:  .11..   chg @ 2   #=2   set={1, 2}
 5:  111..   chg @ 0   #=3   set={0, 1, 2}
 6:  1.1..   chg @ 1   #=2   set={0, 2}
 7:  ..1..   chg @ 0   #=1   set={2}
 8:  ..11.   chg @ 3   #=2   set={2, 3}
 9:  1.11.   chg @ 0   #=3   set={0, 2, 3}
10:  1111.   chg @ 1   #=4   set={0, 1, 2, 3}
11:  .111.   chg @ 0   #=3   set={1, 2, 3}
12:  .1.1.   chg @ 2   #=2   set={1, 3}
13:  11.1.   chg @ 0   #=3   set={0, 1, 3}
14:  1..1.   chg @ 1   #=2   set={0, 3}
15:  ...1.   chg @ 0   #=1   set={3}
16:  ...11   chg @ 4   #=2   set={3, 4}
17:  1..11   chg @ 0   #=3   set={0, 3, 4}
18:  11.11   chg @ 1   #=4   set={0, 1, 3, 4}
19:  .1.11   chg @ 0   #=3   set={1, 3, 4}
20:  .1111   chg @ 2   #=4   set={1, 2, 3, 4}
21:  11111   chg @ 0   #=5   set={0, 1, 2, 3, 4}
22:  1.111   chg @ 1   #=4   set={0, 2, 3, 4}
23:  ..111   chg @ 0   #=3   set={2, 3, 4}
24:  ..1.1   chg @ 3   #=2   set={2, 4}
25:  1.1.1   chg @ 0   #=3   set={0, 2, 4}
26:  111.1   chg @ 1   #=4   set={0, 1, 2, 4}
27:  .11.1   chg @ 0   #=3   set={1, 2, 4}
28:  .1..1   chg @ 2   #=2   set={1, 4}
29:  11..1   chg @ 0   #=3   set={0, 1, 4}
30:  1...1   chg @ 1   #=2   set={0, 4}
31:  ....1   chg @ 0   #=1   set={4}
```

Figure 5.20: The subsets of the set {0, 1, 2, 3, 4} in minimal-change order.

The subsets of a set with 5 elements in minimal-change order are shown in figure 5.20, it is the output of [FXT: demo comb/subset-gray-delta-demo.cc]. The implementation [FXT: `class subset_gray_delta` in comb/subset-gray-delta.h] uses the Gray code of binary words and updates the position corresponding to the bit that changes in the Gray code:

```
class subset_gray_delta
// Subsets of the set {0,1,2,...,n-1} in minimal-change (Gray code) order.
// O(1) algorithm.
{
public:
    ulong *x_;  // current subset as delta-set
    ulong n_;   // number of elements in set <= BITS_PER_LONG
    ulong k_;   // number of elements in current subset
```

```
    ulong ch_;  // element that was changed with latest call to next()
    ulong ct_;  // gray_code(ct_) corresponds to the current subset
    ulong mct_; // max value of ct.
public:
    subset_gray_delta(ulong nn)
    {
        n_ = (nn ? nn : 1);  // not zero
        x_ = new ulong[n_];
        mct_ = (1UL<<nn) - 1;
        empty();
    }
    ~subset_gray_delta()  { delete [] x_; }

    ulong empty()
    {
        ct_ = 0;
        k_ = 0;
        ch_ = n_ - 1;
        for (ulong j=0; j<n_; ++j)  x_[j] = 0;
        return  k_;
    }
    const ulong * data() const { return x_; }
    ulong current()  const  { return ct_; }
    ulong num()  const  { return k_; }
    ulong next()
    {
        if ( ct_ == mct_ )
        {
            ct_ = 0;
            x_[n_-1] = 0;
            ch_ = n_ - 1;
            return 0;
        }
        ++ct_;
        ch_ = lowest_bit_idx( ct_ );
        x_[ch_] = 1 - x_[ch_];
        k_ += (x_[ch_] ? 1 : -1);
        return  k_;
    }
    ulong prev()
    {
        if ( ct_ == 0 )
        {
            ct_ = mct_;
            k_ = 1;
            x_[n_-1] = 1;
            ch_ = n_ - 1;
            return 1;
        }
        ch_ = lowest_bit_idx( ct_ );
        x_[ch_] = 1 - x_[ch_];
        k_ += (x_[ch_] ? +1 : -1);
        --ct_;
        return  k_;
    }
}
```

About 110 million subsets are generated per second.

The class [FXT: `class subset_gray` in comb/subset-gray.h] generates the subsets of $\{1, 2, \ldots, n\}$ as sets. The underlying idea is described in section 1.16.3 on page 35. The demo [FXT: demo comb/subset-gray-demo.cc] shows how to use the class.

```
class subset_gray
// Subsets of the set {1,2,...,n} in minimal-change (Gray code) order.
// O(1) algorithm following Jenkyns ("Loopless Gray Code Algorithms").
{
public:
    ulong *x_;  // data k-subset of {1,2,...,n} in x[1,...,k]
    ulong n_;   // subsets of n-set
    ulong k_;   // number of elements in subset
```

```
public:
    subset_gray(ulong n)
        : n_(n)
    {
        x_ = new ulong[n_+1];
        x_[0] = 0;
        empty();
    }

    ~subset_gray()  { delete [] x_; }

    ulong empty()  { k_ = 0;  return k_; }
    ulong first()  { x_[1] = n_;  k_ = 1;  return k_; }
    ulong last()  { x_[1] = 1;  k_ = 1;  return k_; }

    const ulong * data() const { return x_+1; }
    const ulong num() const { return k_; }

    ulong next_even()
    { // either remove or add n to the (end of the) set
        if ( x_[k_]==n_ )   --k_;
        else  { ++k_; x_[k_] = n_; }
        return  k_;
    }

    ulong next_odd()
    {  // either remove x[k]-1 (from position k-1) or insert it as second last element
        if ( x_[k_]-1==x_[k_-1] )   { x_[k_-1] = x_[k_];  --k_; }
        else  { x_[k_+1] = x_[k_];  --x_[k_];  ++k_; }
        return  k_;
    }

    ulong next()
    {
        if ( 0==(k_&1 ) ) return next_even();
        else              return next_odd();
    }

    ulong prev()
    {
        if ( 0==(k_&1 ) )  // k even
        {
            if ( 0==k_ )  return last();
            return next_odd();
        }
        else  return next_even();
    }
};
```

More than 160 million subsets are generated per second.

5.13 Subsets ordered by number of elements

Sometimes it is useful to generate all subsets ordered with respect to the number of elements, that is starting with the 1-element subsets, continuing with 2-element subsets and so on until the full set is reached. For that purpose one needs to generate the combinations of 1 from n, 2 from n and so on. There are of course many orderings of that type, practical choices are limited by the various generators for combinations one wants to use. When using the colex-order for the combinations one obtains what is shown in figure 5.21 on the following page.

The class implementing the obvious algorithm is [FXT: `class subset_monotone` in comb/subset-monotone.h]. The above list can be generated via [FXT: demo comb/subset-monotone-demo.cc]:

```
    ulong n = 5;
    subset_monotone so(n);
    const ulong *x = so.data();
    ulong num, idx = 0;
    do
    {
        num = so.next();
        ++idx;
        cout << setw(2) << idx << ":  ";
```

```
 1:  1....   #=1   set={0}
 2:  .1...   #=1   set={1}
 3:  ..1..   #=1   set={2}
 4:  ...1.   #=1   set={3}
 5:  ....1   #=1   set={4}
 6:  11...   #=2   set={0, 1}
 7:  1.1..   #=2   set={0, 2}
 8:  .11..   #=2   set={1, 2}
 9:  1..1.   #=2   set={0, 3}
10:  .1.1.   #=2   set={1, 3}
11:  ..11.   #=2   set={2, 3}
12:  1...1   #=2   set={0, 4}
13:  .1..1   #=2   set={1, 4}
14:  ..1.1   #=2   set={2, 4}
15:  ...11   #=2   set={3, 4}
16:  111..   #=3   set={0, 1, 2}
17:  11.1.   #=3   set={0, 1, 3}
18:  1.11.   #=3   set={0, 2, 3}
19:  .111.   #=3   set={1, 2, 3}
20:  11..1   #=3   set={0, 1, 4}
21:  1.1.1   #=3   set={0, 2, 4}
22:  .11.1   #=3   set={1, 2, 4}
23:  1..11   #=3   set={0, 3, 4}
24:  .1.11   #=3   set={1, 3, 4}
25:  ..111   #=3   set={2, 3, 4}
26:  1111.   #=4   set={0, 1, 2, 3}
27:  111.1   #=4   set={0, 1, 2, 4}
28:  11.11   #=4   set={0, 1, 3, 4}
29:  1.111   #=4   set={0, 2, 3, 4}
30:  .1111   #=4   set={1, 2, 3, 4}
31:  11111   #=5   set={0, 1, 2, 3, 4}
32:  .....   #=0   set={}
```

Figure 5.21: Subsets of a five element set in a monotone order.

```
        // print as bit set:
        for (ulong k=0; k<n; ++k)  cout << (x[k]?'1':'.');
        cout << "   #=" << num;

        // print as set:
        print_delta_set_as_set("   set=", x, n);
        cout << endl;
    }
    while ( num );
```

Replacing the colex-comb engine by a minimal-change engine (see section 5.3 on page 156) gives the additional feature of minimal changes between the subsets.

5.14 Subsets ordered with De Bruijn sequences

A curious sequence of all subsets of a given set can be generated using a binary *De Bruijn sequence* that is a cyclical sequence of zeros and ones that contains each n-bit word once. In figure 5.22 on the facing page the empty places of the subsets are included to make the nice property apparent. The underlying De Bruijn sequence (DBS) is

00000100011001010011101011011111

(rotated left in the example so that the empty set appears at the end). Each subset is made from its predecessor by shifting it to the right and inserting the current element from the DBS.

The utility class [FXT: `class subset_debruijn` in comb/subset-debruijn.h] uses [FXT: `class debruijn` in comb/debruijn.h] (which in turn uses [FXT: `class prime_string` in comb/primestring.h]). An algorithm for the generation of binary DBSs is given in section 6.1 on page 217.

Figure 5.22 was created via [FXT: demo comb/subset-debruijn-demo.cc]:

```
ulong n = 5;
subset_debruijn sdb(n);
```

```
 0:    {0,  ,  ,  ,  }   #=1    {0}
 1:    { , 1,  ,  ,  }   #=1    {1}
 2:    { ,  , 2,  ,  }   #=1    {2}
 3:    { ,  ,  , 3,  }   #=1    {3}
 4:    {0,  ,  ,  , 4}   #=2    {0, 4}
 5:    {0, 1,  ,  ,  }   #=2    {0, 1}
 6:    { , 1, 2,  ,  }   #=2    {1, 2}
 7:    { ,  , 2, 3,  }   #=2    {2, 3}
 8:    {0,  ,  , 3, 4}   #=3    {0, 3, 4}
 9:    { , 1,  ,  , 4}   #=2    {1, 4}
10:    {0,  , 2,  ,  }   #=2    {0, 2}
11:    { , 1,  , 3,  }   #=2    {1, 3}
12:    { ,  , 2,  , 4}   #=2    {2, 4}
13:    {0,  ,  , 3,  }   #=2    {0, 3}
14:    {0, 1,  ,  , 4}   #=3    {0, 1, 4}
15:    {0, 1, 2,  ,  }   #=3    {0, 1, 2}
16:    { , 1, 2, 3,  }   #=3    {1, 2, 3}
17:    {0,  , 2, 3, 4}   #=4    {0, 2, 3, 4}
18:    { , 1,  , 3, 4}   #=3    {1, 3, 4}
19:    {0,  , 2,  , 4}   #=3    {0, 2, 4}
20:    {0, 1,  , 3,  }   #=3    {0, 1, 3}
21:    { , 1, 2,  , 4}   #=3    {1, 2, 4}
22:    {0,  , 2, 3,  }   #=3    {0, 2, 3}
23:    {0, 1,  , 3, 4}   #=4    {0, 1, 3, 4}
24:    {0, 1, 2,  , 4}   #=4    {0, 1, 2, 4}
25:    {0, 1, 2, 3,  }   #=4    {0, 1, 2, 3}
26:    {0, 1, 2, 3, 4}   #=5    {0, 1, 2, 3, 4}
27:    { , 1, 2, 3, 4}   #=4    {1, 2, 3, 4}
28:    { ,  , 2, 3, 4}   #=3    {2, 3, 4}
29:    { ,  ,  , 3, 4}   #=2    {3, 4}
30:    { ,  ,  ,  , 4}   #=1    {4}
31:    { ,  ,  ,  ,  }   #=0    {}
```

Figure 5.22: Subsets of a five element set in an order corresponding to a De Bruijn sequence.

```
for (ulong j=0; j<=n; ++j)  sdb.next();  // cosmetics: end with empty set
ulong ct = 0;
do
{
    cout << "    " << setw(2) << ct << ":";
    ulong num = print_delta_set_as_set("    ", sdb.data(), n, 1);
    cout << "   #=" << num;
    print_delta_set_as_set("    ", sdb.data(), n);
    cout << endl;
    sdb.next();
}
while ( ++ct < (1UL<<n) );
```

5.15 k-subsets where k lies in a given range

A recursive routine that generates all k-subsets where lies in a prescribed range is [FXT: `class ksubset_rec` in comb/ksubset-rec.h]. The constructor has just one argument, the number of elements of the (super-)set:

```
class ksubset_rec
// k-susbsets where kmin<=k<=kmax in various orders.
// Recursive CAT algorithm.
{
public:
    long n_; // subsets of a n-element set
    long kmin_, kmax_;  //  k-subsets where kmin<=k<=kma
    long *rv_;  // record of visits in graph (list of elements in subset)
    ulong ct_;   // count subsets
    ulong rct_;  // count recursions (==work)
    ulong rq_;   // condition that determines the order
    ulong pq_;   // condition that determines the (printing) order
    ulong nq_;   // whether to reverse order
    // function to call with each combination:
```

```
 0:   1...1   .....    11....   ......   1...1   ......
 1:   1...11  ....P.   111...   ..P...   1...1.  ....PM
 2:   1...1.  ....M    11.1..   ..MP..   1...11  ....P
 3:   1..1..  ...PM.   11..1.   ...MP.   1..11.  ...P.M
 4:   1..11.  ....P.   11...1   ....MP   1..1.1  ....MP
 5:   1..1.1  ....MP   1.1..1   .MP...   1..1..  .....M
 6:   1.1..1  ..PM..   1.1.1.   ....PM   1.1...  ..PM..
 7:   1.1.1.  ....PM   1.11..   ...PM.   1.1..1  .....P
 8:   1.11..  ...PM.   1.1...   ...M..   1.1.1.  ....PM
 9:   1.1...  ...M..   1..1..   ..MP..   1.11..  ...PM.
10:   11....  .PM...   1..11.   ....P.   111...  .P.M..
11:   111...  ..P...   1..1.1   ....MP   11.1..  ..MP..
12:   11.1..  ..MP..   1...11   ...MP.   11..1.  ...MP.
13:   11..1.  ...MP.   1...1.   .....M   11...1  ....MP
14:   11...1  ....MP   1....1   ....MP   11....  .....M
15:   .11..1  M.P...   .1...1   MP....   .11...  M.P...
16:   .11.1.  ...PM    .1..11   ....P.   .11..1  .....P
17:   .111..  ...PM.   .1..1.   .....M   .11.1.  ....PM
18:   .11...  ...M..   .1.1..   ...PM.   .111..  ...PM.
19:   .1.1..  ..MP..   .1.11.   ....P.   .1.11.  ..M.P.
20:   .1.11.  ....P.   .1.1.1   ....MP   .1.1.1  ....MP
21:   .1.1.1  ....MP   .11..1   ..PM..   .1.1..  .....M
22:   .1..11  ...MP.   .11.1.   ....PM   .1..1.  ...MP.
23:   .1..1.  .....M   .111..   ...PM.   .1..11  .....P
24:   .1...1  ....MP   .11...   ...M..   .1...1  ....M.
25:   ..1..1  .MP...   ..11..   .M.P..   ..1..1  .MP...
26:   ..1.11  ....P.   ..111.   ....P.   ..1.1.  ....PM
27:   ..1.1.  .....M   ..11.1   ....MP   ..1.11  .....P
28:   ..11..  ...PM.   ..1.11   ...MP.   ..111.  ...P.M
29:   ..111.  ....P.   ..1.1.   .....M   ..11.1  ....MP
30:   ..11.1  ....MP   ..1..1   ....MP   ..11..  .....M
31:   ...111  ..M.P.   ...1.1   ..MP..   ...11.  ..M.P.
32:   ...11.  .....M   ...111   ....P.   ...111  .....P
33:   ...1.1  ....MP   ...11.   .....M   ...1.1  ....M.
34:   ....11  ...MP.   ....11   ...M.P   ....11  ...MP.
```

Figure 5.23: Three orders of the k-subsets (where $2 \leq k \leq 3$) of a six-element set.

```
 0:   ......   ......                    32:   1....1   ....MP    0 5
 1:   1.....   P.....    0                33:   .1...1   MP....    1 5
 2:   11....   .P....    0 1              34:   .1..11   ....P.    1 4 5
 3:   111...   ..P...    0 1 2            35:   .1..1.   .....M    1 4
 4:   1111..   ...P..    0 1 2 3          36:   .1.1..   ...PM.    1 3
 5:   11111.   ....P.    0 1 2 3 4        37:   .1.11.   ....P.    1 3 4
 6:   111111   .....P    0 1 2 3 4 5      38:   .1.111   .....P    1 3 4 5
 7:   1111.1   ....M.    0 1 2 3 5        39:   .1.1.1   ....M.    1 3 5
 8:   111.11   ...MP.    0 1 2 4 5        40:   .11..1   ..PM..    1 2 5
 9:   111.1.   .....M    0 1 2 4          41:   .11.1.   ....PM    1 2 4
10:   111..1   ....MP   0 1 2 5          42:   .11.11   .....P    1 2 4 5
11:   11.1.1   ..MP..   0 1 3 5          43:   .111.1   ...PM.    1 2 3 5
12:   11.111   ....P.   0 1 3 4 5        44:   .11111   ....P.    1 2 3 4 5
13:   11.11.   .....M   0 1 3 4          45:   .1111.   .....M    1 2 3 4
14:   11.1..   ....M.   0 1 3            46:   .111..   ....M.    1 2 3
15:   11..1.   ...MP.   0 1 4            47:   .11...   ...M..    1 2
16:   11..11   ....P    0 1 4 5          48:   .1....   ..M...    1
17:   11...1   ...M.    0 1 5            49:   ..1...   .MP...    2
18:   1.1..1   .MP...   0 2 5            50:   ..11..   ...P..    2 3
19:   1.1.1.   ....PM   0 2 4            51:   ..111.   ....P.    2 3 4
20:   1.1.11   .....P   0 2 4 5          52:   ..1111   .....P    2 3 4 5
21:   1.11.1   ...PM.   0 2 3 5          53:   ..11.1   ....M.    2 3 5
22:   1.1111   ....P.   0 2 3 4 5        54:   ..1.11   ...MP.    2 4 5
23:   1.111.   .....M   0 2 3 4          55:   ..1.1.   .....M    2 4
24:   1.11..   ....M.   0 2 3            56:   ..1..1   ....MP    2 5
25:   1.1...   ...M..   0 2              57:   ...1.1   ..MP..    3 5
26:   1..1..   ..MP..   0 3              58:   ...111   ....P.    3 4 5
27:   1..11.   ....P.   0 3 4            59:   ...11.   .....M    3 4
28:   1..111   .....P   0 3 4 5          60:   ...1..   ....M.    3
29:   1..1.1   ....M.   0 3 5            61:   ....1.   ...MP.    4
30:   1...11   ...MP.   0 4 5            62:   ....11   .....P    4 5
31:   1...1.   .....M   0 4              63:   .....1   ....M.    5
```

Figure 5.24: All subsets of a six-element set in an order where either one element is added or removed or one element moves to an adjacent position. The property holds also for the subsequences of k-subsets where $0 \leq k < n$.

```
    void (*visit_)(const ksubset_rec &, long);
public:
    ksubset_rec(ulong n)
    {
        n_ = n;
        rv_ = new long[n_+1];
        ++rv_;
        rv_[-1] = -1UL;
    }
    ~ksubset_rec()
    {
        --rv_;
        delete [] rv_;
    }
```

One has to supply the interval for k (variables `kmin` and `kmax`) and a function that will be called with each subset. The argument `rq` determines which of the sixteen different ordering are chosen, the order can be reversed with nonzero `nq`.

```
    void generate(void (*visit)(const ksubset_rec &, long),
                  long kmin, long kmax, ulong rq, ulong nq=0)
    {
        ct_ = 0;
        rct_ = 0;
        kmin_ = kmin;
        kmax_ = kmax;
        if ( kmin_ > kmax_ )  swap2(kmin_, kmax_);
        if ( kmax_ > n_ )  kmax_ = n_;
        if ( kmin_ > n_ )  kmin_ = n_;
        visit_ = visit;
        rq_ = rq % 4;
        pq_ = (rq>>2) % 4;
        nq_ = nq;
        next_rec(0);
    }
private:
    void next_rec(long d);
};
```

The recursive routine itself is given in [FXT: file comb/ksubset-rec.cc]:

```
void
ksubset_rec::next_rec(long d)
{
    if ( d>kmax_ )  return;
    ++rct_;  // measure computational work
    long rv1 = rv_[d-1];  // left neighbor
    bool q;
    switch ( rq_ % 4 )
    {
    case 0:  q = 1;  break;
    case 1:  q = !(d&1);  break;
    case 2:  q = rv1&1;  break;
    case 3:  q = (d^rv1)&1;  break;
    }
    if ( nq_ )  q = !q;
    long x0 = rv1 + 1;
    long rx = n_ - (kmin_ - d);
    long x1 = min2( n_-1, rx );
#define PCOND(x) if ( (pq_==x) && (d>=kmin_) )  { visit_(*this, d);  ++ct_; }
    PCOND(0);
    if ( q )  // forward:
    {
        PCOND(1);
        for (long x=x0; x<=x1; ++x)  { rv_[d] = x;  next_rec(d+1); }
        PCOND(2);
    }
    else  // backward:
    {
        PCOND(2);
        for (long x=x1; x>=x0; --x)  { rv_[d] = x;  next_rec(d+1); }
```

```
        PCOND(1);
    }
    PCOND(3);
#undef PCOND
}
```

The routine is pretty fast, about 50 million subsets per second can be generated. Usage of the class is exemplified in [FXT: demo comb/ksubset-rec-demo.cc]. Figure 5.23 on page 184 and figure 5.24 were created with this program.

Further routines

```
     1:   ...11  .....  {3, 4}                ...11  .....  {3, 4}
     2:   ..11.  ..P.M  {2, 3}                ..1.1  ..PM.  {2, 4}
     3:   ..111  ....P  {2, 3, 4}             ..111  ...P.  {2, 3, 4}
     4:   ..1.1  ...M.  {2, 4}                ..11.  ....M  {2, 3}
     5:   .11..  .P..M  {1, 2}                .11..  .P.M.  {1, 2}
     6:   .11.1  ....P  {1, 2, 4}             .11.1  ....P  {1, 2, 4}
     7:   .1111  ...P.  {1, 2, 3, 4}          .1111  ...P.  {1, 2, 3, 4}
     8:   .111.  ....M  {1, 2, 3}             .111.  ....M  {1, 2, 3}
     9:   .1.1.  ..M..  {1, 3}                .1.11  ..M.P  {1, 3, 4}
    10:   .1.11  ....P  {1, 3, 4}             .1.1.  ....M  {1, 3}
    11:   .1..1  ...M.  {1, 4}                .1..1  ...MP  {1, 4}
    12:   11...  P...M  {0, 1}                1...1  PM...  {0, 4}
    13:   11..1  ....P  {0, 1, 4}             1..11  ...P.  {0, 3, 4}
    14:   11.11  ...P.  {0, 1, 3, 4}          1..1.  ....M  {0, 3}
    15:   11.1.  ....M  {0, 1, 3}             1.1..  ..PM.  {0, 2}
    16:   1111.  ..P..  {0, 1, 2, 3}          1.1.1  ....P  {0, 2, 4}
    17:   111.1  ...MP  {0, 1, 2, 4}          1.111  ...P.  {0, 2, 3, 4}
    18:   111..  ....M  {0, 1, 2}             1.11.  ....M  {0, 2, 3}
    19:   1.1..  .M...  {0, 2}                111..  .P.M.  {0, 1, 2}
    20:   1.1.1  ....P  {0, 2, 4}             111.1  ....P  {0, 1, 2, 4}
    21:   1.111  ...P.  {0, 2, 3, 4}          1111.  ...PM  {0, 1, 2, 3}
    22:   1.11.  ....M  {0, 2, 3}             11.11  ..M.P  {0, 1, 3, 4}
    23:   1..1.  ..M..  {0, 3}                11.1.  ....M  {0, 1, 3}
    24:   1..11  ....P  {0, 3, 4}             11..1  ...MP  {0, 1, 4}
    25:   1...1  ...M.  {0, 4}                11...  ....M  {0, 1}
```

Figure 5.25: The (25) k-subsets where $2 \le k \le 4$ of a five-element set in a minimal-change (left) and a near-perfect order (right).

Efficient algorithms for the generation of Gray codes for subsets satisfying additional conditions are given in [63]. The class [FXT: `class ksubset_gray` in comb/ksubset-gray.h] for the generation of k-subsets of a set where k lies in a prescribed range uses an algorithm from that source. Its usage is demonstrated in [FXT: demo comb/ksubset-gray-demo.cc], the k-subsets where $2 \le k \le 4$ in the order generated by the algorithm are shown in figure 5.25 (left).

A near-perfect order of the k-subsets with k in a given range is shown at the right of figure 5.25. The list was produced with [FXT: demo comb/ksubset-near-perfect-demo.cc] which uses [FXT: `class ksubset_near_perfect` in comb/ksubset-near-perfect.h]. The recursion is [FXT: file comb/ksubset-near-perfect.cc]:

```
void
ksubset_near_perfect::next_rec(ulong d, ulong p, ulong e1)
{
    long rp = n_ - d;  // remaining places
    long re = kmin_ - e1;  // remaining elements
    if ( rp < re )  return;

    if ( e1 > kmax_ )  return;

    if ( (d==n_) && (e1>=kmin_) )
    {
        ++ct_;  // count objects
        visit_(*this);
    }
```

```
    if ( d<n_ )
    {
        ++rct_;
        ulong q = p & 1;
        if ( nq_ )  q = !q;
        ulong pp;
#define PP  pp=( rv_[d] ? (d&1) : p )
        rv_[d]= q;  PP;  next_rec(d+1, pp, e1+rv_[d]);
        rv_[d]=!q;  PP;  next_rec(d+1, pp, e1+rv_[d]);
#undef PP
    }
}
```

The survey [93] discusses Gray codes for various types of combinatorial objects, such as set partitions. It contains many references for further research.

5.16 Mixed radix numbers in lexicographic order *

```
. . . .
1 . . .         2 . . .
1 1 . .         2 1 . .         . 1 . .
1 1 1 .         2 1 1 .         . 1 1 .
1 1 1 1         2 1 1 1         . 1 1 1
1 1 1 2         2 1 1 2         . 1 1 2
1 1 2 .         2 1 2 .         . 1 2 .
1 1 2 1         2 1 2 1         . 1 2 1
1 1 2 2         2 1 2 2         . 1 2 2
1 1 . 1         2 1 . 1         . 1 . 1
1 1 . 2         2 1 . 2         . 1 . 2
1 2 . .         2 2 . .         . 2 . .
1 2 1 .         2 2 1 .         . 2 1 .
1 2 1 1         2 2 1 1         . 2 1 1
1 2 1 2         2 2 1 2         . 2 1 2
1 2 2 .         2 2 2 .         . 2 2 .
1 2 2 1         2 2 2 1         . 2 2 1
1 2 2 2         2 2 2 2         . 2 2 2
1 2 . 1         2 2 . 1         . 2 . 1
1 2 . 2         2 2 . 2         . 2 . 2
1 . 1 .         2 . 1 .         . . 1 .
1 . 1 1         2 . 1 1         . . 1 1
1 . 1 2         2 . 1 2         . . 1 2
1 . 2 .         2 . 2 .         . . 2 .
1 . 2 1         2 . 2 1         . . 2 1
1 . 2 2         2 . 2 2         . . 2 2
1 . . 1         2 . . 1         . . . 1
1 . . 2         2 . . 2         . . . 2
```

Figure 5.26: A generalization of the lexicographic ordering for binary words for radix 3 and word length 4 (dots for zeros, column-wise for readability). Successive words differ in at most three positions.

The algorithm for the generation of subsets in lexicographic order given section 5.11 on page 177 can be generalized for mixed radix numbers. An example for radix 3 and word length 4 is shown in figure 5.26. The sequence can be generated by using a list of all 4-digit radix-3 numbers, replacing all trailing zeros by a character that is sorted before all numbers (a dot works with ASCII characters), replacing the remaining zeros by a character that is sorted after all numbers (a 'z' with ASCII characters), and sorting the usual way.

A routine for generating successive words in lexicographic order for arbitrary radices is implemented in [FXT: `class mixedradix_lex` in comb/mixedradix-lex.h].

```
class mixedradix_lex
// Mixed radix (generalized) lexicographic sequence
{
public:
    ulong n_;   // Length of strings
    ulong *a_;  // digits
    ulong *m1_; // m1[k] == radix-1 at position k
    ulong e_;   // aux
public:
```

```
mixedradix_lex(const ulong *m, ulong n, ulong mm=0)
    : n_(n)
{
    a_ = new ulong[2*n_ + 1];
    a_[0] = ~0UL; // sentinel
    ++a_;  // nota bene

    m1_ = a_ + n_;

    // if mm given it is used as radix for all digits:
    if ( 0==mm )  for (ulong k=0; k<n_; ++k)  m1_[k] = m[k] - 1;
    else          for (ulong k=0; k<n_; ++k)  m1_[k] = mm - 1;

    first();
}

~mixedradix_lex()
{
    --a_;  // nota bene
    delete [] a_;
}

const ulong *data()  const  { return a_; }

void first()
{
    for (ulong k=0; k<n_; ++k)  a_[k] = 0;
    e_ = 0;
}

void last()
{
    for (ulong k=0; k<n_; ++k)  a_[k] = 0;
    e_ = n_ - 1;
    a_[e_] = m1_[e_];
}
```

The method `next()` updates the string of digits:

```
bool next()
// Return n if current was last
{
    ulong ae = a_[e_] + 1;
    if ( ae <= m1_[e_] )
    {
        a_[e_] = ae;
        if ( e_ < n_-1 )  ++e_;
    }
    else
    {
        a_[e_] = 0;
        long j = e_ - 1;
        while ( 0==a_[j] )  --j;  // can touch sentinel
        if ( j<0 )  { first();  return 0; }  // was last

        ulong aj = a_[j] + 1;
        if ( aj <= m1_[j] )  a_[j] = aj;
        else
        {
            a_[j] = 0;
            ++j;
            a_[j] = 1;
        }

        e_ = j;
        if ( e_ < n_-1 )  ++e_;
    }
    return 1;
}
```

The method `prev()` allows to go back to the previous string:

```
bool prev()
// Return n if current was first
{
    long j = e_;
    while ( 0==a_[j] )  --j;  // can touch sentinel
    if ( j<0 )  { last();  return 0; }  // was first

    ulong aj = a_[j];
    --aj;
```

```
        // can touch sentinel:
        if ( 0==aj )
        {
            if ( 0!=a_[j-1] )
            {
                a_[j] = aj;
                return 1;
            }

            a_[j-1] = m1_[j-1];
        }

        a_[n_-1] = m1_[n_-1];
        e_ = n_ - 1;

        if ( aj )  a_[j] = aj;
        if ( (ulong)j != n_-1 )  a_[j] = aj;

        return  1;
    }
```

radices	4 3 2	2 3 4
0:	. . .	. . .
1:	1 . .	1 . .
2:	1 1 .	1 1 .
3:	1 1 1	1 1 1
4:	1 2 .	1 1 2
5:	1 2 1	1 1 3
6:	1 . 1	1 2 .
7:	2 . .	1 2 1
8:	2 1 .	1 2 2
9:	2 1 1	1 2 3
10:	2 2 .	1 . 1
11:	2 2 1	1 . 2
12:	2 . 1	1 . 3
13:	3 . .	. 1 .
14:	3 1 .	. 1 1
15:	3 1 1	. 1 2
16:	3 2 .	. 1 3
17:	3 2 1	. 2 .
18:	3 . 1	. 2 1
19:	. 1 .	. 2 2
20:	. 1 1	. 2 3
21:	. 2 .	. . 1
22:	. 2 1	. . 2
23:	. . 1	. . 3

Figure 5.27: Numbers in the factorial mixed-radix bases $[4, 3, 2]$ and $[2, 3, 4]$ in lexicographic order.

The program [FXT: demo comb/mixedradix-lex-demo.cc] prints the sequence for radices that can be specified for each digit. The generated sequences for radices 4,3,2 and 2,3,4 are shown in figure 5.27.

The algorithm is constant amortized time (CAT). The worst performance occurs when all digits are radix 2, then about 115 million objects can be created per second. With radix 4 about 170 million, with radix 16 about 270 million objects per second are computed.

For the friends of the command line: the lex-ordered sequence shown as left column of figure 5.27 can be obtained via

```
for n in {0,1,2,3}{0,1,2}{0,1}; do echo $n; done \
  | sed  's/0\+$//; s/0/z/g;' | sort  | sed 's/z/ /g'
```

For the right column use

```
for n in {0,1}{0,1,2}{0,1,2,3}; do [... as before]
```

5.17 Permutations

In this section algorithms for the generation of all permutations are presented. These are typically useful in situations where an exhaustive search over all permutations is needed. At the time of writing the pre-

fascicles of Knuth's *The Art of Computer Programming* Volume 4 became available [DEK4]. Therefore I will not elaborate on the underlying algorithms, for a thorough discussion see Knuth's text.

Algorithms for application, inversion and composition of permutations and the generation of random permutations are given in chapter 2.

5.17.1 Lexicographic order

```
                permutation       inverse
       1:      [ 0, 1, 2, 3 ]    [ 0, 1, 2, 3 ]
       2:      [ 0, 1, 3, 2 ]    [ 0, 1, 3, 2 ]
       3:      [ 0, 2, 1, 3 ]    [ 0, 2, 1, 3 ]
       4:      [ 0, 2, 3, 1 ]    [ 0, 3, 1, 2 ]
       5:      [ 0, 3, 2, 1 ]    [ 0, 3, 2, 1 ]
       6:      [ 0, 3, 1, 2 ]    [ 0, 2, 3, 1 ]
       7:      [ 1, 0, 2, 3 ]    [ 1, 0, 2, 3 ]
       8:      [ 1, 0, 3, 2 ]    [ 1, 0, 3, 2 ]
       9:      [ 1, 2, 0, 3 ]    [ 2, 0, 1, 3 ]
      10:      [ 1, 2, 3, 0 ]    [ 3, 0, 1, 2 ]
      11:      [ 1, 3, 2, 0 ]    [ 3, 0, 2, 1 ]
      12:      [ 1, 3, 0, 2 ]    [ 2, 0, 3, 1 ]
      13:      [ 2, 1, 0, 3 ]    [ 2, 1, 0, 3 ]
      14:      [ 2, 1, 3, 0 ]    [ 3, 1, 0, 2 ]
      15:      [ 2, 0, 1, 3 ]    [ 1, 2, 0, 3 ]
      16:      [ 2, 0, 3, 1 ]    [ 1, 3, 0, 2 ]
      17:      [ 2, 3, 0, 1 ]    [ 2, 3, 0, 1 ]
      18:      [ 2, 3, 1, 0 ]    [ 3, 2, 0, 1 ]
      19:      [ 3, 1, 2, 0 ]    [ 3, 1, 2, 0 ]
      20:      [ 3, 1, 0, 2 ]    [ 2, 1, 3, 0 ]
      21:      [ 3, 2, 1, 0 ]    [ 3, 2, 1, 0 ]
      22:      [ 3, 2, 0, 1 ]    [ 2, 3, 1, 0 ]
      23:      [ 3, 0, 2, 1 ]    [ 1, 3, 2, 0 ]
      24:      [ 3, 0, 1, 2 ]    [ 1, 2, 3, 0 ]
```

Figure 5.28: All permutations of 4 elements in lexicographic order (left) and their inverses (right).

When generated in lexicographic order the permutations appear as if (read as numbers and) sorted numerically in ascending order, see figure 5.28.

Recursive algorithm

A simple recursive algorithm for the generation of the permutations of n elements in lexicographic order can be described as follows: Put each of the n element of the array to the first position and generate all permutations of $n-1$ elements. If n equals one, print the permutation.

The algorithm is implemented in [FXT: `class perm_lex_rec` in comb/permlex-rec.h]:

```
class perm_lex_rec
{
public:
    ulong *pp_;  // permutation
    ulong n_;    // number of elements
    void (*visit_)(const perm_lex_rec &);  // function to call with each permutation

public:
    perm_lex_rec(ulong n)
    {
        n_ = n;
        pp_ = new ulong[n_];
    }

    ~perm_lex_rec()
    { delete [] pp_; }

    void init()
```

```
{
    for (ulong k=0; k<n_; ++k)  pp_[k] = k;
}
void generate(void (*visit)(const perm_lex_rec &))
{
    visit_ = visit;
    init();
    next_rec(0);
}
```

The recursive function `next_rec()` is

```
void next_rec(ulong d)
{
    if ( d==n_-1 )
    {
        visit_(*this);
    }
    else
    {
        for (ulong x=d; x<n_; ++x)  // increasing order
        {
            swap2(pp_[d], pp_[x]);
            next_rec(d+1);
            swap2(pp_[d], pp_[x]);
        }
    }
}
```

The algorithm works because when the function `next()` returns the elements are in the same order as they were initially. The order is the lexicographic order because the elements are moved to the first position in ascending order. With the '`for`'-statement changed to

```
for (ulong x=n_-1; (long)x>=(long)d; --x)
```

the permutation would appear in reversed lexicographic order.

	permutation	inverse
1:	[1, 2, 3, 4, 0]	[4, 0, 1, 2, 3]
2:	[1, 2, 4, 0, 3]	[3, 0, 1, 4, 2]
3:	[1, 3, 0, 4, 2]	[2, 0, 4, 1, 3]
4:	[1, 3, 4, 2, 0]	[4, 0, 3, 1, 2]
5:	[1, 4, 3, 0, 2]	[3, 0, 4, 2, 1]
6:	[1, 4, 0, 2, 3]	[2, 0, 3, 4, 1]
7:	[2, 0, 3, 4, 1]	[1, 4, 0, 2, 3]
8:	[2, 0, 4, 1, 3]	[1, 3, 0, 4, 2]
9:	[2, 3, 1, 4, 0]	[4, 2, 0, 1, 3]
10:	[2, 3, 4, 0, 1]	[3, 4, 0, 1, 2]
11:	[2, 4, 3, 1, 0]	[4, 3, 0, 2, 1]
12:	[2, 4, 1, 0, 3]	[3, 2, 0, 4, 1]
13:	[3, 2, 0, 4, 1]	[2, 4, 1, 0, 3]
14:	[3, 2, 4, 1, 0]	[4, 3, 1, 0, 2]
15:	[3, 0, 1, 4, 2]	[1, 2, 4, 0, 3]
16:	[3, 0, 4, 2, 1]	[1, 4, 3, 0, 2]
17:	[3, 4, 0, 1, 2]	[2, 3, 4, 0, 1]
18:	[3, 4, 1, 2, 0]	[4, 2, 3, 0, 1]
19:	[4, 2, 3, 0, 1]	[3, 4, 1, 2, 0]
20:	[4, 2, 0, 1, 3]	[2, 3, 1, 4, 0]
21:	[4, 3, 1, 0, 2]	[3, 2, 4, 1, 0]
22:	[4, 3, 0, 2, 1]	[2, 4, 3, 1, 0]
23:	[4, 0, 3, 1, 2]	[1, 3, 4, 2, 0]
24:	[4, 0, 1, 2, 3]	[1, 2, 3, 4, 0]

Figure 5.29: All cyclic permutations of 5 elements in lexicographic order (left) and their inverses (right).

A tiny change in the function allows to generate the cyclic permutations (see page 108 section 2.13.7). The only change is to skip the case $x = d$ in the loop:

```
for (ulong x=d+1; x<n_; ++x)
```

The cyclic permutations of five elements in lexicographic order are shown in figure 5.29. The program [FXT: demo comb/permlex-rec-demo.cc] was used to create the figures 5.28 and 5.29.

```
void visit(const perm_lex_rec &P)  // function to call with each permutation
```

```
{
    // Print the permutation
}
int
main(int argc, char **argv)
{
    ulong n = 4;  // Number of elements to permute
    bool cq = 0;  // Whether to generate only cyclic permutations

    perm_lex_rec P(n);

    if ( cq )  P.generate_cyclic(visit);
    else       P.generate(visit);

    return 0;
}
```

The given implementation generates about 40 million permutations per second.

Iterative algorithm

An implementation of an iterative algorithm is given as [FXT: `class perm_lex` in comb/permlex.h].

```
class perm_lex
{
protected:
    ulong *p;  // current permutation of {0, 1, ..., n-1}
    ulong n;   // number of elements to permute
public:
    perm_lex(ulong nn)
    {
        n = (nn > 0 ? nn : 1);
        p = new ulong[n];
        first();
    }

    ~perm_lex() { delete [] p; }

    void first()
    {
        for (ulong i=0; i<n; i++)  p[i] = i;
    }

    const ulong *data()  const  { return p; }
```

It is computed by negating with each swap that occurs with the transitions. Here the sign is represented as zero or one.

The only nontrivial part is the `next()`-method that computes the next permutation with each call. The routine `perm_lex::next()` is based on code by Glenn Rhoads who in turn ascribes the algorithm to Dijkstra.

```
    bool next()
    {
        const ulong n1 = n - 1;
        ulong i = n1;
        do
        {
            --i;
            if ( (long)i<0 )  return  0;  // last sequence is falling seq.
        }
        while ( p[i] > p[i+1] );

        ulong j = n1;
        while ( p[i] > p[j] )  --j;

        swap2(p[i], p[j]);

        ulong r = n1;
        ulong s = i + 1;
        while ( r > s )
        {
            swap2(p[r], p[s]);
            --r;
            ++s;
        }
```

```
        return  1;
    }
};
```

Using the class is no black magic [FXT: demo comb/permlex-demo.cc]:

```
    perm_lex perm(n);
    const ulong *x = perm.data();
    do
    {
        // do something, e.g. just print the permutation:
        for (ulong i=0; i<n; ++i)  cout << x[i] << " ";
        cout << endl;
    }
    while ( perm.next() );
```

The routine generates about 100 million permutations per second.

5.17.2 An order from mixed-radix counting

```
 1:   p=[ 0, 1, 2, 3 ]   d=[ 0, 0, 0 ]   0
 2:   p=[ 1, 0, 2, 3 ]   d=[ 1, 0, 0 ]   1
 3:   p=[ 2, 0, 1, 3 ]   d=[ 0, 1, 0 ]   0
 4:   p=[ 0, 2, 1, 3 ]   d=[ 1, 1, 0 ]   1
 5:   p=[ 1, 2, 0, 3 ]   d=[ 0, 2, 0 ]   0
 6:   p=[ 2, 1, 0, 3 ]   d=[ 1, 2, 0 ]   2
 7:   p=[ 3, 0, 1, 2 ]   d=[ 0, 0, 1 ]   0
 8:   p=[ 0, 3, 1, 2 ]   d=[ 1, 0, 1 ]   1
 9:   p=[ 1, 3, 0, 2 ]   d=[ 0, 1, 1 ]   0
10:   p=[ 3, 1, 0, 2 ]   d=[ 1, 1, 1 ]   1
11:   p=[ 0, 1, 3, 2 ]   d=[ 0, 2, 1 ]   0
12:   p=[ 1, 0, 3, 2 ]   d=[ 1, 2, 1 ]   2
13:   p=[ 2, 3, 0, 1 ]   d=[ 0, 0, 2 ]   0
14:   p=[ 3, 2, 0, 1 ]   d=[ 1, 0, 2 ]   1
15:   p=[ 0, 2, 3, 1 ]   d=[ 0, 1, 2 ]   0
16:   p=[ 2, 0, 3, 1 ]   d=[ 1, 1, 2 ]   1
17:   p=[ 3, 0, 2, 1 ]   d=[ 0, 2, 2 ]   0
18:   p=[ 0, 3, 2, 1 ]   d=[ 1, 2, 2 ]   2
19:   p=[ 1, 2, 3, 0 ]   d=[ 0, 0, 3 ]   0
20:   p=[ 2, 1, 3, 0 ]   d=[ 1, 0, 3 ]   1
21:   p=[ 3, 1, 2, 0 ]   d=[ 0, 1, 3 ]   0
22:   p=[ 1, 3, 2, 0 ]   d=[ 1, 1, 3 ]   1
23:   p=[ 2, 3, 1, 0 ]   d=[ 0, 2, 3 ]   0
24:   p=[ 3, 2, 1, 0 ]   d=[ 1, 2, 3 ]   3
```

Figure 5.30: All permutations of 4 elements in an order where the first $j+1$ elements are reversed when the first j digits change in the mixed-radix counting sequence with radices [2, 3, 4, ...].

A surprisingly simple algorithm for the generation of all permutations is obtained by mixed-radix counting with the radices [2, 3, 4, ...] (column `d=[...]` in figure 5.30). Whenever the first j digits change with an increment then the permutation is updated by reversing the first $j+1$ elements. An implementation is [FXT: `class perm_rev` in comb/permrev.h]:

```
class perm_rev
{
public:
    ulong *d_;  // mixed radix digits with radix = [2, 3, 4, ..., n-1, (sentinel=-1)]
    ulong *p_;  // permutation
    ulong n_;   // permutations of n elements
public:
    perm_rev(ulong n)
    {
        n_ = n;
        p_ = new ulong[n_];
        d_ = new ulong[n_];
```

```
        d_[n-1] = -1UL;  // sentinel
        first();
    }

    ~perm_rev()
    {
        delete [] p_;
        delete [] d_;
    }

    void first()
    {
        for (ulong k=0; k<n_-1; ++k)  d_[k] = 0;
        for (ulong k=0; k<n_; ++k)  p_[k] = k;
    }

    void last()
    {
        for (ulong k=0; k<n_-1; ++k)  d_[k] = k+1;
        for (ulong k=0; k<n_; ++k)  p_[k] = n_-1-k;
    }
```

The update routines are quite concise:

```
    bool next()
    {
        // increment mixed-radix number:
        ulong j = 0;
        while ( d_[j]==j+1 )  { d_[j]=0; ++j; }
        if ( j<n_-1 ) // only if no overflow
        {
            ++d_[j];
            reverse(p_, j+2); // update permutation
        }
        return (j!=n_-1);  // j==n-1 for last permutation
    }

    bool prev()
    {
        // decrement mixed-radix number:
        ulong j = 0;
        while ( d_[j]==0 )  { d_[j]=j+1; ++j; }
        if ( j<n_-1 ) // only if no overflow
        {
            --d_[j];
            reverse(p_, j+2); // update permutation
        }
        return (j!=n_-1);  // j==n-1 for last permutation
    }
};
```

An upper bound for the average number of elements that are moved in the transitions when generating all $N = n!$ permutations is $e \approx 2.7182818$ so the algorithm is CAT. The implementation is actually fast, it generates more than 100 million permutations per second. Usage of the class is as simple as:

```
ulong n = 4;  // Number of elements to permute
perm_rev P(n);
P.first();
do
{
    // Use permutation here
}
while ( P.next() );
```

Figure 5.30 on the previous page was produced with [FXT: demo comb/permrev-demo.cc].

The generated sequence of permutation can easily and with little chance of error be created by hand, try it! One can further easily convert a mixed-radix (factorial) number into the corresponding permutation as exemplified for the 16-th permutation ($15 = 1 \cdot 1 + 1 \cdot 2 + 2 \cdot 6$, so `d=[1,1,2]`):

```
 1:   p=[ 0, 1, 2, 3 ]   d=[ 0, 0, 0 ] // start
13:   p=[ 2, 3, 0, 1 ]   d=[ 0, 0, 2 ] // right rotate all elements twice
15:   p=[ 0, 2, 3, 1 ]   d=[ 0, 1, 2 ] // right rotate first three elements
16:   p=[ 2, 0, 3, 1 ]   d=[ 1, 1, 2 ] // right rotate first two elements
```

Therefore the class has the method

```
    void goto_d(const ulong *d)
    // Goto permutation corresponding to d[] (i.e. unrank d[]).
    // d[] must be a valid factorial mixed-radix string:
    // d[]==[d(0), d(1), d(2), ..., d(n-2)]  (n-1 elements) where 0<=d(j)<=j+1
    {
        for (ulong k=0; k<n_; ++k)  p_[k] = k;
        for (ulong k=0; k<n_-1; ++k)  d_[k] = d[k];
        for (long j=n_-2; j>=0; --j)  rotate_right(p_, j+2, d_[j]);
    }
```

5.17.3 Minimal-change order

```
          permutation   swap    inverse p.
    #  0:   0 1 2 3    (0, 0)    0 1 2 3
    #  1:   0 1 3 2    (3, 2)    0 1 3 2
    #  2:   0 3 1 2    (2, 1)    0 2 3 1
    #  3:   3 0 1 2    (1, 0)    1 2 3 0
    #  4:   3 0 2 1    (3, 2)    1 3 2 0
    #  5:   0 3 2 1    (0, 1)    0 3 2 1
    #  6:   0 2 3 1    (1, 2)    0 3 1 2
    #  7:   0 2 1 3    (2, 3)    0 2 1 3
    #  8:   2 0 1 3    (1, 0)    1 2 0 3
    #  9:   2 0 3 1    (3, 2)    1 3 0 2
    # 10:   2 3 0 1    (2, 1)    2 3 0 1
    # 11:   3 2 0 1    (1, 0)    2 3 1 0
    # 12:   3 2 1 0    (3, 2)    3 2 1 0
    # 13:   2 3 1 0    (0, 1)    3 2 0 1
    # 14:   2 1 3 0    (1, 2)    3 1 0 2
    # 15:   2 1 0 3    (2, 3)    2 1 0 3
    # 16:   1 2 0 3    (0, 1)    2 0 1 3
    # 17:   1 2 3 0    (3, 2)    3 0 1 2
    # 18:   1 3 2 0    (2, 1)    3 0 2 1
    # 19:   3 1 2 0    (1, 0)    3 1 2 0
    # 20:   3 1 0 2    (2, 3)    2 1 3 0
    # 21:   1 3 0 2    (0, 1)    2 0 3 1
    # 22:   1 0 3 2    (1, 2)    1 0 3 2
    # 23:   1 0 2 3    (2, 3)    1 0 2 3
```

Figure 5.31: The permutations of 4 elements in a minimal-change order.

When generated in minimal-change order[1] the permutations in a way that between each consecutive two exactly two elements are swapped, see figure 5.31. Note that the swapped pairs are always neighboring elements. Further, in the sequence of the inverse permutations the swapped pair always consists of elements that are neighbors in the unpermuted array.

Often one will only use the indices of the swapped elements to update the visited configurations. A property of the algorithm used is that the inverse permutations are available. The corresponding class [FXT: `class perm_minchange` in comb/permminchange.h] is

```
class perm_minchange
{
public:
    ulong *p_;  // p[n] contains a permutation of {0, 1, ..., n-1}
    ulong *ip_; // ip[n] contains the inverse permutation of p[]
    ulong *d_;  // aux
    ulong *ii_; // aux
    ulong n_;   // number of elements to permute
    ulong sw1_, sw2_; // indices of elements swapped most recently
public:
    perm_minchange(ulong nn)
    {
        n_ = ( nn<1 ? 1 : nn );
        p_  = new ulong[n_];
```

[1] There is more than one minimal-change order. For example, reversing the order yields another one.

```
        ip_ = new ulong[n_];
        d_  = new ulong[n_];
        ii_ = new ulong[n_];
        first();
    }
    ~perm_minchange()
    {
        delete [] p_;
        delete [] ip_;
        delete [] d_;
        delete [] ii_;
    }
    void first()
    {
        for (ulong i=0; i<n_; i++)
        {
            p_[i] = ip_[i] = i;
            d_[i] = -1UL;
            ii_[i] = 0;
        }
        sw1_ = sw2_ = 0;
    }
    ulong next()  { return make_next(n_-1); }  // return 0 if on last permutation
    const ulong *data()  const  { return p_; }
    const ulong *invdata()  const  { return ip_; }
    void get_swap(ulong &s1, ulong &s2)  const  { s1=sw1_; s2=sw2_; }
```

The routine to update the permutation is

```
protected:
    ulong make_next(ulong m)
    {
        ulong i = ii_[m];
        ulong ret = 1;
        if ( i==m )
        {
            d_[m] = -d_[m];
            if ( 0!=m )  ret = make_next(m-1);
            else         ret = 0;
            i = -1UL;
        }
        if ( (long)i>=0 )
        {
            ulong j = ip_[m];
            ulong k = j + d_[m];
            ulong z = p_[k];
            p_[j] = z;
            p_[k] = m;
            ip_[z] = j;
            ip_[m] = k;
            sw1_ = j;  // note that sw1_ == sw2_ +-1 (adjacent positions)
            sw2_ = k;
        }
        ii_[m] = i + 1;
        return  ret;
    }
};
```

The central block (`if ( (long)i>=0 ) {...}`) is based on code by Frank Ruskey / Glenn Rhoads.

Usage of the class is straightforward [FXT: demo comb/permminchange-demo.cc]:

```
    ulong n = 4;
    perm_minchange perm(n);
    const ulong *x = perm.data();
    const ulong *ix = perm.invdata();
    ulong sw1, sw2;
    do
    {
        // do something, e.g. just print the permutation:
```

```
        for (ulong i=0; i<n; ++i)  cout << x[i] << " ";
        // sometimes one only uses the indices swapped ...
        perm.get_swap(sw1, sw2);
        cout << "  swap: (" << sw1 << ", " << sw2 << ") ";
        // ... inverse permutation courtesy of the algorithm
        for (ulong i=0; i<n; ++i)  cout << ix[i] << " ";
    }
    while ( perm.next() );
```

About 85 million permutations per second are generated.

An alternative implementation using the algorithm of Trotter (based on code by Helmut Herold) is given as [FXT: `class perm_trotter` in comb/permtrotter.h]. The update routine is

```
    bool next()
    {
        ulong k = 0;
        ulong m = 0;
        ulong y = p_[m] + d_[m];
        p_[m] = y;
        while ( (y==n_-m) || (y==0) )
        {
            if ( y==0 )
            {
                d_[m] = 1;
                ++k;
            }
            else  d_[m] = -1UL;
            if ( m==n_-2 )
            {
                sw1_ = n_ - 1;
                sw2_ = sw1_ - 1;
                swap2(x_[sw1_], x_[sw2_]);
                // y = 1;
                return 0;
            }
            else
            {
                ++m;
                y = p_[m] + d_[m];
                p_[m] = y;
            }
        }
        sw1_ = y + k;
        sw2_ = sw1_ - 1;  // note: adjacent positions
        swap2(x_[sw1_], x_[sw2_]);
        return 1;
    }
```

The routine is very fast, about 140 million permutations can be generated per second (about 15 cycles per update) [FXT: demo comb/permtrotter-demo.cc].

5.17.4 Derangement order

The enumeration of permutations in derangement order is characterized by the fact that two successive permutations have no element at the same position, see figure 5.32 on the next page. There is no such sequence for $n = 3$.

The utility class, that implements the underlying algorithm is [FXT: `class perm_derange` in comb/permderange.h].

```
class perm_derange
{
public:
    ulong n_;   // number of elements
    ulong *x_;  // current permutation
    ulong ctm_; // counter modulo n
    perm_trotter* pt;
```

```
 0:   0 1 2 3
 1:   3 0 1 2
 2:   1 2 3 0
 3:   2 3 0 1
 4:   1 0 2 3
 5:   3 1 0 2
 6:   0 2 3 1
 7:   2 3 1 0
 8:   1 2 0 3
 9:   3 1 2 0
10:   2 0 3 1
11:   0 3 1 2
12:   2 1 0 3
13:   3 2 1 0
14:   1 0 3 2
15:   0 3 2 1
16:   2 0 1 3
17:   3 2 0 1
18:   0 1 3 2
19:   1 3 2 0
20:   0 2 1 3
21:   3 0 2 1
22:   2 1 3 0
23:   1 3 0 2
```

Figure 5.32: The permutations of 4 elements in derangement order.

```
public:
    perm_derange(ulong nn)
    {
        n_ = ( nn<4 ? 4 : nn );  // 2: trivial, 3: no solution exists,  4..: ok
        x_ = new ulong[n_];
        pt = new perm_trotter(n_-1);
        first();
    }

    ~perm_derange()
    {
        delete [] x_;
        delete pt;
    }

    void first()
    {
        ctm_ = 0;
        for (ulong k=0; k<n_; ++k)  x_[k] = k;
    }
```

The routine to update the permutation is

```
    bool next()
    {
        ++ctm_;

        if ( ctm_>=n_ ) // every n steps: need next perm_trotter
        {
            ctm_ = 0;
            if ( 0==pt->next() )  return 0;

            // copy in:
            const ulong *xx = pt->data();
            for (ulong k=0; k<n_-1; ++k)  x_[k] = xx[k];
            x_[n_-1] = n_-1;  // last element
        }
        else  // rotate
        {
            if ( ctm_==n_-1 )
            {
                rotl1(x_, n_);
            }
            else // last two swapped
            {
                rotr1(x_, n_);
                if ( ctm_==n_-2 )  rotr1(x_, n_);
            }
        }
        return 1;
    }
```

The routines `rotr1()` and `rotl1()` rotate an array one position to the right and left, respectively. These rotations are the performance bottleneck, the cost of one update of a length-n permutation is proportional to n. Still, about 37 million permutations per second are generated for $n = 12$.

The above listing can be generated via [FXT: demo comb/permderange-demo.cc]:

```
ulong n = 4;
perm_derange perm(n);
const ulong *x = perm.data();
ulong ct = 0;
do
{
    cout << "   " << setw(3) << ct << ":   ";
    ++ct;
    for (ulong i=0; i<n; ++i)  cout << x[i] << " ";
    cout << endl;
}
while ( perm.next() );
```

5.17.5 Star-transposition order

```
          permutation   swap    inverse p.
     0:   0 1 2 3    (0, 3)    0 1 2 3
     1:   1 0 2 3    (0, 1)    1 0 2 3
     2:   2 0 1 3    (0, 2)    1 2 0 3
     3:   0 2 1 3    (0, 1)    0 2 1 3
     4:   1 2 0 3    (0, 2)    2 0 1 3
     5:   2 1 0 3    (0, 1)    2 1 0 3
     6:   3 1 0 2    (0, 3)    2 1 3 0
     7:   0 1 3 2    (0, 2)    0 1 3 2
     8:   1 0 3 2    (0, 1)    1 0 3 2
     9:   3 0 1 2    (0, 2)    1 2 3 0
    10:   0 3 1 2    (0, 1)    0 2 3 1
    11:   1 3 0 2    (0, 2)    2 0 3 1
    12:   2 3 0 1    (0, 3)    2 3 0 1
    13:   3 2 0 1    (0, 1)    2 3 1 0
    14:   0 2 3 1    (0, 2)    0 3 1 2
    15:   2 0 3 1    (0, 1)    1 3 0 2
    16:   3 0 2 1    (0, 2)    1 3 2 0
    17:   0 3 2 1    (0, 1)    0 3 2 1
    18:   1 3 2 0    (0, 3)    3 0 2 1
    19:   2 3 1 0    (0, 2)    3 2 0 1
    20:   3 2 1 0    (0, 1)    3 2 1 0
    21:   1 2 3 0    (0, 2)    3 0 1 2
    22:   2 1 3 0    (0, 1)    3 1 0 2
    23:   3 1 2 0    (0, 2)    3 1 2 0
```

Figure 5.33: The permutations of 4 elements in star-transposition order.

Knuth ([DEK4], fasc2B, p.19) gives an algorithm that generates the permutations ordered in a way that each two successive entries in the list differ by a swap of element zero with some other element (star transposition), figure 5.33.

The implementation of the algorithm, ascribed to Gideon Ehrlich, is given in [FXT: `class perm_star` in comb/permstar.h]. The generation of the inverse permutations is an option that is activated by the `#define PERM_STAR_WITH_INVERSE`. If the successive permutations differ by a transposition

```
swap2(a_[0], a_[swp_]);
```

then the inverse can be updated as

```
swap2(ia_[a_[0]], ia_[a_[swp_]]);
```

Note that in the sequence of the inverse permutations the zero is always moved. About 77 million permutations per second are generated and about 115 million when the inverse permutation is not computed. The listing shown in figure 5.33 can be obtained with [FXT: demo comb/permstar-demo.cc].

5.17.6 Permutations with restricted prefixes

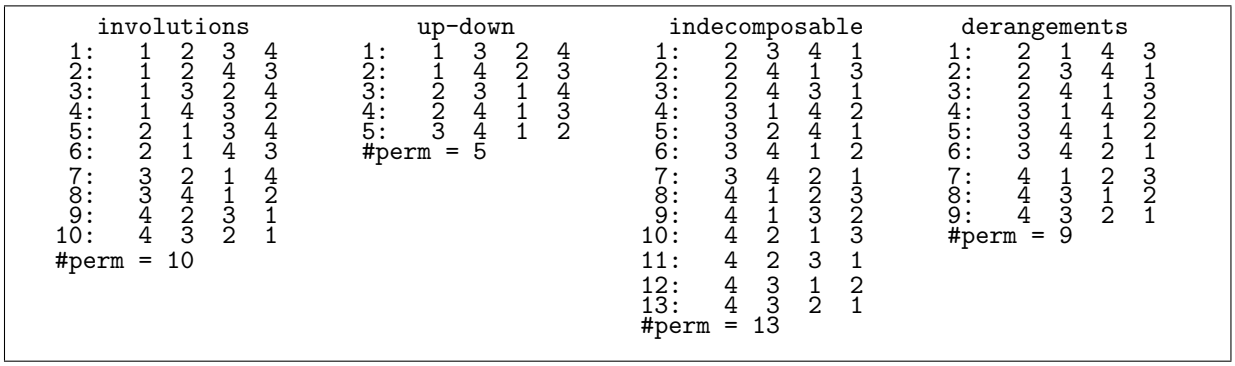


Figure 5.34: Examples of permutations subject to conditions on the prefixes. From left to right: involutions, up-down permutations, indecomposable permutations and derangements.

Knuth [DEK4] gives an algorithm for the generation of all permutations of the set $\{1, 2, \ldots, n\}$ in lex order subject to given restrictions on the prefixes. The straightforward implementation is given as [FXT: `class perm_restrpref` in comb/permrestrpref.h]. The condition (as a function pointer) has to be supplied upon creation of an instance of the class. If performance is crucial both the condition function and the 'visit' function should be inlined.

The program [FXT: demo comb/permrestrpref-demo.cc] demonstrates the usage, it can be used to generate all involutions, up-down, indecomposable, or derangement permutations. The condition functions in the demo are implemented as follows. For involutions:

```
bool cond_inv(const ulong *a, ulong k)
{
    ulong ak = a[k];
    if ( (ak<=k) && (a[ak]!=k) )  return false;
    return true;
}
```

For the *up-down permutations* (or *alternating permutations*):

```
bool cond_updown(const ulong *a, ulong k)
// up-down condition: a1<a2>a3<a4> ...
{
    if ( k<2 ) return true;
    if ( (k%2) )  return ( a[k]<a[k-1] );
    else          return ( a[k]>a[k-1] );
}
```

For *indecomposable* (or *connected*) permutations:

```
ulong N;  // set to n in main()
bool cond_indecomp(const ulong *a, ulong k)
// indecomposable condition: {a1,...,ak} != {1,...,k} for all k<n
{
    if ( k==N )  return true;
    for (ulong i=1; i<=k; ++i)  if ( a[i]>k )  return true;
    return false;
}
```

For derangements:

```
bool cond_derange(const ulong *a, ulong k)
// derangement condition: f[k]!=k for all k
{
    return ( a[k]!=k );
}
```

Figure 5.34 shows the corresponding sequence of the permutations for $n = 4$.

The number of permutations of certain types *

The sequence $I(n)$ of the number of involutions starts as ($n \geq 1$)

```
1, 2, 4, 10, 26, 76, 232, 764, 2620, 9496, 35696, 140152, 568504, 2390480, ...
```

This is sequence A000085 of [OEIS]. The relation

$$I(n) \quad = \quad I(n-1) + (n-1)\, I(n-2) \tag{5.8}$$

can be used for the computation of a table of $I(n)$. The sequence above was computed via

```
N=20; v=vector(N);
v[1]=1; v[2]=2;
for(n=3,N,v[n]=v[n-1]+(n-1)*v[n-2]);
  [1, 2, 4, 10, 26, 76, ... ]
```

An exponential generating function is

$$\sum_{k=0}^{\infty} \frac{I(k)}{k!} \quad = \quad \exp\left(x + x^2/2\right) \tag{5.9}$$

The relation is given in [WilfGF], p.85, as the special case $m = 2$ of the exponential generating function

$$\exp\left(\sum_{d \backslash m} x^d/d\right) \tag{5.10}$$

for the number permutations whose m-th power is the identity.

The sequence $A(n)$ of the number of alternating permutations starts as ($n \geq 1$)

```
1, 1, 2, 5, 16, 61, 272, 1385, 7936, 50521, 353792, 2702765, 22368256, ...
```

It is sequence A000111 of [OEIS], the sequence of the *Euler numbers*. The list can be computed using the relation

$$A(n) \quad = \quad \frac{1}{2} \sum_{k=0}^{n-1} \binom{n-1}{k} A(k)\, A(n-1-k) \tag{5.11}$$

Compute $A(0) = 1$, $A(1)$, ..., $A(N)$ via

```
N=20; v=vector(N+1);
v[0+1]=1;  v[1+1]=1;  v[2+1]=1;  \\ start with zero: v[x] == A(x-1)
for(n=3,N,v[n+1]=1/2*sum(k=0,n-1,binomial(n-1,k)*v[k+1]*v[n-1-k+1]));  v
  [1, 1, 1, 2, 5, 16, 61, 272, ... ]
```

The sequence $C(n)$ of the number of indecomposable permutations starts as ($n \geq 1$)

```
1, 1, 3, 13, 71, 461, 3447, 29093, 273343, 2829325, 31998903, 392743957, ...
```

This is sequence A003319 of [OEIS]. Compute $C(n)$ using

$$C(n) \quad = \quad n! - \sum_{k=1}^{n-1} k!\, C(n-k) \tag{5.12}$$

```
N=20; v=vector(N);
for(n=1,N,v[n]=n!-sum(k=1,n-1,k!*v[n-k]));  v
  [1, 1, 3, 13, 71, 461, 3447, ... ]
```

The sequence $D(n)$ of the number of derangements starts as ($n \geq 1$)

```
0, 1, 2, 9, 44, 265, 1854, 14833, 133496, 1334961, 14684570, 176214841, ...
```

This is sequence A000166 of [OEIS], the *subfactorial numbers*. Compute $D(n)$ using either of

$$D(n) = (n-1)\,[D(n-1)+D(n-2)] \tag{5.13a}$$
$$D(n) = n\,D(n-1)+(-1)^n \tag{5.13b}$$
$$D(n) = \sum_{k=0}^{n}(-1)^{n-k}\,\frac{n!}{(n-k)!} \tag{5.13c}$$
$$D(n) = \lfloor (n!+1/2)/e \rfloor \tag{5.13d}$$

where $e=\exp(1)$. We use the recursion 5.13b:

```
N=20; v=vector(N);
v[1]=0;  v[2]=1;
for(n=3,N,v[n]=(n-1)*(v[n-1]+v[n-2]));  v
  [0, 1, 2, 9, 44, 265, 1854, 14833, ... ]
```

5.18 Gray codes for Fibonacci and Pell words

5.18.1 Gray code for Fibonacci words

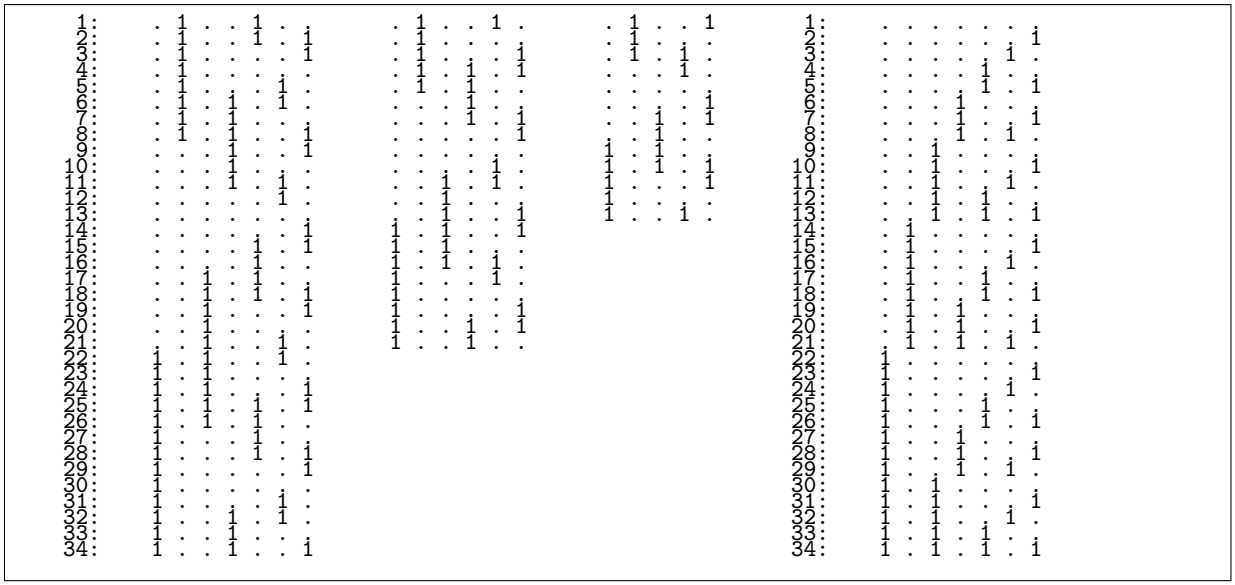


Figure 5.35: Gray codes through the first 34, 21, and 13 Fibonacci words (left) and the first 34 Fibonacci words in counting order (right). Dots are used for zeros.

A recursive routine to generate the Fibonacci words (binary words not containing two consecutive ones) can be given as follows:

```
ulong n; // Length of words
ulong *rv;  // record of visits in graph

void fib_rec(ulong d)
{
    if ( d>=n )  visit();
    else
    {
         rv[d]=0;  fib_rec(d+1);
         rv[d]=1;  rv[d+1]=0;  fib_rec(d+2);
    }
}
```

We allocate one extra element to avoid if-statements in the code:

```
int
main()
{
    n = 7;
    rv = new ulong[n+1];
    fib_rec(0);
    return 0;
}
```

The output (assuming `visit()` simply prints the array) is given in the right of figure 5.35 on the preceding page. It turns out that a simple modification of the routine generates a Gray code through the Fibonacci words.

```
void fib_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        z = !z;
        if ( z )
        {
            rv[d]=0;  fib_rec(d+1, z);
            rv[d]=1;  rv[d+1]=0;  fib_rec(d+2, z);
        }
        else
        {
            rv[d]=1;  rv[d+1]=0;  fib_rec(d+2, z);
            rv[d]=0;  fib_rec(d+1, z);
        }
    }
}
```

The variable z controls the direction in the recursion, it is changed unconditionally with each step. The `if-else` blocks can be merged into

```
        rv[d]=!z;  rv[d+1]= z;  fib_rec(d+1+!z, z);
        rv[d]= z;  rv[d+1]=!z;  fib_rec(d+1+ z, z);
```

If we count the number of non-terminal recursions (that is, incrementing a counter in the `else` branch), we find that this number equals the number of generated words minus one. Thereby the algorithm is CAT (*constant amortized time*). It is also fast in practice, about 80 million objects are generated per second. The code is given in [FXT: demo comb/fibgray-rec-demo.cc]. A bit-level algorithm is given in section 1.31 on page 65.

The analogue routine for the Gray code through the binary words changes the direction in the recursion whenever a one is inserted:

```
void bin_rec(ulong d, bool z)
{
    if ( d==n )  visit();
    else
    {
        if ( z )
        {
            rv[d]=0;  bin_rec(d+1, z);
            rv[d]=1;  bin_rec(d+1, !z);
        }
        else
        {
            rv[d]=1;  bin_rec(d+1, !z);
            rv[d]=0;  bin_rec(d+1, z);
        }
    }
}
```

The direct equivalent with the Fibonacci type recursion

```
void foo_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        if ( z )
        {
            rv[d]=0;  foo_rec(d+1, z);
            rv[d]=1;  rv[d+1]=0;  foo_rec(d+2, !z);
        }
        else
```

```
        {
            rv[d]=1;  rv[d+1]=0;  foo_rec(d+2, !z);
            rv[d]=0;  foo_rec(d+1, z);
        }
    }
}
```

gives the Fibonacci words in the order of the binary Gray code which is not a Gray code for the Fibonacci words (for $n = 5$):

```
 1:    1 . . . .
 2:    1 . . . 1
 3:    1 . . 1 .
 4:    1 . 1 . 1
 5:    1 1 . . .
 6:    . 1 . . .
 7:    . 1 . . 1
 8:    . 1 . 1 .
 9:    . . 1 . .
10:    . . 1 . 1
11:    . . . 1 .
12:    . . . . 1
13:    . . . . .
```

Note that with the n-bit Fibonacci Gray code the number of ones in the first and last, second and second-last, etc. tracks are equal. Thereby the sequence of reversed words is also a Fibonacci Gray code.

5.18.2 Gray code for Pell words

```
.....1111111111111111112222222    .....1111111111111111112222222
...111111111222.....1111111222..1.    ...1111111222222111111.....111.
..1112..1112..1112..1112..1112    ..11122111..........11122111..1112
.12.12..12.12..12.12.12..12.12.12.    .1221....1221..1221..1221....1221..1221..
```

Figure 5.36: Four digit Pell words in counting order (left) and Gray code order (right). The lowest row is the least significant digit, dots are used for zeros.

A Gray code of the Pell words (ternary words without the substrings `"21"` and `"22"`, see section 24.13 on page 666) can be computed as follows:

```
void pell_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        if ( 0==z )
        {
            rv[d]=0;  pell_rec(d+1, z);
            rv[d]=1;  pell_rec(d+1, zq^z);
            rv[d]=2;  rv[d+1]=0;  pell_rec(d+2, z);
        }
        else
        {
            rv[d]=2;  rv[d+1]=0;  pell_rec(d+2, z);
            rv[d]=1;  pell_rec(d+1, zq^z);
            rv[d]=0;  pell_rec(d+1, z);
        }
    }
}
```

The global boolean variable `zq` controls whether the Gray code (right of figure 5.36) or the counting order (left of figure 5.36) is generated. The code is given in [FXT: demo comb/pellgray-rec-demo.cc]. The computation of a function whose power series coefficients are related to the Pell Gray code is described in section 24.13.3 on page 669.

5.19 A Gray code for sparse signed binary words

Figure 5.37 on the next page shows a minimal-change order (Gray code) for the sparse signed binary words (M.Paul's representation, see section 1.26 on page 55). Note that we allow a digit to switch between +1

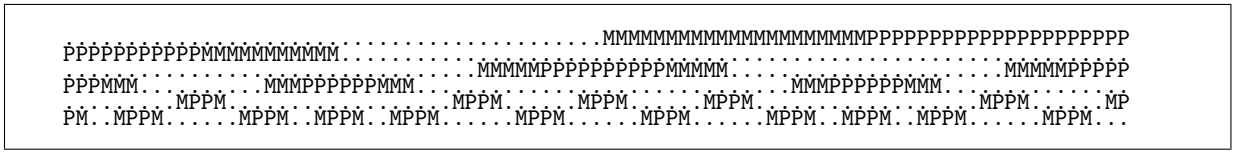


Figure 5.37: A Gray code through the 85 sparse 6-bit signed binary words. Dots are used for zeros, 'P'= +1 and 'M'= −1.

and −1. If all words with any positive digit ('P') are omitted then we obtain the Gray code for Fibonacci words given in section 5.18 on page 202.

A recursive routine for the generation of the Gray code is given in [FXT: demo comb/paulgray-rec-demo.cc]:

```
void sb_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        if ( 0==z )
        {
            rv[d]=0;  sb_rec(d+1, 1);
            rv[d]=-1;  rv[d+1]=0;  sb_rec(d+2, 1);
            rv[d]=+1;  rv[d+1]=0;  sb_rec(d+2, 0);
        }
        else
        {
            rv[d]=+1;  rv[d+1]=0;  sb_rec(d+2, 1);
            rv[d]=-1;  rv[d+1]=0;  sb_rec(d+2, 0);
            rv[d]=0;  sb_rec(d+1, 0);
        }
    }
}
```

The number $P(n)$ of positive n-digit sparse signed binary numbers is (sequence A005578 of [OEIS])

```
   n: 0 1 2 3  4  5  6  7  8   9  10   11   12   13    14    15    16
P(n): 1 2 3 6 11 22 43 86 171 342 683 1366 2731 5462 10923 21846 43691
```

The number $S(n)$ of n-digit sparse signed binary numbers (of both signs) is (sequence A001045 of [OEIS])

```
   n: 0 1 2  3  4  5  6  7   8   9   10   11   12   13    14    15    16
S(n): 1 3 5 11 21 43 85 171 341 683 1365 2731 5461 10923 21845 43691 87381
```

We have

$$P(n) \;=\; \frac{2^{n+1}+1+e}{3} \;=\; 2\,P(n-1)-1-e \;=\; S(n-1)+e \tag{5.14a}$$

$$S(n) \;=\; \frac{2^{n+2}-1+2\,e}{3} \;=\; 2\,S(n-1)-1+2\,e \;=\; 2\,P(n)-1 \tag{5.14b}$$

where $e = n \bmod 2$.

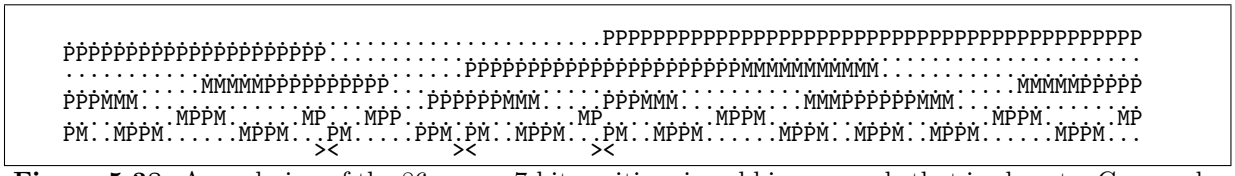


Figure 5.38: An ordering of the 86 sparse 7-bit positive signed binary words that is almost a Gray code. The transitions that are not minimal are marked with '><'

If we start with the following routine that calls `sb_rec()` only after a one has been inserted, we obtain an ordering of the positive numbers:

```
void pos_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
```

```
    {
        if ( 0==z )
        {
            rv[d]=0;  pos_rec(d+1, 1);
            rv[d]=+1;  rv[d+1]=0;  sb_rec(d+2, 1);
        }
        else
        {
            rv[d]=+1;  rv[d+1]=0;  sb_rec(d+2, 0);
            rv[d]=0;  pos_rec(d+1, 0);
        }
    }
}
```

The ordering obtained with n-digit words is a Gray code, except for $n-4$ transitions. The result for $n=7$ is shown in figure 5.38 on the preceding page.

5.20 A Gray code from Hilbert's space-filling curve *

Apart from the nice images one can create from Hilbert's space-filling curve the algorithm (given in section 4.7 on page 147) can be used to generate a Gray code using the following observation: with each increment by one of the linear coordinate t exactly one of the generated x and y changes its value by ± 1. Therefore the concatenation of the[2] Gray codes of x and y gives another Gray code. It turns out that the routine is most effective if the bits of (the Gray code of) x and y are interlaced as with the bit-zip function given in section 1.15 on page 32. The routine is [FXT: `hilbert_gray_code()` in bits/hilbert.cc]:

```
ulong
hilbert_gray_code(ulong t)
//
// A Gray code based on the function lin2hilbert().
// Equivalent to the following sequence of statements:
//   hilbert(t, x, y);
//   x=gray_code(x);
//   y=gray_code(y);
//   return bitzip(x, y);
{
    ulong g = 0;
    ulong c01 = 2;  // 0 for transposed output (swapped x, y)
    for (ulong i=0; i<(BITS_PER_LONG/2); ++i)
    {
        ulong abi = t >> (BITS_PER_LONG-2);
        t <<= 2;

        ulong st = htab[ (c01<<2) | abi ];
        c01 = st & 3;

        g <<= 2;
        g |= (st>>2);
    }
    g ^= ( (g & 0x55555555UL) >> 2 );  // 32 bit version
    g ^= ( (g & 0xaaaaaaaaUL) >> 2 );  // 32 bit version
    return g;
}
```

The demo [FXT: demo bits/hilbert-demo.cc] should clarify the idea, it gives the output shown in figure 5.39 on the next page.

5.21 A string substitution engine

String substitution: A finite set of symbols ('alphabet'), a set of substitution rules that map symbols to words (strings of symbols) and a start value ('axiom'). The rules have to be applied in parallel. An example, let $\{x, A, B\}$ be the symbols and $\{x \mapsto A,\ A \mapsto Bx,\ B \mapsto Bx\}$ the substitution rules. Start

[2] actually any Gray code does it, we use the standard binary version here.

```
    t ->    binary(x)  binary(y)= ( x,  y)   gx=gray(x)  gy=gray(y)  z=bitzip(gx,gy) == g
#  0 ->    x=......   y=...... = ( 0,  0)   gx=......   gy=......   z=......   g=......
#  1 ->    x=.....1   y=...... = ( 1,  0)   gx=.....1   gy=......   z=.....1   g=.....1
#  2 ->    x=.....1   y=.....1 = ( 1,  1)   gx=.....1   gy=.....1   z=....11   g=....11
#  3 ->    x=......   y=.....1 = ( 0,  1)   gx=......   gy=.....1   z=....1.   g=....1.
#  4 ->    x=......   y=....1. = ( 0,  2)   gx=......   gy=....11   z=..1.1.   g=..1.1.
#  5 ->    x=......   y=....11 = ( 0,  3)   gx=......   gy=....1.   z=..1...   g=..1...
#  6 ->    x=.....1   y=....11 = ( 1,  3)   gx=.....1   gy=....1.   z=..1..1   g=..1..1
#  7 ->    x=.....1   y=....1. = ( 1,  2)   gx=.....1   gy=....11   z=..1.11   g=..1.11
#  8 ->    x=....1.   y=....1. = ( 2,  2)   gx=....11   gy=....11   z=..1111   g=..1111
#  9 ->    x=....1.   y=....11 = ( 2,  3)   gx=....11   gy=....1.   z=..11.1   g=..11.1
# 10 ->    x=....11   y=....11 = ( 3,  3)   gx=....1.   gy=....1.   z=..11..   g=..11..
# 11 ->    x=....11   y=....1. = ( 3,  2)   gx=....1.   gy=....11   z=..111.   g=..111.
# 12 ->    x=....11   y=.....1 = ( 3,  1)   gx=....1.   gy=.....1   z=...11.   g=...11.
# 13 ->    x=....1.   y=.....1 = ( 2,  1)   gx=....11   gy=.....1   z=...111   g=...111
# 14 ->    x=....1.   y=...... = ( 2,  0)   gx=....11   gy=......   z=...1.1   g=...1.1
# 15 ->    x=....11   y=...... = ( 3,  0)   gx=....1.   gy=......   z=...1..   g=...1..
# 16 ->    x=...1..   y=...... = ( 4,  0)   gx=...11.   gy=......   z=.1.1..   g=.1.1..
# 17 ->    x=...1..   y=.....1 = ( 4,  1)   gx=...11.   gy=.....1   z=.1.11.   g=.1.11.
# 18 ->    x=...1.1   y=.....1 = ( 5,  1)   gx=...111   gy=.....1   z=.1.111   g=.1.111
# 19 ->    x=...1.1   y=...... = ( 5,  0)   gx=...111   gy=......   z=.1.1.1   g=.1.1.1
# 20 ->    x=...11.   y=...... = ( 6,  0)   gx=...1.1   gy=......   z=.1...1   g=.1...1
# 21 ->    x=...111   y=...... = ( 7,  0)   gx=...1..   gy=......   z=.1....   g=.1....
# 22 ->    x=...111   y=.....1 = ( 7,  1)   gx=...1..   gy=.....1   z=.1..1.   g=.1..1.
# 23 ->    x=...11.   y=.....1 = ( 6,  1)   gx=...1.1   gy=.....1   z=.1..11   g=.1..11
# 24 ->    x=...11.   y=....1. = ( 6,  2)   gx=...1.1   gy=....11   z=.11.11   g=.11.11
# 25 ->    x=...111   y=....1. = ( 7,  2)   gx=...1..   gy=....11   z=.11.1.   g=.11.1.
# 26 ->    x=...111   y=....11 = ( 7,  3)   gx=...1..   gy=....1.   z=.11...   g=.11...
# 27 ->    x=...11.   y=....11 = ( 6,  3)   gx=...1.1   gy=....1.   z=.11..1   g=.11..1
# 28 ->    x=...1.1   y=....11 = ( 5,  3)   gx=...111   gy=....1.   z=.111.1   g=.111.1
# 29 ->    x=...1.1   y=....1. = ( 5,  2)   gx=...111   gy=....11   z=.11111   g=.11111
# 30 ->    x=...1..   y=....1. = ( 4,  2)   gx=...11.   gy=....11   z=.1111.   g=.1111.
# 31 ->    x=...1..   y=....11 = ( 4,  3)   gx=...11.   gy=....1.   z=.111..   g=.111..
```

Figure 5.39: Generation of a Gray code via the Hilbert curve.

```
Number of symbols = 3
Start: x
Rules:
  x --> A
  A --> Bx
  B --> Bx
--------------
0:   (#=1)
  x
1:   (#=1)
  A
2:   (#=2)
  Bx
3:   (#=3)
  BxA
4:   (#=5)
  BxABx
5:   (#=8)
  BxABxBxA
6:   (#=13)
  BxABxBxABxABx
7:   (#=21)
  BxABxBxABxABxBxABxBxA
8:   (#=34)
  BxABxBxABxABxBxABxBxABxABxBxABxABx
9:   (#=55)
  BxABxBxABxABxBxABxBxABxABxBxABxABxBxABxBxABxABxBxABxBxA
```

Figure 5.40: Rules, axiom, and first evolution steps for a specific substitution engine.

with a single x. The observed evolution is shown in figure 5.40 on the preceding page. It is trivial to implement a specific system, for the above:

```
void fx(ulong n);
void fA(ulong n);
void fB(ulong n);

void fx(ulong n)
{
    if ( 0==n ) { cout << "x"; return; }
    fA(n-1);
}

void fA(ulong n)
{
    if ( 0==n ) { cout << "A"; return; }
    fB(n-1);  fx(n-1);
}

void fB(ulong n)
{
    if ( 0==n ) { cout << "B"; return; }
    fB(n-1);  fx(n-1);
}

int main()
{
    ulong n = 10;  // max depth

    for (ulong k=0; k<=n; ++k)
    {
        fx(k);
        cout << endl;
    }

    return 0;
}
```

A utility class that allows to create string substitution engines at run time is given in [FXT: `class string_subst` in comb/stringsubst.h]:

```
class string_subst
{
public:
    // example values generate rabbit sequence:
    ulong nsym_;    // # of symbols
    // == 2
    char *symbol_;   // alphabet
    // == { '0', '1' };
    char **symrule_; // symrule_[i] string to replace i-th symbol with
    // == { "0", "1",  "1", "10" };  for  0 |-> 1,  1 |-> 10

    ulong xlate_[256];  // translate char --> rule
    //  'a' --> symrule[xlate_['a']]

    ulong cmax_; // max string length
    char  *cc_;  // string to hold result
    ulong ctc_;  // count chars of result actually produced

public:
    string_subst(int cmax, int nsym, char** symrule);
```

Rules and symbols have to be supplied on construction. The member function `subst` takes an axiom and the number of generations as arguments and returns the numbers of produced characters. It calls the recursive function `do_subst`:

```
    void do_subst(ulong n, const char *rule)
    {
        if ( 0==n )  // add symbols to string:
        {
            for (ulong i=0;  ctc_<cmax_;  ++i)
            {
                char c = rule[i];
                if ( 0==c )  break;
                cc_[ctc_] = c;
                ++ctc_;
            }
        }
        else  // recurse:
```

```
        {
            for (ulong i=0;  0!=rule[i];  ++i)
            {
                ulong r = (ulong)rule[i];
                ulong x = xlate_[r];
                do_subst(n-1, symrule_[x]);  // recursion
            }
        }
    }

    ulong subst(ulong maxn, const char *start)
    // maxn:=number of generations,  start:=axiom
    {
        ctc_ = 0;
        do_subst(maxn, start);
        cc_[ctc_] = 0;  // terminate string
        return  ctc_;
    }
```

The resulting string is stored in an array of characters that can be accessed via the member function `string()`. The characters that do not fit into the array are silently discarded. Using an array of one million characters, the rules and axiom as follows

```
Number of symbols = 5
Start: a
Rules:
  a --> -bF+aFa+Fb-
  b --> +aF-bFb-Fa+
  + --> +
  - --> -
  F --> F
```

Then one obtains (displaying only the string lengths from the fourth generation on):

```
 0:   (#=1)
   a
 1:   (#=11)
   -bF+aFa+Fb-
 2:   (#=51)
   -+aF-bFb-Fa+F+-bF+aFa+Fb-F-bF+aFa+Fb-+F+aF-bFb-Fa+-
 3:   (#=211)
   -+-bF+aFa+Fb-F-+aF-bFb-Fa+F+aF-bFb-Fa+-F-bF+aFa+Fb-+F+-+aF-bFb-  \
    Fa+F+-bF+aFa+Fb-F-bF+aFa+Fb-+F+aF-bFb-Fa+-F-+aF-bFb-Fa+F+-bF+a  \
    Fa+Fb-F-bF+aFa+Fb-+F+aF-bFb-Fa+-+F+-bF+aFa+Fb-F-+aF-bFb-Fa+F+a  \
    F-bFb-Fa+-F-bF+aFa+Fb-+-
 4:   (#=851)
 5:   (#=3411)
 6:   (#=13651)
 7:   (#=54611)
 8:   (#=218451)
 9:   (#=873811)
10:   (#=1000000)  // <-- truncation
11:   (#=1000000)
```

The computation takes a split second only. The strings generated by the shown rule correspond to the Hilbert curves in successively finer resolutions if one initializes

```
position:         (x, y)   := (0,0)
direction:        (dx, dy) := (1,0)
```

and identifies:

```
    '-'  ==  "turn left":   { t= dx; dx=-dy; dy=t; }
    '+'  ==  "turn right":  { t=-dx; dx= dy; dy=t; }
    'F'  ==  "advance":     { line(x, y, x+dx, y+dy);  x+=dx; y+=dy; }
    either 'a' or 'b'  ==   (ignore)
```

The demo code in [FXT: demo ds/stringsubst-demo.cc] prints evolutions for several rules.

The number of symbols that occur in each generation can be computed by a simple matrix power. With i different symbols let $n_j(k)$ be the number of occurrences of the j-th symbol at generation k. Then $n_j(0)$ is determined by the axiom and

$$\vec{N}(k) \quad := \quad [n_1(k), n_2(k), \ldots, n_i(k)]^T \tag{5.15}$$

can be computed as

$$\vec{N}(k) \quad = \quad M^k \, \vec{N}(0) \tag{5.16}$$

where M is a matrix with columns according to the occurrences of the symbols in the transfer rules. For example, with the rules and axiom

```
Number of symbols = 3
Start: x
Rules:                 x  A  B
  x --> A     1 --> [0, 1, 0]
  A --> Bx    2 --> [1, 0, 1]  =: transpose(M)
  B --> Bx    3 --> [1, 0, 1]
```

we obtain

$$M \;=\; \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \qquad M^k \;=\; \begin{bmatrix} F_{k-1} & F_k & F_k \\ F_{k-2} & F_{k-1} & F_{k-1} \\ F_{k-1} & F_k & F_k \end{bmatrix} \qquad k \geq 2 \tag{5.17}$$

where F_k is the k-th Fibonacci number. For the tenth generation, $k = 10$:

$$M^{10}\,[1,0,0]^T \;=\; \begin{bmatrix} 34 & 55 & 55 \\ 21 & 34 & 34 \\ 34 & 55 & 55 \end{bmatrix} [1,0,1]^T \;=\; [34,21,34]^T \tag{5.18}$$

So we have 34 '`x`', 21 '`A`' and 34 '`B`' or a total of 89 symbols.

For the 'Hilbert engine'

```
Number of symbols = 5
Start: a
Rules:
  a --> -bF+aFa+Fb-   1 --> [2, 2, 2, 2, 3]
  b --> +aF-bFb-Fa+   2 --> [2, 2, 2, 2, 3]
  + --> +             3 --> [0, 0, 1, 0, 0] =: transpose(m)
  - --> -             4 --> [0, 0, 0, 1, 0]
  F --> F             5 --> [0, 0, 0, 0, 1]
```

we find

```
 k     total    [   n(a),    n(b),    n(+),   n(-),    n(F) ]
 0         1    [      1,       0,       0,      0,       0 ]
 1        11    [      2,       2,       2,      2,       3 ]
 2        51    [      8,       8,      10,     10,      15 ]
 3       211    [     32,      32,      42,     42,      63 ]
 4       851    [    128,     128,     170,    170,     255 ]
 5      3411    [    512,     512,     682,    682,    1023 ]
 6     13651    [   2048,    2048,    2730,   2730,    4095 ]
 7     54611    [   8192,    8192,   10922,  10922,   16383 ]
 8    218451    [  32768,   32768,   43690,  43690,   65535 ]
 9    873811    [ 131072,  131072,  174762, 174762,  262143 ]
10   3495251    [ 524288,  524288,  699050, 699050, 1048575 ]
```

In chapter 24 on page 633 the string substitution idea is used to construct fast iterations for various functions.

The problem of the identification of a finite automaton (substitution engine) that generates a given infinite string is discussed in [73].

5.22 Necklaces and Lyndon words

A sequence that is minimal among all its cyclic rotations is called a *necklace*. When there are k possible values for each element one talks about an n-bead, k-color (or k-ary length-n) necklaces. We restrict our attention to the case were only two sorts of beads are allowed and represent them by 0 and 1.

Scanning all binary words of length n as to whether they are necklaces can easily be achieved by testing whether the word `x` is equal to the return value of the function `bit_cyclic_min(x,n)` shown in section 1.12. For n up to 8 one obtains the sequences of binary necklaces shown in figure 5.41 on the facing page. As 2^n words have to be tested this brute-force approach gets ineffective for large n. Luckily there is both a much better algorithm for generating all necklaces and a formula for their number.

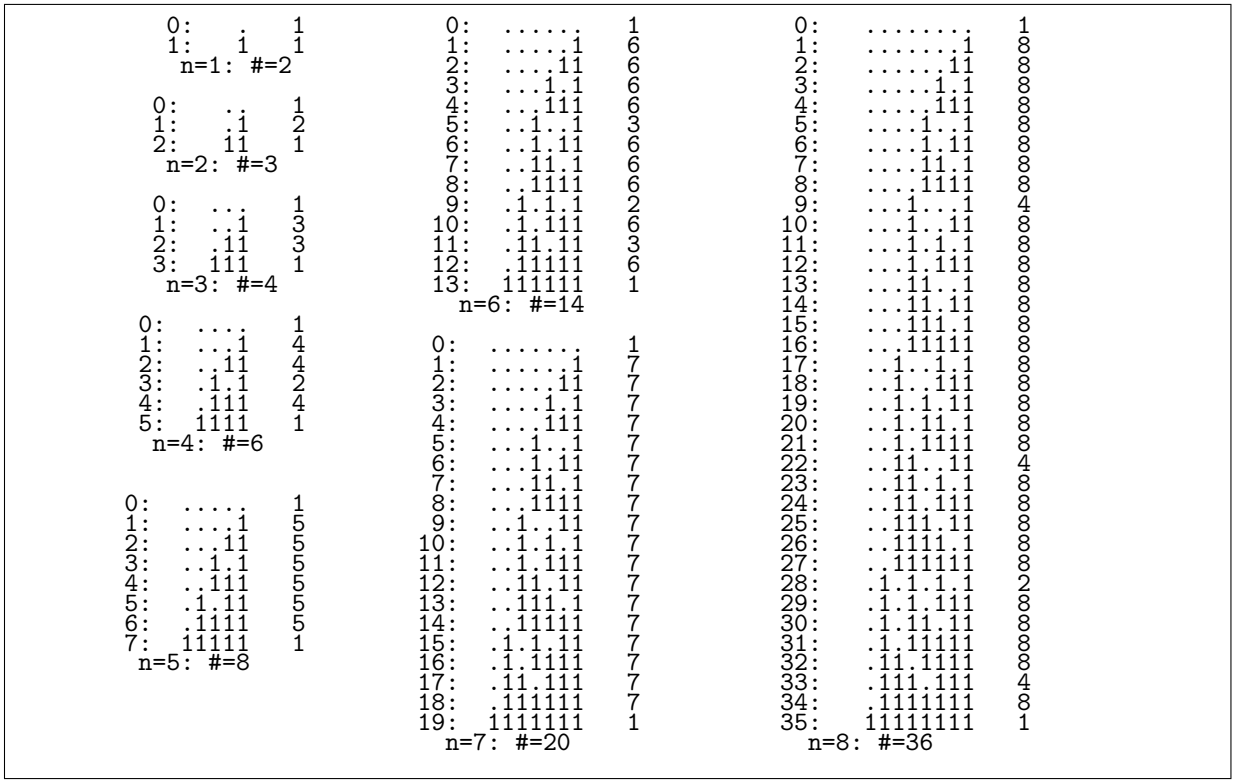


Figure 5.41: All binary necklaces of lengths up to 8 and their periods. Dots represent zeros.

Not all necklaces are created equal. Each necklace can be assigned a period that is a divisor of the length. That period is the smallest (nonzero) cyclic shift that transforms the word into itself. The periods are given directly right to each necklace in figure 5.41. For n prime the only periodic necklaces are those two that contain all ones or zeros. Aperiodic (or equivalently, period equals length) necklaces are called *Lyndon words*.

For a length-n binary word x the function `bit_cyclic_period(x,n)` from section 1.13 on page 25 returns the period of the word.

5.22.1 Generating all necklaces

The following algorithm for generation of all necklaces actually produces *pre-necklaces* a subset of which are the necklaces. A pre-necklace is a string that is the prefix of some necklace. The *FKM algorithm* (for Fredericksen, Kessler, Maiorana) to generate all k-ary length-n pre-necklaces proceeds as follows:

1. Initialize the word $F = [f_1, f_2, \ldots, f_n]$ to all zeros. Set $i = 1$.
2. (Visit pre-necklace F. If i divides n then F is a necklace. If i equals n then F is a Lyndon word.)
3. Find the largest index i so that $f_i < k - 1$. Increment f_i.
4. Fill the word F with copies of $[f_1, \ldots, f_i]$.
5. If $F = [k-1, k-1, \ldots, k-1]$ then terminate, else goto step 2.

Implementation is straightforward, the crucial steps of [FXT: demo comb/necklace-fkm-demo.cc] are

```
for (ulong j=1; j<=n; ++j)  f[j] = 0;  // Initialize to zero
bool nq = 1;  // whether pre-necklace is a necklace
bool lq = 0;  // whether pre-necklace is a Lyndon word
ulong i = 1;
```

```
 1:   . . . .   i=1  N
 2:   . . . 1   i=4  N  L
 3:   . . . 2   i=4  N  L
 4:   . . 1 .   i=3
 5:   . . 1 1   i=4  N  L
 6:   . . 1 2   i=4  N  L
 7:   . . 2 .   i=3
 8:   . . 2 1   i=4  N  L
 9:   . . 2 2   i=4  N  L
10:   . 1 . 1   i=2  N
11:   . 1 . 2   i=4  N  L
12:   . 1 1 .   i=3
13:   . 1 1 1   i=4  N  L
14:   . 1 1 2   i=4  N  L
15:   . 1 2 .   i=3
16:   . 1 2 1   i=4  N  L
17:   . 1 2 2   i=4  N  L
18:   . 2 . 2   i=2  N
19:   . 2 1 .   i=3
20:   . 2 1 1   i=4  N  L
21:   . 2 1 2   i=4  N  L
22:   . 2 2 .   i=3
23:   . 2 2 1   i=4  N  L
24:   . 2 2 2   i=4  N  L
25:   1 1 1 1   i=1  N
26:   1 1 1 2   i=4  N  L
27:   1 1 2 1   i=3
28:   1 1 2 2   i=4  N  L
29:   1 2 1 2   i=2  N
30:   1 2 2 1   i=3
31:   1 2 2 2   i=4  N  L
32:   2 2 2 2   i=1  N
32 (4, 3) pre-necklaces,
24 necklaces and 18 Lyndon words.

 1:   . . . . . .   i=1  N
 2:   . . . . . 1   i=6  N  L
 3:   . . . . 1 .   i=5
 4:   . . . . 1 1   i=6  N  L
 5:   . . . 1 . .   i=4
 6:   . . . 1 . 1   i=6  N  L
 7:   . . . 1 1 .   i=5
 8:   . . . 1 1 1   i=6  N  L
 9:   . . 1 . . 1   i=3  N
10:   . . 1 . 1 .   i=5
11:   . . 1 . 1 1   i=6  N  L
12:   . . 1 1 . .   i=4
13:   . . 1 1 . 1   i=6  N  L
14:   . . 1 1 1 .   i=5
15:   . . 1 1 1 1   i=6  N  L
16:   . 1 . 1 . 1   i=2  N
17:   . 1 . 1 1 .   i=5
18:   . 1 . 1 1 1   i=6  N  L
19:   . 1 1 . 1 1   i=3  N
20:   . 1 1 1 . 1   i=4
21:   . 1 1 1 1 .   i=5
22:   . 1 1 1 1 1   i=6  N  L
23:   1 1 1 1 1 1   i=1  N
23 (6, 2) pre-necklaces,
14 necklaces and 9 Lyndon words.
```

Figure 5.42: Ternary length-4 (left) and binary length-6 (right) pre-necklaces as generated by the FKM algorithm. Dots are used for zeros, necklaces are marked with '`N`', Lyndon with '`L`'.

```
while ( 1 )
{
    // Print necklace:
    cout << setw(3) << pct << ":  ";
    for (ulong j=1; j<=n; ++j)  cout << " " << f[j];
    cout << "   i=" << i;
    if ( nq )  cout << "  N";
    if ( lq )  cout << "  L";
    cout << endl;

    // Find largest index where we can increment:
    i = n;
    while ( f[i]==k-1 )  { --i; };
    ++f[i];

    if ( i==0 )  break;

    // Copy periodically:
    for (ulong j=1,t=i+1; t<=n; ++j,++t)  f[t] = f[j];

    nq = ( (n%i)==0 );  // necklace if i divides n
    lq = ( i==n );      // Lyndon word if i equals n
}
```

Two example runs are shown in figure 5.42. About 23 million necklaces per second are generated.

The binary necklaces that fit into a machine word can be generated via [FXT: `class bit_necklace` in bits/bitnecklace.h] which produces about 30 million necklaces per second. A demo is given in [FXT: demo bits/bitnecklace-demo.cc]. An routine that creates binary necklaces in an array is given in [FXT: file comb/binaryprimestring.h]. It can generate about 78 million necklaces per second, see [FXT: demo comb/binaryprimestring-demo.cc].

The binary necklaces of length n can be used as cycle leaders in the length-2^n zip permutation (and its inverse) that is discussed in section 2.5 on page 87. An algorithm for the generation of all irreducible binary polynomials via necklaces is described in section 22.6 on page 594.

$O(1)$ algorithm for length an exponent of a Mersenne prime

```
 0 :   a= ......1 =    1    ==   ......1
 1 :   a= .....11 =    3    ==   .....11
 2 :   a= ...1..1 =    9    ==   ...1..1
 3 :   a= ..11.11 =   27    ==   ..11.11
 4 :   a= 1.1...1 =   81    ==   ...11.1
 5 :   a= 111.1.. =  116    ==   ..111.1
 6 :   a= 1.1111. =   94    ==   .1.1111
 7 :   a= ..111.. =   28    ==   ....111
 8 :   a= 1.1.1.. =   84    ==   ..1.1.1
 9 :   a= 11111.1 =  125    ==   .111111
10 :   a= 1111..1 =  121    ==   ..11111
11 :   a= 11.11.1 =  109    ==   .11.111
12 :   a= 1..1..1 =   73    ==   ..1..11
13 :   a= 1.111.. =   92    ==   ..1.111
14 :   a= ..1.11. =   22    ==   ...1.11
15 :   a= 1....1. =   66    ==   ....1.1
16 :   a= 1...111 =   71    ==   ...1111
17 :   a= 1.1.11. =   86    ==   .1.1.11
18 :   a= ....1.. =    4    ==   ......1 <--= sequence restarts
19 :   a= ...11.. =   12    ==   .....11
20 :   a= .1..1.. =   36    ==   ...1..1
21 :   a= 11.11.. =  108    ==   ..11.11
22 :   a= 1...11. =   70    ==   ...11.1
23 :   a= 1.1..11 =   83    ==   ..111.1
24 :   a= 1111.1. =  122    ==   .1.1111
25 :   a= 111.... =  112    ==   ....111
[--snip--]
```

Figure 5.43: Generation of all (18) 7-bit Lyndon words as binary representations of the powers modulo 127 of the primitive root 3. The right column gives the cyclic minima.

The length-n Lyndon words for n an exponent of a Mersenne prime $m = 2^n - 1$ can be generated efficiently by simply going through the powers of a primitive root r of m until the second word with just one bit is reached. With $n = 7$, $m = 127$ and the primitive root $r = 3$ we get the sequence shown in figure 5.43.

CAT algorithm

A constant amortized time (CAT) algorithm to generate all k-ary length-n (pre-)necklaces is given in [35]. The crucial part of a recursive algorithm [FXT: demo comb/necklace-cat-demo.cc] is the function

```
ulong K, N;  // K-ary pre-necklaces of length N
ulong f[N];
void crsms_gen(ulong n, ulong p)
{
    if ( n > N )  visit(p);  // pre-necklace in f[1,...,N]
    else
    {
        f[n] = f[n-p];
        crsms_gen(n+1, p);
        for (ulong j=f[n-p]+1; j<K; ++j)
        {
            f[n] = j;
            crsms_gen(n+1, n);
        }
    }
}
```

After initializing the array with zeros the function must be called with both arguments equal to one. The routine generates about 70 million pre-necklaces per second.

5.22.2 The number of necklaces

The number of binary necklaces of length n is

$$N_n \quad = \quad \frac{1}{n}\sum_{d\backslash n}\varphi(d)\,2^{n/d} \quad = \quad \frac{1}{n}\sum_{j=1}^{n}2^{\gcd(j,n)} \tag{5.19}$$

Replace 2 by k to get the number for k different sorts of beads (possible values at each digit). For $n \leq 40$ the formula gives (sequence A000031 of [OEIS]):

n	N_n	n	N_n	n	N_n	n	N_n
1	2	11	188	21	99880	31	69273668
2	3	12	352	22	190746	32	134219796
3	4	13	632	23	364724	33	260301176
4	6	14	1182	24	699252	34	505294128
5	8	15	2192	25	1342184	35	981706832
6	14	16	4116	26	2581428	36	1908881900
7	20	17	7712	27	4971068	37	3714566312
8	36	18	14602	28	9587580	38	7233642930
9	60	19	27596	29	18512792	39	14096303344
10	108	20	52488	30	35792568	40	27487816992

The number of Lyndon words (aperiodic necklaces) is

$$L_n \quad = \quad \frac{1}{n}\sum_{d\backslash n}\mu(d)\,2^{n/d} \quad = \frac{1}{n}\sum_{d\backslash n}\mu(n/d)\,2^{d} \tag{5.20}$$

The function $\mu(.)$ is defined in relation 22.30 on page 597. For $n \leq 40$ one finds:

n	L_n	n	L_n	n	L_n	n	L_n
1	2	11	186	21	99858	31	69273666
2	1	12	335	22	190557	32	134215680
3	2	13	630	23	364722	33	260300986
4	3	14	1161	24	698870	34	505286415
5	6	15	2182	25	1342176	35	981706806
6	9	16	4080	26	2580795	36	1908866960
7	18	17	7710	27	4971008	37	3714566310
8	30	18	14532	28	9586395	38	7233615333
9	56	19	27594	29	18512790	39	14096302710
10	99	20	52377	30	35790267	40	27487764474

For prime $n = p$ we have $L_p = N_p - 2$ and

$$L_p \quad = \quad \frac{2^p - 2}{p} \quad = \frac{1}{p}\sum_{k=1}^{p-1}\binom{p}{k} \tag{5.21}$$

The latter form transpires that there are exactly $\binom{p}{k}/p$ Lyndon words with k ones for $1 \leq k \leq p-1$. The difference of two is due to the necklaces that consist of all zeros or ones. The number of irreducible binary polynomials (see section 22.5 on page 593) of degree-n also equals L_n. For the equivalence between necklaces and irreducible polynomials see [35].

n:	N_n	$N_{\binom{n}{0}}$	$N_{\binom{n}{1}}$	$N_{\binom{n}{2}}$	$N_{\binom{n}{3}}$	$N_{\binom{n}{4}}$	$N_{\binom{n}{5}}$	$N_{\binom{n}{6}}$	$N_{\binom{n}{7}}$	$N_{\binom{n}{8}}$	$N_{\binom{n}{9}}$	$N_{\binom{n}{10}}$
1:	2	1	1									
2:	3	1	1	1								
3:	4	1	1	1	1							
4:	6	1	1	2	1	1						
5:	8	1	1	2	2	1	1					
6:	14	1	1	3	4	3	1	1				
7:	20	1	1	3	5	5	3	1	1			
8:	36	1	1	4	7	10	7	4	1	1		
9:	60	1	1	4	10	14	14	10	4	1	1	
10:	108	1	1	5	12	22	26	22	12	5	1	1
11:	188	1	1	5	15	30	42	42	30	15	5	1
12:	352	1	1	6	19	43	66	80	66	43	19	6
13:	632	1	1	6	22	55	99	132	132	99	55	22
14:	1182	1	1	7	26	73	143	217	246	217	143	73
15:	2192	1	1	7	31	91	201	335	429	429	335	201
16:	4116	1	1	8	35	116	273	504	715	810	715	504
17:	7712	1	1	8	40	140	364	728	1144	1430	1430	1144
18:	14602	1	1	9	46	172	476	1038	1768	2438	2704	2438
19:	27596	1	1	9	51	204	612	1428	2652	3978	4862	4862
20:	52488	1	1	10	57	245	776	1944	3876	6310	8398	9252

Figure 5.44: The number of binary necklaces with a prescribed number of ones.

n:	L_n	$L_{\binom{n}{0}}$	$L_{\binom{n}{1}}$	$L_{\binom{n}{2}}$	$L_{\binom{n}{3}}$	$L_{\binom{n}{4}}$	$L_{\binom{n}{5}}$	$L_{\binom{n}{6}}$	$L_{\binom{n}{7}}$	$L_{\binom{n}{8}}$	$L_{\binom{n}{9}}$	$L_{\binom{n}{10}}$
1:	2	1	1									
2:	1	0	1	0								
3:	2	0	1	1	0							
4:	3	0	1	1	1	0						
5:	6	0	1	2	2	1	0					
6:	9	0	1	2	3	2	1	0				
7:	18	0	1	3	5	5	3	1	0			
8:	30	0	1	3	7	8	7	3	1	0		
9:	56	0	1	4	9	14	14	9	4	1	0	
10:	99	0	1	4	12	20	25	20	12	4	1	0
11:	186	0	1	5	15	30	42	42	30	15	5	1
12:	335	0	1	5	18	40	66	75	66	40	18	5
13:	630	0	1	6	22	55	99	132	132	99	55	22
14:	1161	0	1	6	26	70	143	212	245	212	143	70
15:	2182	0	1	7	30	91	200	333	429	429	333	200
16:	4080	0	1	7	35	112	273	497	715	800	715	497
17:	7710	0	1	8	40	140	364	728	1144	1430	1430	1144
18:	14532	0	1	8	45	168	476	1026	1768	2424	2700	2424
19:	27594	0	1	9	51	204	612	1428	2652	3978	4862	4862
20:	52377	0	1	9	57	240	775	1932	3876	6288	8398	9225

Figure 5.45: The number of binary Lyndon words with a prescribed number of ones.

5.22.3 The number of necklaces with fixed content

Let $N_{\binom{n}{n_0}}$ be the number of binary length-n necklaces with exactly n_0 zeros (and $n_1 = n - n_0$ ones). We call these *necklaces with fixed density*. One has

$$N_{\binom{n}{n_0}} = \frac{1}{n} \sum_{j \backslash g} \varphi(j) \binom{n/j}{d/j} \tag{5.22}$$

where $g = \gcd(n, n_0)$. One has the symmetry relation $N_{\binom{n}{n_0}} = N_{\binom{n}{n-n_0}} = N_{\binom{n}{n_1}}$. A table of small values is given in figure 5.44 on the previous page.

Let $L_{\binom{n}{n_0}}$ be the number of binary length-n Lyndon words with exactly n_0 ones (*Lyndon words with fixed density*), then

$$L_{\binom{n}{n_0}} = \frac{1}{n} \sum_{j \backslash g} \mu(j) \binom{n/j}{d/j} \tag{5.23}$$

where $g = \gcd(n, n_0)$. The symmetry relation is the same as for $N_{\binom{n}{n_0}}$. A table of small values is given in figure 5.45 on the preceding page.

Let $N(n_0, n_1, \ldots, n_{k-1})$ be the number of k-symbol length-n necklaces with n_j occurrences of symbol j. For the number of such *necklaces with fixed content* we have ($n = \sum_{j<s} n_j$ and):

$$N(n_0, n_1, \ldots, n_{k-1}) = \frac{1}{n} \sum_{d \backslash g} \varphi(d) \frac{(n/d)!}{(n_0/d)! \cdots (n_{k-1}/d)!} \tag{5.24}$$

where $g = \gcd(n_0, n_1, \ldots, n_{k-1})$. The equivalent formula for the *Lyndon words with fixed content* is

$$L(n_0, n_1, \ldots, n_{k-1}) = \frac{1}{n} \sum_{d \backslash g} \mu(d) \frac{(n/d)!}{(n_0/d)! \cdots (n_{k-1}/d)!} \tag{5.25}$$

where $g = \gcd(n_0, n_1, \ldots, n_{k-1})$. The number of strings with given content equals

$$S(n_0, n_1, \ldots, n_{k-1}) = \frac{n!}{n_0! \cdots n_{k-1}!} \tag{5.26}$$

The relations were taken from [88] and [92] which also give efficient algorithms for the generation of necklaces and Lyndon words with fixed density [88] and content [92].

Chapter 6

Shift registers

In this chapter we describe shift register sequences (SRS) and their generation via linear feedback shift registers (LFSR). The underlying mechanism is the modular arithmetic of binary polynomials described in section 22.3 on page 579. We give an expression for the number of shift registers sequences of maximal length, the so-called m-sequences. Two related mechanisms, feedback carry shift registers (FCSR) and linear hybrid cellular automata (LHCA), are described.

Most of the algorithms given can easily be implemented in hardware.

6.1 Linear feedback shift registers (LFSR)

Multiplication of a binary polynomial A by x modulo a polynomial C is particularly easy as shown near the beginning of section 22.3: Simply shift the input to the left (multiplication) and if the result $A \cdot x$ has the same degree as C then subtract (XOR) the polynomial C (modular reduction).

The underlying mechanism of shifting and conditionally feeding back certain bits is called a *linear feedback shift register* (LFSR). A *shift register sequence* (SRS) can be obtained by computing $A_k = x^k$, $k = 0, 1, \ldots, 2^n - 1$ modulo C and setting bit k of the SRS to the least significant bit of A_k.

When the modulus C is a primitive polynomial of degree n then the SRS is a sequence of zeros and ones that contains all nonzero words of length n. Further, if a word W is updated at each step by left shifting and adding the bit of the SRS then this sequence also contains all nonzero words.

This is demonstrated in [FXT: demo gf2n/lfsr-demo.cc], which for $n = 4$ uses the primitive polynomial $C = x^4 + x + 1$ and gives the output:

```
    c = 1..11 == 0x13 == 19  (deg = 4)
     0    w =   15  = 1111  1  ...1 =    1 = a
     1    w =   14  = 111.  .  ..1. =    2 = a
     2    w =   12  = 11..  .  .1.. =    4 = a
     3    w =    8  = 1...  .  1... =    8 = a
     4    w =    1  = ...1  1  ..11 =    3 = a
     5    w =    2  = ..1.  .  .11. =    6 = a
     6    w =    4  = .1..  .  11.. =   12 = a
     7    w =    9  = 1..1  1  1.11 =   11 = a
     8    w =    3  = ..11  1  .1.1 =    5 = a
     9    w =    6  = .11.  .  1.1. =   10 = a
    10    w =   13  = 11.1  1  .111 =    7 = a
    11    w =   10  = 1.1.  .  1111 =   14 = a
    12    w =    5  = .1.1  1  1111 =   15 = a
    13    w =   11  = 1.11  1  11.1 =   13 = a
    14    w =    7  = .111  1  1..1 =    9 = a
 >>  0    w =   15  = 1111  1  ...1 =    1 = a  << period restarts
     1    w =   14  = 111.  .  ..1. =    2 = a
     2    w =   12  = 11..  .  .1.. =    4 = a
     3    w =    8  = 1...  .  1... =    8 = a
     4    w =    1  = ...1  1  ..11 =    3 = a
     5    w =    2  = ..1.  .  .11. =    6 = a
```

Here we pasted the first few lines after the end of the actual output to emphasize the periodicity of the sequences. The corresponding SRS of period 15 is (extra spaces to mark start of period):

```
1 0 0 0 1 0 0 1 1 0 1 0 1 1 1    1 0 0 0 1 0 0 1 1 0 1 0 1 1 1    1 0 ...
```

In fact any of the bits of the words $A_k = x^k \bmod C$ (or linear combination of two or more bits) could be used, each producing a cyclically shifted version of the SRS.

The cheapest way to generate a SRS is to compute the negative powers x^{-k} modulo C, that is, to repeatedly divide by x:

```
ulong c = a primitive polynomial;
ulong n = degree of C;
ulong a = 1;
for (ulong k=0; k<n; ++k)
{
    ulong s = a & 1;
    // Use s here.
    if ( s )  a ^= c;
    a >> 1;
}
```

The routine will work for $\deg(C) < n$ where n is the number of bits in a machine word. A version that also works for $\deg(C) = n$ is given in section 22.3 on page 579.

A C++ class implementing a LFSR with a few convenient extras is [FXT: `class lfsr` in bpol/lfsr.h]:

```
class lfsr
// (binary) Linear Feedback Shift Register
// Produces a shift register sequence (SRS)
// generated by  a_k=x^k (mod c) where
// c is a primitive polynomial of degree n.
// The period of SRS is 2^n - 1
// (non-primitive c lead to smaller periods)
{
public:
    ulong a_;  // internal state (polynomial modulo c)
    ulong w_;  // word of the shift_register_sequence (SRS)
    ulong c_;  // (mod 2) poly  e.g. x^4+x+1 == 0x13 == 1..11
    ulong h_;  // highest bit in SRS word  e.g. (above) == 16 = 1...
    ulong mask_;  // mask  e.g. (above) == 15 == 1111
    ulong n_;  // degree of polynomial  e.g. (above) == 4
public:
    lfsr(ulong n, ulong c=0)
    // n: degree of polynomial c
    // c: polynomial (defaults to minimum weight primitive polynomial)
    {
 [--snip--]
```

The crucial computation is implemented as

```
ulong next()
{
    ulong s = a_ & h_;
    a_ <<= 1;
    w_ <<= 1;
    if ( 0!=s )
    {
        a_ ^= c_;
        w_ |= 1;
    }
    w_ &= mask_;
    return w_;
}
```

Up to the lines that update the word `w_` this function is identical to `bitpolmod_times_x()` given in section 22.3 on page 579.

The method `next_w()` skips to the next word by calling `next()` n times:

```
ulong next_w()
{
    for (ulong k=0; k<n_; ++k)  next();
```

```
    return w_;
}
```

Let a and w a pair of values that correspond to each other. The following two methods allow to directly set one of these two while keeping the pair consistent. Setting a to a given value:

```
void set_a(ulong a)
{
    a_ = a;
    w_ = 0;
    ulong b = 1;
    for (ulong j=0; j<n_; ++j)
    {
        if ( a & 1 )
        {
            w_ |= b;
            a ^= c_;
        }
        b <<= 1;
        a >>= 1;
    }
}
```

The loop executes n times where n is the degree of the modulus. Setting w to a given value:

```
void set_w(ulong w)
{
    w_ = w;
    a_ = 0;
    ulong c = c_;
    while ( w )
    {
        if ( w & 1 )  a_ ^= c;
        c <<= 1;
        w >>= 1;
    }
    a_ &= mask_;
}
```

The supplied value must be nonzero for both methods.

Going back one step is possible via the method `prev()`

```
public:
    ulong prev()
    {
        prev_a();
        set_a(a_);
        return w_;
    }
```

which calls `prev_a()`:

```
private:
    void prev_a()
    {
        ulong s = a_ & 1;
        a_ >>= 1;
        if ( s )
        {
            a_ ^= (c_>>1);
            a_ |= h_;  // so it works for  n_ == BITS_PER_LONG
        }
    }
```

The method `prev_a()` leaves the value of w inconsistent with a and cannot be called directly. Note that stepping back is more expensive than stepping forward because `set_a(.)` is rather expensive.

It is also possible to go backwards word-wise:

```
    ulong prev_w()
    {
        for (ulong k=0; k<n_; ++k)  prev_a();
        set_a(a_);
        return w_;
    }
```

As this routine involves only one call to `set_a(.)` it is about as expensive as stepping one word forward using `next_w()`.

6.2 Galois and Fibonacci setup

The type of shift registers considered so far is the so-called *Galois setup* of a binary shift register. The mechanism is to detect whether a one is being shifted out and, if so, subtract the polynomial modulus. The left- and right- shift operations can be implemented as

```
ulong galois_left(ulong x, ulong c, ulong h)
{
    ulong s = x & h;
    x <<= 1;
    if ( 0!=s )  x ^= c;
    return x;
}
```

and

```
ulong galois_right(ulong x, ulong c, ulong)
{
    ulong s =  ( x & 1UL );
    x >>= 1;
    if ( s )  x ^= (c>>1);
    return x;
}
```

```
    c = .11..1 == 0x19 == 25  (deg = 4)
    r = .1..11 == 0x13 == 19  (deg = 4)

          -------- Galois -------------          --------- Fibonacci ---------
     k:     Lc      Lr      Rc      Rr            Lc      Lr      Rc      Rr
     1:    ...1    ...1    ...1    ...1          ...1    ...1    ...1    ...1
     2:    ..1.    ..1.    .11.    .1.1          ..1.    ..11    .1..    .1..
     3:    .1..    .1..    ..11    .11.          .1..    .111    .11.    ..1.
     4:    1...    1...    ...1    1111          1..1    1111    111.    ...1
     5:    1..1    ..11    .11.    111.          ..11    111.    1111    1..1
     6:    1.11    .11.    1.1.    .111          .11.    11.1    .111    11..
     7:    1111    11..    ..1.    1.1.          11.1    1.1.    1.11    .11.
     8:    .111    1.11    111.    ..1.          1.1.    .1.1    .1.1    1.11
     9:    111.    .1.1    .111    1.11          .1.1    1.11    1.1.    .1.1
    10:    .1.1    1.1.    1111    11..          1.11    .11.    11.1    1.1.
    11:    1.1.    .111    1.11    ..11          .111    11..    .11.    11.1
    12:    11.1    111.    1..1    ...1          1111    1..1    ...1    111.
    13:    ..11    1111    1...    1...          111.    ...1    1..1    1111
    14:    .11.    11.1    .1..    .1..          11..    ..1.    .1..    .111
    15:    11..    1..1    ...1    ...1          1...    1...    ...1    ...1
    16:    ...1    ...1    ...1    ...1          ...1    ...1    ...1    ...1
```

Figure 6.1: Sequences of words generated with the Galois- and Fibonacci- mechanisms, either with the left- or the right- shift (capital letters '`L`' and '`R`' on top of columns) and primitive polynomial '`c`' or its reciprocal '`r`'. Each track of any sequence is a shift register sequence.

Four sequences of binary words that (starting identically and) are generated with either the left- or right-shift and a primitive polynomial or its reciprocal (reversed word) are shown in figure 6.1. A different set of sequences shown in the same figure is obtained via the so-called *Fibonacci setup*. In the Fibonacci setup the sum (modulo 2) of bits determined by the used polynomial is shifted in at each step.

The left- and right- shift operations can be implemented as

```
ulong fibonacci_left(ulong x, ulong c, ulong h)
{
    x <<= 1;
    ulong s = parity( x & c );
    if ( 0!=s )  x ^= 1;
    x &= ~(h<<1);  // remove excess bit at high end
    return x;
}
```

and

```
ulong fibonacci_right(ulong x, ulong c, ulong h)
{
    ulong s = parity( x & c );
    x >>= 1;
    if ( s )  x ^= (h);
    return x;
}
```

As the parity computation is expensive on most machines (see section 1.16.1 on page 34) the Galois setup should usually be preferred. The programs [FXT: demo gf2n/lfsr-fibonacci-demo.cc] and [FXT: demo gf2n/lfsr-galois-demo.cc] can be used to create the binary patterns shown in figure 6.1 on the facing page. Both allow to specify the polynomial modulus.

6.3 The number of m-sequences and De Bruijn sequences

The shift register sequences generated with a polynomial degree n is of maximal length if the polynomial is primitive. The corresponding shift register sequences are called *m-sequences*.

We now consider all sequences that are cyclic shifts of each other as the same sequence. For given n there are as many m-sequences as primitive polynomials ($P_n = \varphi(2^n - 1)/n$, see section 22.5 on page 593). These can be generated using the linear feedback shift registers described in section 6.1 on page 217.

One might suspect that the using the powers of other elements than x might lead to additional m-sequences, but this is not the case. Further, the powers of elements of maximal order modulo irreducible non-primitive polynomials do not give additional m-sequences.

The program [FXT: demo gf2n/all-primpoly-srs-demo.cc] computes all m-sequences for a given n. The output for $n = 2, 3, 4, 5, 6$ is shown in figure 6.2.

```
degree = 2
c=111 :  .11

degree = 3
c=1.11 :  ..1.111
c=11.1 :  ..111.1

degree = 4
c=1..11 :  ...1..11.1.1111
c=11..1 :  ...1111.1.11..1

degree = 5
c=1..1.1 :  ....1..1.11..11111...11.111.1.1
c=1111.1 :  ....11..1..11111.111...1.1.11.1
c=11.111 :  ....111..11.11111.1...1..1.1.11
c=1.1111 :  ....1.11.1.1...111.11111..1..11
c=111.11 :  ....11.1.1..1...1.11111.11..111
c=1.1..1 :  ....1.1.111.11...11111..11.1..1

degree = 6
c=1....11 :  .....1....11...1.1..1111.1...111..1..1.11.111.11..11.1.1.111111
c=11..111 :  .....1111..1..1.1.1..11.1....1...1.11.111111.1.111...11..111.11
c=1.11.11 :  .....1.111111..1.1.1...11..1111.111.1.11.1..11.11...1..1....111
c=11.11.1 :  .....111....1..1...11.11..1.11.1.111.1111..11...1.1.1..111111.1
c=111..11 :  .....11.111..11...111.1.111111.11.1...1....1.11..1.1.1..1..1111
c=11....1 :  .....111111.1.1.11..11.111.11.1..1..111...1.1111..1.1...11....1
```

Figure 6.2: All m-sequences for $n = 2$, 3, 4, 5, and 6. Dots denote zeros.

When a zero is inserted to the (unique) run of $n - 1$ zeros in an m-sequence then a *De Bruijn sequence* (DBS) is obtained. A DBS contains all binary words including the all-zero word.

For all $n \geq 4$ given there exist more DBSs than m-sequences. For example, for $n = 6$ there are 6 m-sequences and 67,108,864 DBSs. An brute force search for all DBSs of given length $L = 2^n$ is possible only for tiny n. The program [FXT: demo bits/all-dbs-demo.cc] finds all DBSs for $n = 3, 4, 5$. Its output with $n = 4$ and $n = 5$ (partly) is shown in figure 6.3 on the following page.

```
      n = 4  nn = 16            n = 5  nn = 32
      ....1111.11..1.1          .....11111.111..11.1.11...1.1..1
      ....1111.1.11..1          .....11111.111..11.1.11...1..1.1
      ....1111.1..1.11          .....11111.111..11.1.1..1.11...1
      ....1111..1.11.1          .....11111.111..11.1.1..1...1.11
      ....11.1111..1.1          .....11111.111..11.1.1...1..1.11
      ....11.1.1111..1          .....11111.111..11.1..1.1.11...1
      ....11.1..1.1111          .....11111.111..11.1..1...1.1.11
      ....11..1.1111.1           [--snip--]
      ....1.1111.1..11          .....1...11..1.1.11.111.1..11111
      ....1.1111..11.1          .....1...11..1.1.11.1..11111.111
      ....1.11.1..1111          .....1...11..1.1.11.1..111.11111
      ....1.11..1111.1          .....1...11..1.1..11111.11.1.111
      ....1.1..1111.11          .....1...11..1.1..11111.1.11.111
      ....1.1..11.1111          .....1...11..1.1..111.11.1.11111
      ....1..1111.1.11          .....1...11..1.1..111.1.11.11111
      ....1..11.1.1111          total # of DBS found = 2048
   total # of DBS found = 16
```

Figure 6.3: Left: All De Bruijn sequences for $n = 4$. The first and last sequence correspond to the two m-sequences for $n = 4$. Right: The first and last few De Bruijn sequences for $n = 5$.

The total number of DBSs equals

$$S_n \quad = \quad 2^x \quad \text{where} \quad x = 2^{n-1} - n \tag{6.1}$$

The two DBSs for $n = 3$ are `[...111.1]` and `[...1.111]`=`[1.111...]`, reversed sequences are considered different by the formula. The first few values of S_n are

n	$L_n = 2^n$	$x = 2^{n-1} - n$	$S_n = 2^x$
1	2	0	1
2	4	0	1
3	8	1	2
4	16	4	16
5	32	11	2,048
6	64	26	67,108,864
7	128	57	144,115,188,075,855,872
8	256	120	1,329,227,995,784,915,872,903,807,060,280,344,576

The second column gives the length of the DBSs. One has $S_{n+1} = S_n^2 \, L_{n-1}$, equivalently $x_{n+1} = 2\,x_n + n - 1$.

The general formula for the number of length-n base-m DBSs is $S_n = m!^{m^{n-1}}/m^n$, as given in [DEK4]. A graph theoretical proof of the formula for the case $m = 2$ can be found in [CComb], p.56. For a more efficient approach to generate all DBS of given length see section 7.2.2 on page 237.

6.4 Auto correlation of m-sequences

We have seen that a De Bruijn sequence (DBS) can be obtained from an m-sequence by inserting a single zero at the longest run of zeros. In the other direction, if we take a DBS and delete a zero from the longest run of zeros then we obtain a sequence of length $N = 2^n - 1$ that contains every n-bit nonzero word. But these sequences are *not* m-sequences in general: Most of them can not be obtained with an n-bit LFSR and miss an important property of m-sequences.

For a sequence M of $N - 1$ zeros and ones M_k define the sequence S of elements S_k via

$$S_k \quad := \quad \begin{cases} +1 & \text{if} \quad M_k = 1 \\ -1 & \text{else} \end{cases} \tag{6.2}$$

Then, if M is an length-L m-sequence, we have for the cyclic auto correlation (or *auto correlation function*,

```
 1:  [ +-++--+-+----+++ ]   C= +15 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 2:  [ +-++----+-+--+++ ]   C= +15 -1 -1 -1 -1 +3 -5 -1 -1 -5 +3 -1 -1 -1 -1
 3:  [ +-+-++--+----+++ ]   C= +15 -1 -1 -1 -1 -1 +3 -5 -5 +3 -1 -1 -1 -1 -1
 4:  [ +-+-++----+--+++ ]   C= +15 -1 -1 -1 -1 +3 -1 -5 -5 -1 +3 -1 -1 -1 -1
 5:  [ +-+--++----+-+++ ]   C= +15 -1 -1 -1 -1 -5 +3 -1 -1 +3 -5 -1 -1 -1 -1
 6:  [ +-+--+-++----+++ ]   C= +15 -1 -1 -1 -9 -1 +3 +3 +3 +3 -1 -9 -1 -1 -1
 7:  [ +-+----++--+-+++ ]   C= +15 -1 -1 -1 -1 -5 +3 -1 -1 +3 -5 -1 -1 -1 -1
 8:  [ +-+----+-++--+++ ]   C= +15 -1 -1 -1 -1 -1 -5 +3 +3 -5 -1 -1 -1 -1 -1
 9:  [ +--++-+----+-+++ ]   C= +15 -1 -1 -1 -1 -1 -5 +3 +3 -5 -1 -1 -1 -1 -1
10:  [ +--+-++-+----+++ ]   C= +15 -1 -1 -1 -9 +3 -1 +3 +3 -1 +3 -9 -1 -1 -1
11:  [ +--+-+----++-+++ ]   C= +15 -1 -1 -1 -1 +3 -5 -1 -1 -5 +3 -1 -1 -1 -1
12:  [ +--+----++-+-+++ ]   C= +15 -1 -1 -1 -1 +3 -1 -5 -5 -1 +3 -1 -1 -1 -1
13:  [ +----++-+--+-+++ ]   C= +15 -1 -1 -1 -9 -1 +3 +3 +3 +3 -1 -9 -1 -1 -1
14:  [ +----+-++-+--+++ ]   C= +15 -1 -1 -1 -9 +3 -1 +3 +3 -1 +3 -9 -1 -1 -1
15:  [ +----+-+--++-+++ ]   C= +15 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
16:  [ +----+--++-+-+++ ]   C= +15 -1 -1 -1 -1 -1 +3 -5 -5 +3 -1 -1 -1 -1 -1
```

Figure 6.4: Cyclic auto correlations for all truncated (signed) De Bruijn sequences for $n = 4$. Only two (the first and the second last) out of the 16 sequences are m-sequences.

ACF) of S

$$C_\tau \;:=\; \sum_{k=0}^{L-1} S_k\, S_{k+\tau \bmod L} \;=\; \begin{cases} L & \text{if} \quad \tau = 0 \\ -1 & \text{else} \end{cases} \tag{6.3}$$

where $L = N - 1$ and $N = 2^n$. That is, C_0 equals the length of the sequence, all other entries are of minimal absolute value: they cannot be zero because L is odd.

This property does not hold for most of the 'truncated' DBS (where one zero in the single run of n consecutive zeros is removed). Figure 6.4 shows all (signed) truncated DBS for $n = 4$ and their auto correlations. Only 2 out of the 16 truncated DBS have an auto correlation satisfying relation 6.3, these are exactly the m-sequences for $n = 4$.

For odd primes q one can obtain sequences of length $L = q$ whose ACF satisfies

$$\sum_{k=0}^{L-1} S_k\, S_{k+\tau \bmod L} \;=\; \begin{cases} L-1 & \text{if} \quad \tau = 0 \\ -1 & \text{else} \end{cases} \tag{6.4}$$

However, the sequences start with a single zero: Set $S_0 = 0$ and for $1 \le k < q$ set $S_k = 1$ if k is s square modulo q, else $S_k = -1$. The first three such sequences for primes of the form $4\,k + 1$ and their ACFs are:

```
 5:  S: [0, +1, -1, -1, +1]
     C: [4, -1, -1, -1, -1]

13:  S: [ 0, +1, -1, +1, +1, -1, -1, -1, -1, +1, +1, -1, +1]
     C: [12, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

17:  S: [ 0, +1, +1, -1, +1, -1, -1, -1, +1, +1, -1, -1, -1, +1, -1, +1, +1]
     C: [16, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

With primes of the form $q = 4\,k + 3$ one can construct a sequence of length $L = q$ satisfying relation 6.3 by simply setting $S_0 = 1$ ($S_0 = -1$ also works) in the sequence just constructed. The sequences for the first three primes $q = 4\,k + 3$ and their ACFs are:

```
 3:  S: [+1, +1, -1]
     C: [ 3, -1, -1]

 7:  S: [+1, +1, +1, -1, +1, -1, -1]
     C: [ 7, -1, -1, -1, -1, -1, -1]

11:  S: [+1, +1, -1, +1, +1, +1, -1, -1, -1, +1, -1]
     C: [11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

These sequences can be used for the construction of Hadamard matrices, see section 10.12 on page 334.

6.5 Feedback carry shift register (FCSR)

There is a nice analogue of the LFSR in the modulo world, the *feedback carry shift register* (FCSR). With the LFSR we needed an irreducible ('prime') polynomial C where x has maximal order. The powers of x modulo C did run through all different (nonzero) words. Now take a prime c where 2 has maximal order (that is, 2 is primitive root modulo c, see section 21.5 on page 529). Then the powers of 2 modulo c run through all nonzero values less than c.

The crucial part of the implementation [FXT: `class fcsr` in bpol/fcsr.h] is

```
ulong next()
{
    a_ <<= 1;
    if ( a_ > c_ )  a_ -= c_;

    w_ <<= 1;
    ulong s = a_ & 1;
    w_ |= s;

    w_ &= mask_;
    a_ &= mask_;
    return w_;
}
```

The routine is corresponds to the so-called Galois setup, described (for the LFSR) in section 6.2 on page 220. For a comparison of Galois- and Fibonacci- setup see section 6.2 on page 220 and the paper [48].

```
 c = 1..1.1  = 37

 0 :   a= ....1 =    1    w= .1..11 =   19
 1 :   a= ...1. =    2    w= 1..11. =   38
 2 :   a= ..1.. =    4    w= ..11.. =   12
 3 :   a= .1... =    8    w= .11... =   24
 4 :   a= 1.... =   16    w= 11.... =   48
 5 :   a= 1.1.1 =   32    w= 1....1 =   32
 6 :   a= .11.11 =  27    w= ....1. =    1
 7 :   a= .1..1 =   17    w= ...11. =    3
 8 :   a= 1...1. =  34    w= ..11.. =    6
 9 :   a= .1111. =  31    w= .11.1. =   13
10 :   a= .11..1 =  25    w= .11.11 =   27
11 :   a= ..11.1 =  13    w= 11.111 =   55
12 :   a= .11.1. =  26    w= 1.111. =   46
13 :   a= ..1111 =  15    w= .111.1 =   29
14 :   a= .1111. =  30    w= 111.1. =   58
15 :   a= .1.111 =  23    w= 11.1.1 =   53
16 :   a= .1...1 =   9    w= 1.1.11 =   43
17 :   a= .1..1. =  18    w= .1.11. =   22
18 :   a= 1..1.. =  36    w= 1.11.. =   44
19 :   a= 1...11 =  35    w= .11..1 =   25
20 :   a= 1....1 =  33    w= 11..11 =   51
21 :   a= .111.1 =  29    w= 1..111 =   39
22 :   a= .1.1.1 =  21    w= ..1111 =   15
23 :   a= ...1.1 =   5    w= .11111 =   31
24 :   a= .1.1.. =  10    w= 11111. =   62
25 :   a= .1.1.. =  20    w= 1111.. =   60
26 :   a= ...11 =    3    w= 111..1 =   57
27 :   a= ..11. =    6    w= 11..1. =   50
28 :   a= .11.. =   12    w= 1..1.. =   36
29 :   a= .11... =  24    w= ..1... =    8
30 :   a= .1.11 =   11    w= .1...1 =   17
31 :   a= .1.11. =  22    w= 1...1. =   34
32 :   a= ..111 =    7    w= ...1.1 =    5
33 :   a= ..111. =  14    w= ..1.1. =   10
34 :   a= .111.. =  28    w= .1.1.. =   20
35 :   a= .1..11 =  19    w= 1.1..1 =   41

36 :   a= .....1 =   1    w= .1..11 =   19 <-- period restarts
37 :   a= ...1. =    2    w= 1..11. =   38
38 :   a= ..1.. =    4    w= ..11.. =   12
```

Figure 6.5: Successive states of a FCSR with modulus $c = 37$.

Figure 6.5 shows the successive states of a FCSR with modulus $c = 37$. This is the output of [FXT: demo gf2n/fcsr-demo.cc]. Note that w does not run through all values smaller than c but through a subset of $c - 1$ distinct values smaller than 2^6.

A list of all primes less than 2048 for which 2 is a primitive root is shoen in figure 6.6 on the facing page. The shown sequence is entry A001122 of [OEIS].

For further information on the correspondence between LFSR and FCSR see [68].

```
     x:  p prime with 2 a primitive root, 2**x < p < 2**(x+1)
     1:  3
     2:  5
     3:  11 13
     4:  19 29
     5:  37 53 59 61
     6:  67 83 101 107
     7:  131 139 149 163 173 179 181 197 211 227
     8:  269 293 317 347 349 373 379 389 419 421 443 461 467 491 509
     9:  523 541 547 557 563 587 613 619 653 659 661 677 701 709 757
         773 787 797 821 827 829 853 859 877 883 907 941 947 1019
    10:  1061 1091 1109 1117 1123 1171 1187 1213 1229 1237 1259 1277
         1283 1291 1301 1307 1373 1381 1427 1451 1453 1483 1493 1499
         1523 1531 1549 1571 1619 1621 1637 1667 1669 1693 1733 1741
         1747 1787 1861 1867 1877 1901 1907 1931 1949 1973 1979 1987
         1997 2027 2029
```

Figure 6.6: List of primes $p < 2048$ where 2 is s primitve root.

6.6 Linear hybrid cellular automata (LHCA)

Linear hybrid cellular automata (LHCA) are 1-dimensional cellular automata (with 0 and 1 the only possible states) where two different rules are applied dependent of the position (therefore the 'hybrid' in the name). An implementation in C is [FXT: `lhca_next()` in bpol/lhca.h]:

```
inline ulong lhca_next(ulong x, ulong r, ulong m)
// LHCA := (1-dim) Linear Hybrid Cellular Automaton.
// Return next state (after x) of the LHCA with
// rule (defined by) r:
//   Rule 150 is applied for cells where r is one, rule 90 else.
//   Rule 150 :=  next(x) = x + leftbit(x) + rightbit(x)
//   Rule 90  :=  next(x) = leftbit(x) + rightbit(x)
// Length defined by m:
//   m has to be a burst of the n lowest bits (n: length of automaton)
{
    r &= x;
    ulong t = (x>>1) ^ (x<<1);
    t ^= r;
    t &= m;
    return  t;
}
```

Note that the routine is branch free. Implementation in hardware is trivial as only neighboring bits interact.

The naming convention for the rules is as follows: draw a table of the eight possible states of a cell together with its neighbors then draw the new states below:

```
    XXX  XX0  X0X  X00  0XX  0X0  00X  000
     0    X    0    X    X    0    X    0
```

Now read the lower row as a binary number, the result equals $01011010_2 = 90$, so this is rule 90. Rule 150 corresponds to $10010110_2 = 150$:

```
    XXX  XX0  X0X  X00  0XX  0X0  00X  000
     X    0    0    X    0    X    X    0
```

A run of successive values for the length-16 weight-2 rule $r = 4001_{16}$ starting with 1 is shown on the left side of figure 6.7 on the next page.

It turns out that for certain r the successive states x have the maximal period $m = 2^n - 1$, all nonzero values occur. This is demonstrated in [FXT: demo gf2n/lhca-demo.cc], which for $n = 5$ and rule $r = 1$ gives the output shown in the middle of figure 6.7 on the following page.

Rule sets (with minimal weight r) that lead to maximal period are given in [34]. The list [FXT: `minweight_lhca_rule()` in bpol/minweightlhcarule.cc] was generated from that source:

```
#define R1(n,s1)     (1UL<<s1)
#define R2(n,s1,s2)  (1UL<<s1) | (1UL<<s2)
extern const ulong minweight_lhca_rule[]=
// LHCA rules of minimum weight that lead to maximal period.
{
      0,  // (empty)
```

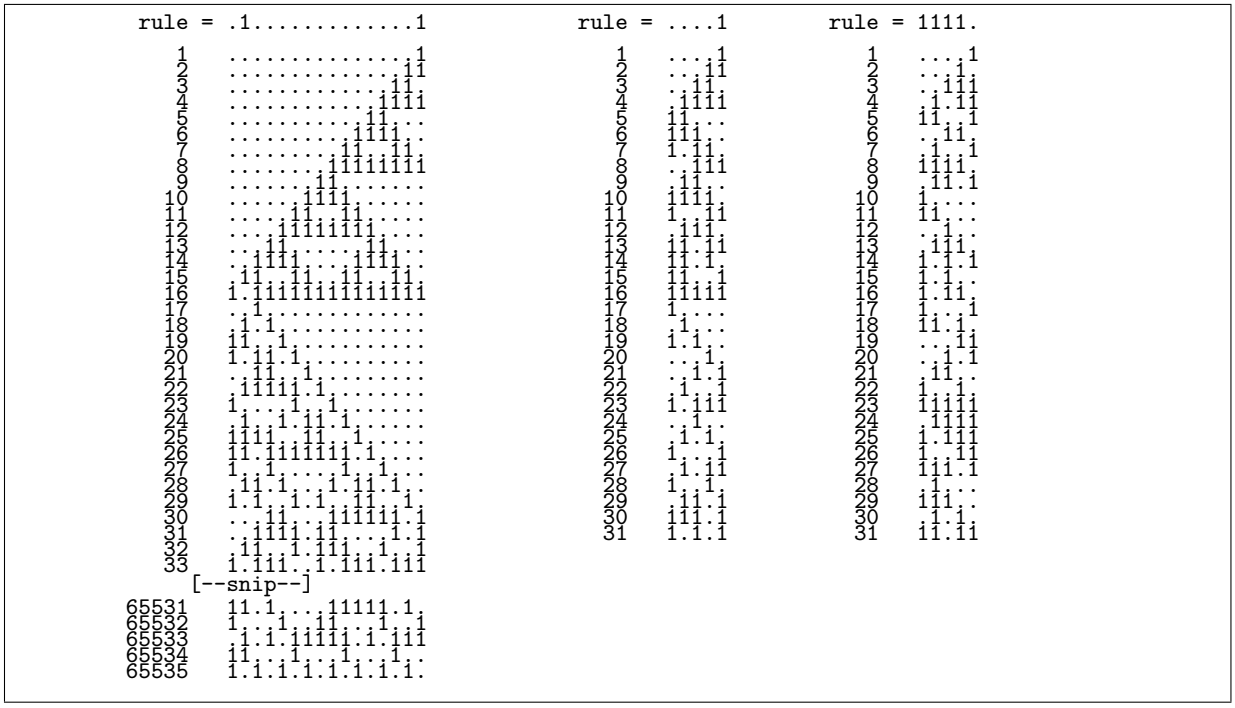


Figure 6.7: Partial run of a 16-bit LHCA (left) and complete runs of two 5-bit LHCAs (middle and right). These LHCAs have maximal periods.

```
 R1(  1,  0),
 R1(  2,  0),
 R1(  3,  0),
 R2(  4,  0, 2),
 R1(  5,  0),
 R1(  6,  0),
 R1(  7,  2),
 R2(  8,  1, 2),
 R1(  9,  0),
 R2( 10,  1, 6),
 R1( 11,  0),
 R2( 12,  2, 6),
 R1( 13,  4),
 [--snip--]
};
```

Up to $n = 500$ there is always a rule with weight at most 2 that leads to the maximal period. The full list of these rules is given in [FXT: file data/minweight-lhca-rules.txt].

6.6.1 Conversion to binary polynomials

To convert a length-n LHCA to a binary polynomial proceed as follows: Initialize $p_{-1} = 0$, $p_0 = 1$ and iterate for $k = 1, 2, \ldots, n$:

$$p_k \quad = \quad (x + r_{k-1})\, p_{k-1} + p_{k-2} \tag{6.5}$$

where r_i denotes bit i of rule r. The degree of the returned polynomial p_n is n. An implementation of the algorithm is [FXT: `lhca2poly()` in bpol/lhca.h]:

```
inline ulong lhca2poly(ulong r, ulong n)
// Return binary polynomial p that corresponds
//  to the length-n LHCA rule r.
// p has degree n.
{
```

```
    ulong p2 = 0,  p1 = 1;
    while ( n-- )
    {
        ulong m = r & 1;
        r >>= 1;
        ulong p = (p1<<1) ^ p2;
        if ( m )  p ^= p1;
        p2 = p1;  p1 = p;
    }
    return  p1;
}
```

```
                        LHCA rule                          polynomial
 1:    r=...............................1  c=.............................11  r= [0]
 2:    r=...............................1  c=............................111  r= [0]
 3:    r=...............................1  c=...........................11.1  r= [0]
 4:    r=..............................1.1  c=...........................1..11  r= [0, 2]
 5:    r=...............................1  c=..........................11.111  r= [0]
 6:    r=...............................1  c=.........................111..11  r= [0]
 7:    r=.............................1..  c=........................11.....1  r= [2]
 8:    r=.............................11.  c=.......................1...111.1  r= [1, 2]
 9:    r=...............................1  c=......................11.111..11  r= [0]
10:    r=.........................1....1.  c=.....................1...11.1111  r= [1, 6]
11:    r=.............................1  c=....................11.1...111.1  r= [0]
12:    r=.........................1...1..  c=...................1.....11111.1  r= [2, 6]
13:    r=.............................1....  c=.................11..11.....111  r= [4]
14:    r=...............................1  c=................111..11......11  r= [0]
15:    r=.............................1..  c=................11.....1...1...1  r= [2]
16:    r=.................1.............1  c=..............1..11.111.....111  r= [0, 14]
17:    r=...........................1....  c=.............11..11..11.1....11  r= [4]
18:    r=..............1..............1  c=............1..1..1.1.1.....1.1  r= [0, 16]
19:    r=.............................1..  c=............11.....111.1....11.1  r= [2]
20:    r=.............................11.  c=...........1...111.....1...1...1  r= [1, 2]
21:    r=......................1........1  c=..........1.11..1....1..1.1.1111  r= [0, 9]
22:    r=.............................1....  c=.........1111..11...1.11.1111.11  r= [4]
23:    r=...............................1  c=........11.1...1.......111.1...1  r= [0]
24:    r=...................1...1.......  c=.......1.....1.1...1.1..11111..1  r= [7, 11]
25:    r=.......................1........  c=......11..11..11.........111..11  r= [8]
26:    r=...............................1  c=.....111..11.111..........11.111  r= [0]
27:    r=............1..................1  c=....1.111..1..111....11.1..111.1  r= [0, 19]
28:    r=.............................1..  c=...111111.1.1111........1.1.1111  r= [2]
29:    r=...............................1  c=..11.111.....111.............111  r= [0]
30:    r=...............................1  c=.111..11......11.............11  r= [0]
31:    r=......................1.........  c=11.....................1...1...1  r= [10]
```

Figure 6.8: Minimum-weight LHCA rules and the corresponding binary polynomials.

The first minimum-weight LHCA rules and their binary polynomials are shown in figure 6.8. The table was created by the demo [FXT: demo gf2n/lhca2poly-demo.cc].

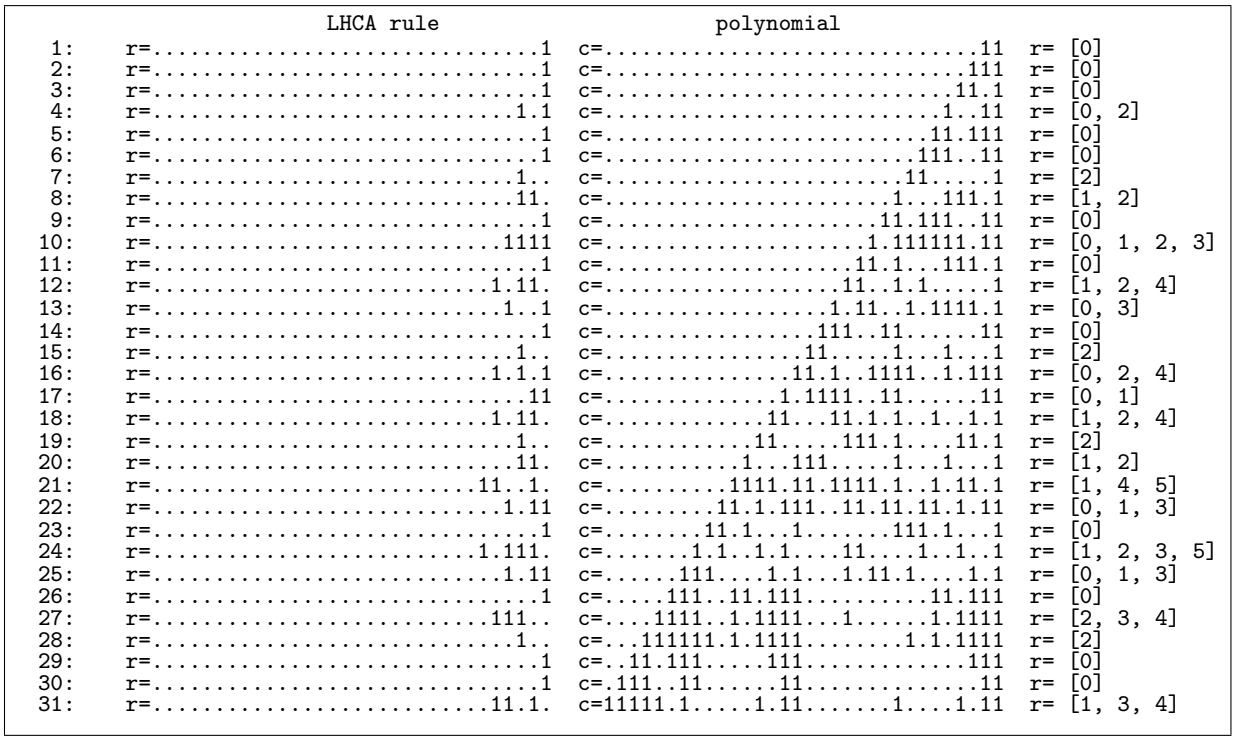
For rules of maximal period the polynomials are primitive. In fact there is an isomorphism between the cycle structure of the successive states of an LHCA and its corresponding shift register.

An LHCA rule and its reverse give the identical polynomial.

The polynomials corresponding to a LHCA rule and its complement are either both reducible or irreducible: If $p(x)$ corresponds to the LHCA rule r then $p(x+1)$ corresponds to the rule that is the complement of r.

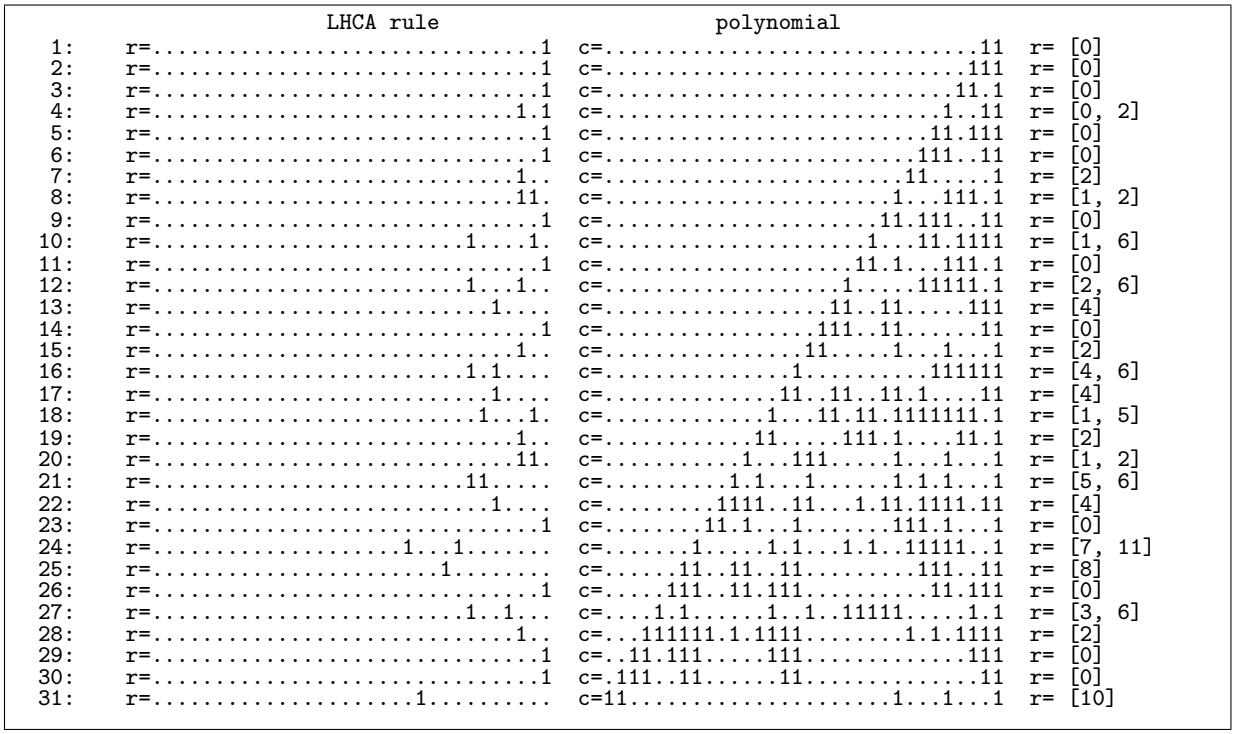
Figure 6.9 on the following page shows a list of LHCA rules where the highest bit lies in the lowest possible position. The list can be produced with [FXT: demo gf2n/lowbit-lhca-demo.cc]. For software implementation of LHCA that need more than one machine word low-bit rules are advantageous. A table of low-bit LHCA rules corresponding to primitive polynomials up to $n = 400$ is given in [FXT: file data/lowbit-lhca-rules.txt]. The maximal value of a rule that occurs is $r = 1293$ for $n = 380$ so the table can be stored compactly.

Figure 6.10 on the next page shows a list of LHCA rules that have minimal weight and the highest bit in the lowest possible position. The list can be produced with [FXT: demo gf2n/minweight-lowbit-lhca-demo.cc].



	LHCA rule	polynomial	
1:	r=.....1	c=.....11	r= [0]
2:	r=.....1	c=.....111	r= [0]
3:	r=.....1	c=.....11.1	r= [0]
4:	r=.....1.1	c=.....1..11	r= [0, 2]
5:	r=.....1	c=.....11.111	r= [0]
6:	r=.....1	c=.....111..11	r= [0]
7:	r=.....1..	c=.....11.....1	r= [2]
8:	r=.....11.	c=.....1...111.1	r= [1, 2]
9:	r=.....1	c=.....11.111..11	r= [0]
10:	r=.....1111	c=.....1.111111.11	r= [0, 1, 2, 3]
11:	r=.....1	c=.....11.1...111.1	r= [0]
12:	r=.....1.11.	c=.....11..1.1.....1	r= [1, 2, 4]
13:	r=.....1..1	c=.....1.11..1.1111.1	r= [0, 3]
14:	r=.....1	c=.....111..11.....11	r= [0]
15:	r=.....1..	c=.....11.....1...1...1	r= [2]
16:	r=.....1.1.1	c=.....11.1..1111..1.111	r= [0, 2, 4]
17:	r=.....11	c=.....1.1111..11.....11	r= [0, 1]
18:	r=.....1.11.	c=.....11...11.1.1..1..1.1	r= [1, 2, 4]
19:	r=.....1..	c=.....11.....111.1...11.1	r= [2]
20:	r=.....11.	c=.....1...111.....1...1...1	r= [1, 2]
21:	r=.....11..1.	c=.....1111.11.1111.1..1.11.1	r= [1, 4, 5]
22:	r=.....1.11	c=.....11.1.111..11.11.11.1.11	r= [0, 1, 3]
23:	r=.....1	c=.....11.1...1.....111.1...1	r= [0]
24:	r=.....1.111.	c=.....1.1..1.1...11....1..1..1	r= [1, 2, 3, 5]
25:	r=.....1.11	c=.....111....1.1...1.11.1....1.1	r= [0, 1, 3]
26:	r=.....1	c=.....111..11.111.....11.111	r= [0]
27:	r=.....111..	c=....1111..1.1111...1.....1.1111	r= [2, 3, 4]
28:	r=.....1..	c=...111111.1.1111.....1.1.1111	r= [2]
29:	r=.....1	c=..11.111.....111.....111	r= [0]
30:	r=.....1	c=.111..11.....11.....11	r= [0]
31:	r=.....11.1.	c=11111.1.....1.11.....1....1.11	r= [1, 3, 4]

Figure 6.9: Low-bit LHCA rules and the corresponding binary polynomials.



	LHCA rule	polynomial	
1:	r=.....1	c=.....11	r= [0]
2:	r=.....1	c=.....111	r= [0]
3:	r=.....1	c=.....11.1	r= [0]
4:	r=.....1.1	c=.....1..11	r= [0, 2]
5:	r=.....1	c=.....11.111	r= [0]
6:	r=.....1	c=.....111..11	r= [0]
7:	r=.....1..	c=.....11.....1	r= [2]
8:	r=.....11.	c=.....1...111.1	r= [1, 2]
9:	r=.....1	c=.....11.111..11	r= [0]
10:	r=.....1....1.	c=.....1...11.1111	r= [1, 6]
11:	r=.....1	c=.....11.1...111.1	r= [0]
12:	r=.....1...1..	c=.....1.....11111.1	r= [2, 6]
13:	r=.....1....	c=.....11..11.....111	r= [4]
14:	r=.....1	c=.....111..11.....11	r= [0]
15:	r=.....1..	c=.....11.....1...1...1	r= [2]
16:	r=.....1.1....	c=.....1.....111111	r= [4, 6]
17:	r=.....1....	c=.....11..11..11.1....11	r= [4]
18:	r=.....1...1.	c=.....1...11.11.1111111.1	r= [1, 5]
19:	r=.....1..	c=.....11.....111.1....11.1	r= [2]
20:	r=.....11.	c=.....1...111.....1...1...1	r= [1, 2]
21:	r=.....11.....	c=.....1.1...1.....1.1.1...1	r= [5, 6]
22:	r=.....1....	c=.....1111..11...1.11.1111.11	r= [4]
23:	r=.....1	c=.....11.1...1.....111.1...1	r= [0]
24:	r=.....1...1.....	c=.....1.....1.1...1.1..11111..1	r= [7, 11]
25:	r=.....1.....	c=.....11..11..11.....111..11	r= [8]
26:	r=.....1	c=.....111..11.111.....11.111	r= [0]
27:	r=.....1..1...	c=....1.1.....1..1..11111.....1.1	r= [3, 6]
28:	r=.....1..	c=...111111.1.1111.....1.1.1111	r= [2]
29:	r=.....1	c=..11.111.....111.....111	r= [0]
30:	r=.....1	c=.111..11.....11.....11	r= [0]
31:	r=.....1.....	c=11.....1...1...1	r= [10]

Figure 6.10: Minimum-weight low-bit LHCA rules and the corresponding binary polynomials.

6.6.2 Conversion of irreducible binary polynomials to LHCAs

The LHCA corresponding to an irreducible binary polynomial $p(x)$ proceeds in two steps. Firstly, the quadratic equation (over $GF(2^n)$)

$$Z^2 + (x^2 + x)\, p'(x)\, Z + 1 \quad = \quad 0 \mod p(x) \tag{6.6}$$

must be solved for Z. The algorithm is given in section 23.4 on page 612, the implementation is given in [FXT: file bpol/bitpolmodsolvequadratic.cc]. The second step is a GCD computation. Set $Z_{n-1} := p$ and $Z_{n-2} := Z$, successively compute Z_{n-3}, Z_{n-4}, ..., Z_0 so that

$$\begin{aligned}
Z_{n-1} &= (x + r_0)\, Z_{n-2} + Z_{n-3} && (6.7a)\\
Z_{n-2} &= (x + r_1)\, Z_{n-3} + Z_{n-4} && (6.7b)\\
&\vdots \\
Z_2 &= (x + r_{n-3})\, Z_1 + 1 && (6.7c)\\
Z_1 &= (x + r_{n-2})\, Z_0 + 1 && (6.7d)\\
Z_0 &= (x + r_{n-1})\, 1 + 0 && (6.7e)
\end{aligned}$$

Each step consists of a computation of polynomial quotient and remainder (see section 22.1 on page 569). The vector $[r_{n-1}, r_{n-2}, \ldots, r_0]$ then gives the LHCA rule. The implementation is given [FXT: file bpol/bitpol2lhca.cc]:

```
ulong
poly2lhca(ulong p)
// Return LHCA rule corresponding to the binary polynomial P.
// Must have: P irreducible.
{
    ulong dp = bitpol_deriv(p);
    const ulong h = bitpol_h(p);

    ulong b = dp;
    b ^= bitpolmod_times_x(b, p, h); // p' * (x+1)
    b = bitpolmod_times_x(b, p, h);  // p' * (x^2+x)

    ulong r0, r1;  // solutions of  1 + (p'*(x*x+x))*z + z*z == 0 modulo p
    bool q = bitpolmod_solve_quadratic(1, b, 1, r0, r1, p);
    if ( 0==q )  return 0;

    // GCD steps:
    ulong r = 0;  // rule vector
    ulong x = p,  y = r0;  // same result with r1
    while ( y )
    {
        ulong tq, tr;
        bitpol_divrem(x, y, tq, tr);
        r <<= 1;
        r |= (tq & 1);
        x = y;
        y = tr;
    }

    return r;
}
```

The described algorithm is given in the very readable paper [33] which is recommended for further studies. Note that the paper uses the reversed rule which can be obtained by inserting the line

```
    r = revbin(r, bitpol_deg(p) );
```

just before the final return statement. The program [FXT: demo gf2n/poly2lhca-demo.cc] converts a given polynomial into the corresponding LHCA rule.

6.7 Additive linear hybrid cellular automata

The algorithm for the conversion of LHCA rules to binary polynomials is a special case of a general method for additive cellular automata. An automaton is called additive if, for all words a and b,

$$N(a) + N(b) \quad = \quad N(a + b) \tag{6.8}$$

where $N(x)$ is the next state after the state x and addition is bit-wise (XOR). For additive automata the action of $N(.)$ on a word can be described by a matrix over $GF(2)$: Let e_k be the word with bit number k the only set bit (e_k, $k = 0, \ldots, n-1$ is the canonical basis). The matrix whose k-th row is $N(e_k)$ is the matrix we seek.

For example, for the automaton whose action $N(.)$ can be implemented as [FXT: demo gf2n/clhca-demo.cc]

```
ulong clhca_next(ulong x, ulong r, ulong n)
// Compute next state of a linear hybrid cellular automaton
//  with cyclic boundary condition (CLHCA).
{
    r &= x;
    ulong t = x ^ bit_rotate_right(x, 1, n);
    t ^= r;
    return  t;
}
```

we obtain for the rule $r := [r_0, r_1, \ldots, r_{n-1}]$ the matrix

$$M_r \quad := \quad \begin{bmatrix} s_0 & & & & & & & 1 \\ 1 & s_1 & & & & & & \\ & 1 & s_2 & & & & & \\ & & 1 & s_3 & & & & \\ & & & \ddots & \ddots & & & \\ & & & & 1 & s_{n-3} & & \\ & & & & & 1 & s_{n-2} & \\ & & & & & & 1 & s_{n-1} \end{bmatrix} \tag{6.9}$$

where s_k is the complement of r_k and blank entries denote zero. The binary polynomial corresponding to the automaton is the characteristic polynomial of the matrix M_r. We use the routine `bitmat_charpoly()` given in [FXT: file bmat/bitmatcharpoly.cc] (which uses a reduction to the Hessenberg form as given in [Cohen]):

```
ulong clhca2poly(ulong r, ulong n)
// Compute the binary polynomial corresponding
//   to the length-n CLHCA with rule r.
{
    ALLOCA(ulong, M, n);
    for (ulong k=0; k<n; ++k)
    {
        M[k] = clhca_next( 1UL<<k, r, n );
    }
    ulong c = bitmat_charpoly(M, n);
    return c;
}
```

Note that with our particular example the matrix is already in Hessenberg form so we could directly call `bitmat_hessenberg2charpoly()`.

Properties of the CLHCA

It turns out that the polynomials for the given automata depend only on the number of bits set in the rule r. Figure 6.11 on the facing page shows the successive states for all length-4 CLHCA with rules of weight 3. As there are essentially only n different automata of length n we only need to investigate the rules with the lowest k bits bits set for $k = 0, 1, \ldots, n-1$. The polynomials for the length-17 automata are shown in figure 6.12 on the next page. If we write $C_k(x)$ for the polynomial for the k-bit rule then $C_{n-k}(x) = C_k(x+1)$. The polynomials can actually be computed with the simple routine

```
          r = .111    r = 1.11    r = 11.1    r = 111.

      1:      ...1        ...1        ...1        ...1
      2:      1...        1...        1...        1..1
      3:      11..        .1..        .1..        11.1
      4:      111.        .11.        ..1.        1111
      5:      1111        .111        ..11        111.
      6:      .111        1111        1.11        .111
      7:      1.11        1.11        1111        1.1.
      8:      .1.1        11.1        11.1        .1.1
      9:      1.1.        1.1.        111.        1.11
     10:      11.1        .1.1        .1.1        11..
     11:      .11.        111.        1.1.        .11.
     12:      ..11        ..11        .111        ..11
     13:      1..1        1..1        1..1        1...
     14:      .1..        11..        11..        .1..
     15:      ..1.        ..1.        .11.        ..1.
```

Figure 6.11: Rules of identical weight lead to essentially identical CLHCA: All tracks in the successive states of all weight-3 (length-4) automata are cyclic shifts of each other.

```
     k      r =  CLHCA rule            c = binary polynomial
     0:     r = ................       c = 11..............1.
     1:     r = ...............1       c = 1.................11
     2:     r = ..............11       c = 1111111111111111.1
     3:     r = .............111       c = 1.1.1.1.1.1.1.1.1..1  P
     4:     r = ............1111       c = 11..11..11..11...1
     5:     r = ...........11111       c = 1...1...1...1....1  P
     6:     r = ..........111111       c = 1111....1111.....1  P
     7:     r = .........1111111       c = 1.1.....1.1......1
     8:     r = ........11111111       c = 11......11.......1
     9:     r = ........111111111      c = 1.......1........1
    10:     r = .......1111111111      c = 11111111.........1
    11:     r = ......11111111111      c = 1.1.1.1..........1  P
    12:     r = .....111111111111      c = 11..11...........1  P
    13:     r = ....1111111111111      c = 1...1............1
    14:     r = ...11111111111111      c = 1111.............1  P
    15:     r = ..111111111111111      c = 1.1..............1
    16:     r = .1111111111111111      c = 11...............1
    17:     r = 11111111111111111      c = 1................1
```

Figure 6.12: Binary polynomials (right) corresponding to rule vectors for the length-17 cyclic linear hybrid cellular automata (left) with k bits set. Automata with maximal period are marked with '`P`'.

```
ulong clhca2poly(ulong r, ulong n)
{
    ulong c = 1UL << n;
    for (ulong k=0; k<n; ++k)  if ( 0==(r & (1UL<<k)) )  c ^= (c>>1);
    c ^= 1;
    return c;
}
```

With $n = 5$ there are just two rules that lead to maximal periods, $r = [0, 0, 0, 1, 1]$ (weight 2), and $r = [0, 0, 1, 1, 1]$ (weight 3). The successive states for both rules are shown in figure 6.13 on the following page. The polynomials corresponding to the rules of minimal weight for all length-n automata where $n \leq 30$ are given in figure 6.14 on the next page.

```
 w=2:  r=...11          w=3:  r=..111          w=1:  r=....1

 1:      ....1          1:      ....1          1:      ....1
 2:      1....          2:      1....          2:      1....
 3:      11...          3:      11...          3:      11...
 4:      1.1..          4:      1.1..          4:      1.1..
 5:      1111.          5:      11.1.          5:      1111.
 6:      1..11          6:      1.1.1          6:      1...1
 7:      .1..1          7:      .1.1.          7:      .1...
 8:      111..          8:      .11.1          8:      .11..
 9:      1..1.          9:      1111.          9:      .1.1.
10:      11..1         10:      1.111         10:      .1111
11:      ..1..         11:      .1.11         11:      11..1
12:      ..11.         12:      111.1         12:      ..1..
13:      ..111         13:      ..11.         13:      ..11.
14:      1.111         14:      ...11         14:      ..1.1
15:      .1111         15:      1...1         15:      1.11.
16:      11.11         16:      .1...         16:      111.1
17:      ..1.1         17:      .11..         17:      ...1.
18:      1.11.         18:      .111.         18:      ...11
19:      11111         19:      .1111         19:      1..11
20:      ...11         20:      11111         20:      .1.11
21:      1...1         21:      ..111         21:      11111
22:      .1...         22:      1..11
23:      .11..         23:      .1..1
24:      .1.1.         24:      111..
25:      .11.1         25:      1.11.
26:      11.1.         26:      11.11
27:      1.1.1         27:      ..1.1
28:      .111.         28:      1..1.
29:      .1.11         29:      11..1
30:      111.1         30:      ..1..
31:      ...1.         31:      ...1.
```

Figure 6.13: The length-5, weight-w CLHCA has maximal period for $w = 2$ and $w = 3$. The successive states are shown for both automata (left and middle). The weight-1 rule leads to a period of 21 (right).

```
  n  w:     c = polynomial

  2  1:     c = 111
  3  1:     c = 1.11
  4  3:     c = 11..1
  5  2:     c = 1111.1
  6  1:     c = 11..111
  7  1:     c = 1.1.1.11
 [8]
  9  4:     c = 11..11...1
 10  3:     c = 11111111..1
 11  2:     c = 11......11.1
[12,13,14]
 15  4:     c = 1111....1111...1
[16]
 17  3:     c = 1.1.1.1.1.1.1.1..1
 18  7:     c = 1111....1111......1
[19]
 20  3:     c = 11............11..1
 21  2:     c = 1111..........1111.1
 22 21:     c = 11..............1....1
 23  5:     c = 1.1............1.1....1
[24]
 25  3:     c = 1.1.1.1.........1.1.1.1..1
[26,27]
 28  9:     c = 1111............1111........1
 29  2:     c = 1111....1111....1111....1111.1
[30]
```

Figure 6.14: The polynomials (right) corresponding rules of lowest weight (w) so that the length-n ($n \leq 30$) CLHCA has maximal period.

Chapter 7

Searching paths in directed graphs

In this chapter we describe how certain combinatorial structures can be represented as paths or cycles in a directed graph.

As an example consider Gray codes of n-bit binary words: we are looking for sequences of all 2^n binary words such that only one bit changes between two successive words. A convenient representation of the search space is that of a graph. The nodes are the binary words and an edge is drawn between two nodes if the node's values differ by exactly one bit.

Every path that visits all nodes of that graph corresponds to a Gray code. If the path is a cycle then a Gray cycle was found. In general we can, depending on the size of the problem,

1. generate all such objects or show that none exists
2. try to find at least one such object

We will see how to reduce the search space if additional constraints are imposed on the paths. Finally, we show how careful optimization can lead to surprising algorithms for objects of a size where one would hardly expect to get a result at all. In fact, Gray cycles through the n-bit binary Lyndon words for all prime n less or equal to 37 are determined.

Terminology and conventions

We will use the terms *node* (instead of *vertex*) and *edge* (sometimes called *arc*). We restrict our attention to *directed graphs* (or *digraphs*) as undirected graphs are just the special case of these: an edge in an undirected graph corresponds to two antiparallel edges (think: 'arrows') in a directed graph.

A length-k *path* is a sequence of nodes where an edge leads from each node to its successor. A path is called *simple* if the nodes are pairwise distinct. We restrict our attention to simple paths of length N where N is the number of nodes of in the graph. We use the term *full path* for a simple path of length N.

If in a simple path there is an edge from the last node of the path to the starting node the path is a *cycle* (or *circuit*). A full path that is a cycle is called a *Hamiltonian cycle*, a graph containing such a cycle is called *Hamiltonian*.

We allow for *loops* (edges that start and point to the same node). Graphs that contain loops are called *pseudo graphs*. The algorithms used will effectively ignore loops.

We disallow *multigraphs* (where multiple edges can start and end at the same two nodes), as these would lead to repeated output of identical objects.

The *neighbors* of a node are those nodes where outgoing edges point to. Neighbors can be reached with one step. The neighbors of a node a called *adjacent* to it. The *adjacency matrix* of a graph with N nodes is a $N \times N$ matrix A where $A_{i,j} = 1$ if there is an edge from node i to node j, else $A_{i,j} = 0$. While trivial to implement (and later modify) we will not use this kind of representation as the memory requirement would be prohibitive for large graphs.

7.1 Representation of digraphs

For our purposes a static implementation of the graph as arrays of nodes and (outgoing) edges will do. The container class `digraph` merely allocates memory for the nodes and edges. The correct initialization is left to the user [FXT: `class digraph` in graph/digraph.h]:

```
class digraph
{
public:
    ulong ng_;   // number of Nodes of Graph
    ulong *ep_;  // e[ep[k]], ..., e[ep[k+1]-1]: outgoing connections of node k
    ulong *e_;   // outgoing connections (Edges)
    ulong *vn_;  // optional: sorted values for nodes
    // if vn is used, then node k must correspond to vn[k]
public:
    digraph(ulong ng, ulong ne, ulong *&ep, ulong *&e, bool vnq=false)
        : ng_(0), ep_(0), e_(0), vn_(0)
    {
        ng_ = ng;
        ep_ = new ulong[ng_+1];
        e_ = new ulong[ne];
        ep = ep_;
        e = e_;
        if ( vnq )  vn_ = new ulong[ng_];
    }
    ~digraph()
    {
        delete [] ep_;
        delete [] e_;
        if ( vn_ )  delete [] vn_;
    }
 [--snip--]
    void get_edge_idx(ulong p, ulong &fe, ulong &en)  const
    // Setup fe and en so that the nodes reachable from p are
    //   e[fe], e[fe+1], ..., e[en-1].
    // Must have:  0<=p<ng
    {
        fe = ep_[p];   // (index of) First Edge
        en = ep_[p+1]; // (index of) first Edge of Next node
    }
 [--snip--]
    void print(const char *bla=0)  const;
};
```

The nodes reachable from node p could be listed using

```
    // ulong p; // == position
    cout << "The nodes reachable from node " << p << " are:" << endl;
    ulong fe, en;
    g_.get_edge_idx(p, fe, en);
    for (ulong ep=fe; ep<en; ++ep)  cout << e_[ep] << endl;
```

Using our representation there is no cheap method to find the incoming edges. We will not need this information for our purposes. If the graph is known to be undirected, the same routine obviously lists the incoming edges.

Initialization routines for certain digraphs are declared in [FXT: file graph/digraphspecial.h]. A simple example is [FXT: `make_complete_digraph()` in graph/digraphcomplete.cc]:

```
digraph
```

```
make_complete_digraph(ulong n)
// Initialization for the complete graph.
{
    ulong ng = n, ne = n*(n-1);
    ulong *ep, *e;
    digraph dg(ng, ne, ep, e);
    ulong j = 0;
    for (ulong k=0; k<ng; ++k)  // for all nodes
    {
        ep[k] = j;
        for (ulong i=0; i<n; ++i)  // connect to all nodes
        {
            if ( k==i )  continue;  // no loops
            e[j++] = i;
        }
    }
    ep[ng] = j;
    return  dg;
}
```

We initialize the *complete graph*[1] for $n = 5$ and print it:

```
    digraph dg = make_complete_digraph(5);
    dg.print("Graph =");
```

We get

```
Graph =
Node: Edge0 Edge1 ...
  0:    1   2   3   4
  1:    0   2   3   4
  2:    0   1   3   4
  3:    0   1   2   4
  4:    0   1   2   3
  #nodes=5  #edges=20
```

For many purposes it suffices to implicitly represent the nodes as values p with $0 \le p < N$. If not, the values of the nodes have to be stored in the array `vn_[]`. One such example is a graph where the value of node p is the p-th (cyclically minimal) Lyndon word that we will meet at the end of this chapter.

In order to be able to find a node by value reasonably fast the array should be sorted in ascending order so that binary search can be used.

7.2 Searching full paths

In order to search full paths starting from some position p_0 we need to additional arrays for the bookkeeping: A record of the path so far, its k-th entry shall be p_k, the node visited at step k. Further a tag array that shall contain a zero for nodes not visited so far, else one.

The crucial parts of the implementation [FXT: `class digraph_paths` in graph/digraphpaths.h] are:

```
class digraph_paths
// Find all full paths in a directed graph.
{
public:
    digraph &g_;  // the graph
    ulong *rv_;  // Record of Visits: rv[k] == node visited at step k
    ulong *qq_;  // qq[k] == whether node k has been visited yet
 [--snip--]
    // function to call with each path found with all_paths():
    ulong (*pfunc_)(digraph_paths &);
 [--snip--]
    // function to impose condition with all_cond_paths():
    bool (*cfunc_)(digraph_paths &, ulong ns);
public:
    // graph/digraph.cc:
    digraph_paths(digraph &g);
```

[1] The complete graph is the undirected graph that has edges between any two of its nodes.

```
    ~digraph_paths();
 [--snip--]
    bool path_is_cycle()  const;
 [--snip--]
    void print_path() const;
 [--snip--]
    // graph/digraphpaths-search.cc:
    ulong all_paths(ulong (*pfunc)(digraph_paths &),
                    ulong ns=0, ulong p=0, ulong maxnp=0);
    void next_path(ulong ns, ulong p);  // called by all_paths()
 [--snip--]
};
```

We could have used a bit-array for the tag values `qq_`. It turns out that some additional information can be saved there as we will see in a moment.

To keep matters simple a recursive algorithm is used to search for (full) paths. After initialization of the arrays by `all_paths()` the routine [FXT: `digraph_paths::next_path()` in graph/digraphpaths-search.cc] is called. When a path is found a user supplied function `pfunc()` is called that, for example, may print the result.

The essential parts of `next_path()` are

```
void
digraph_paths::next_path(ulong ns, ulong p)
// ns+1 == how many nodes seen
// p == position (node we are on)
{
    rv_[ns] = p;  // record position
    ++ns;

    if ( ns==ng_ )  // all nodes seen ?
    {
        pfunc_(*this);
    }
    else
    {
        qq_[p] = 1; // mark position as seen (else loops lead to errors)
        ulong fe, en;
        g_.get_edge_idx(p, fe, en);
        ulong fct = 0;  // count free reachable nodes  // FCT
        for (ulong ep=fe; ep<en; ++ep)
        {
            ulong t = g_.e_[ep]; // next node
            if ( 0==qq_[t] )  // node free?
            {
                ++fct;
                qq_[p] = fct; // mark position as seen: record turns  // FCT
                next_path(ns, t);
            }
        }
        // if ( 0==fct )  { "dead end: this is a U-turn"; }  // FCT

        qq_[p] = 0; // unmark position
    }
}
```

The lines that are commented with `// FCT` record which among the free nodes is visited. The algorithm still works if these lines are commented out.

7.2.1 Paths in the complete graph: permutations

The demo in [FXT: demo graph/graph-perm-demo.cc] shows the paths in the complete graph from section 7.1, page 235. The version here is slightly simplified:

```
ulong pfunc_perm(digraph_paths &dp)
// Function to be called with each path:
//  print all but the first node.
{
    const ulong *rv = dp.rv_;
```

```
    ulong ng = dp.ng_;
    cout << setw(4) << dp.pfct_ << ":  ";
    for (ulong k=1; k<ng; ++k)  cout << " " << rv[k];
    cout << endl;
    return 1;
}
int
main(int argc, char **argv)
{
    ulong n = 5;
    digraph dg = make_complete_digraph(n);
    digraph_paths dp(dg);
    dg.print("Graph =");
    cout << endl;
    dp.all_paths(pfunc_perm, 0, 0, maxnp);
    return 0;
}
```

The output is [FXT: demo graph/graph-perm-out.txt]:

```
Graph =
Node: Edge0 Edge1 ...
  0:    1   2   3   4
  1:    0   2   3   4
  2:    0   1   3   4
  3:    0   1   2   4
  4:    0   1   2   3
  #nodes=5  #edges=20

   0:   1 2 3 4
   1:   1 2 4 3
   2:   1 3 2 4
   3:   1 3 4 2
   4:   1 4 2 3
   5:   1 4 3 2
   6:   2 1 3 4
   7:   2 1 4 3
   8:   2 3 1 4
   9:   2 3 4 1
  10:   2 4 1 3
  11:   2 4 3 1
  12:   3 1 2 4
  13:   3 1 4 2
  14:   3 2 1 4
  15:   3 2 4 1
  16:   3 4 1 2
  17:   3 4 2 1
  18:   4 1 2 3
  19:   4 1 3 2
  20:   4 2 1 3
  21:   4 2 3 1
  22:   4 3 1 2
  23:   4 3 2 1
```

These are the permutations of the numbers $1, 2, 3, 4$ in lexicographic order (see section 5.17.1 on page 190).

7.2.2 Paths in the De Bruijn graph: De Bruijn sequences

The graph with $2\,n$ nodes and two outgoing edges from node k to $2\,k \bmod 2\,n$ and $2\,k+1 \bmod 2\,n$ is called a *De Bruijn graph*. For $n = 8$ the graph is (printed horizontally):

```
    Node:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
Edge 0:  0  2  4  6  8 10 12 14  0  2  4  6  8 10 12 14
Edge 1:  1  3  5  7  9 11 13 15  1  3  5  7  9 11 13 15
```

The graph has two loops at the first and the last node. All paths in the De Bruijn graph are cycles, the graph is Hamiltonian.

With n a power of two the paths correspond to the *De Bruijn sequences* (DBS) of length $2\,n$. The graph has as many full paths as there are DBSs and the zeros/ones in the DBS correspond to even/odd values of the nodes, respectively. This is demonstrated in [FXT: demo graph/graph-debruijn-demo.cc] (shortened):

```
ulong pq = 1; // whether and what to print with each cycle

ulong pfunc_db(digraph_paths &dp)
// Function to be called with each cycle.
{
    switch ( pq )
    {
```

```
    case 0:  break;  // just count
    case 1:  // print lowest bits (De Bruijn sequence)
        {
            ulong *rv = dp.rv_, ng = dp.ng_;
            for (ulong k=0; k<ng; ++k)  cout << (rv[k]&1UL ? '1' : '.');
            cout << endl;
            break;
        }
    [--snip--]
    }
    return 1;
}
int main(int argc, char **argv)
{
    ulong n = 8;
    NXARG(pq, "what to do in pfunc()");
    ulong maxnp = 0;
    NXARG(maxnp, "stop after maxnp paths (0: never stop)");
    ulong p0 = 0;
    NXARG(p0, "start position <2*n");

    digraph dg = make_debruijn_digraph(n);
    digraph_paths dp(dg);

    dg.print_horiz("Graph =");

    // call pfunc() with each cycle:
    dp.all_paths(pfunc_db, 0, p0, maxnp);

    cout << "n = " << n;
    cout << " (ng=" << dg.ng_ << ")";
    cout << "   #cycles = " << dp.cct_;
    cout << endl;

    return 0;
}
```

The macro `NXARG(.,.)` is defined in [FXT: demo nextarg.h]. The output is

```
    arg 1: 8 == n  [size of graph == 2*n]  default=8
    arg 2: 1 == pq  [what to do in pfunc()]  default=1
    arg 3: 0 == maxnp  [stop after maxnp paths (0: never stop)]  default=0
    arg 4: 0 == p0  [start position <2*n]  default=0
    Graph =
      Node:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
    Edge 0:  0  2  4  6  8 10 12 14  0  2  4  6  8 10 12 14
    Edge 1:  1  3  5  7  9 11 13 15  1  3  5  7  9 11 13 15
    .1..11.1.1111...
    .1..1111.1.11...
    .1.1..11.1111...
    .1.1..1111.11...
    .1.11..1111.1...
    .1.11.1..1111...
    .1.1111..11.1...
    .1.1111.1..11...
    .11..1.1111.1...
    .11.1..1.1111...
    .11.1.1111..1...
    .11.1111..1.1...
    .1111..1.11.1...
    .1111.1..1.11...
    .1111.1.11..1...
    .1111.11..1.1...
    n = 8 (ng=16)   #cycles = 16
```

The algorithm is a very effective way of generating all DBSs of a given length, the 67,108,864 DBSs of length 64 are generated in 140 seconds when printing is disabled (set argument `pq` to zero), corresponding to a rate of more than 450,000 cycles per second.

Setting the argument `pq` to 4 prints the binary values of the successive nodes in the path in horizontal order:

```
./bin 32 4 1
    -#----##---#-#---###--#--#-##--##-#--####-#-#-###-##-######-----
    --#----##---#-#---###--#--#-##--##-#--####-#-#-###-##-######----
    ---#----##---#-#---###--#--#-##--##-#--####-#-#-###-##-######---
    ----#----##---#-#---###--#--#-##--##-#--####-#-#-###-##-######--
    -----#----##---#-#---###--#--#-##--##-#--####-#-#-###-##-######-
    ------#----##---#-#---###--#--#-##--##-#--####-#-#-###-##-######
```

The graph is constructed in a way that each word is the predecessor shifted by one with either zero or one inserted at position zero (top row).

The number of cycles in the De Bruijn graph equals the number of degree-n normal binary polynomials. An efficient procedure to compute it is given in section 23.10 on page 629. A closed form for the special case $n = 2^k$ is given in section 6.3 on page 221.

7.2.3 A modified De Bruijn graph: complement-shift sequences

A modification of the De Bruijn graph forces the nodes to be the complement of its predecessor shifted by one (again with either zero or one inserted at position zero). The routine to set up the graph is [FXT: `make_complement_shift_digraph()` in graph/digraphdebruijn.cc]:

```
digraph
make_complement_shift_digraph(ulong n)
{
    ulong ng = 2*n, ne = 2*ng;
    ulong *ep, *e;
    digraph dg(ng, ne, ep, e);
    ulong j = 0;
    for (ulong k=0; k<ng; ++k)  // for all nodes
    {
        ep[k] = j;
        ulong r = (2*k) % ng;
        e[j++] = r;  // connect node k to node (2*k) mod ng
        r = (2*k+1) % ng;
        e[j++] = r;  // connect node k to node (2*k+1) mod ng
    }
    ep[ng] = j;
    // Here we have a De Bruijn graph.
    for (ulong k=0,j=ng-1; k<j; ++k,--j) swap2(e[ep[k]], e[ep[j]]); // end with ones
    for (ulong k=0,j=ng-1; k<j; ++k,--j) swap2(e[ep[k]+1], e[ep[j]+1]);
    return  dg;
}
```

The demo [FXT: demo graph/graph-complementshift-demo.cc] gives:

```
arg 1: 32 == n  [size of graph == 2*n]  default=32
arg 2: 4 == pq  [whether and what to print with each path]  default=2
arg 3: 1 == maxnp  [stop after maxnp paths (0: never stop)]  default=1

------#---#-#-##----##--#--#-###---###-#--##-####--######-##-#-#
-######-###-#-#--####--##-##-#---###---#-##--#----##------#--#-#
-#------#---#-#-##----##--#--#-###---###-#--##-####--######-##-#
-#-######-###-#-#--####--##-##-#---###---#-##--#----##------#--#
-#-#------#---#-#-##----##--#--#-###---###-#--##-####--######-##
-#-#-######-###-#-#--####--##-##-#---###---#-##--#----##------#-
```

For n a power of two the sequence of binary words has the interesting property that the changes between successive words depend on their sequency: Words with higher sequency change in less positions. Further, if two neighbor bits are set in some word then the next word never contains both two bits again. Out of a run of $k > 2$ consecutive set bits in a word only one is is contained in the next word.

7.3 Conditional search

Sometimes one wants to find paths that are subject to certain restrictions. Testing for each path found whether it has the desired property and discarding it if not is the most simple way. However, this will in many cases be extremely ineffective. An upper bound for the number of recursive calls of the search function `next_path()` with an graph with N nodes and an maximal number of v outgoing edges at each node is $u = N^v$.

For example, the graph corresponding to Gray codes of n-bit binary words has $N = 2^n$ nodes and (exactly) $c = n$ outgoing edges at each node. The graph is the n-dimensional hypercube.

n	N	$u = N^c = N^n = 2^{n \cdot n}$
1	2	2
2	4	16
3	8	512
4	16	65,536
5	32	33,554,432
6	64	68,719,476,736
7	128	562,949,953,421,312
8	256	18,446,744,073,709,551,616
9	512	2,417,851,639,229,258,349,412,352
10	1024	1,267,650,600,228,229,401,496,703,205,376

We are obviously interested in the reduction of the size of the search space. This can in many cases be achieved by a function that rejects branches that would lead to a path not satisfying the imposed restrictions.

A conditional search can be started via `all_cond_paths()` that has an additional function pointer `cfunc()` as argument that shall implement the condition. It is declared as

```
bool (*cfunc_)(digraph_paths &, ulong ns);
```

Besides the data from the digraph-class it needs the number of nodes seen so far (`ns`) as an argument. A slight modification of the search routine `next_path()` does what we want:

```
void
digraph_paths::next_cond_path(ulong ns, ulong p)
{
    [--snip--]  // same as next_path()
    if ( ns==ng_ )  // all nodes seen ?
    [--snip--]  // same as next_path()
    else
    {
        qq_[p] = 1; // mark position as seen (else loops lead to errors)
        ulong fe, en;
        g_.get_edge_idx(p, fe, en);
        ulong fct = 0;  // count free reachable nodes
        for (ulong ep=fe; ep<en; ++ep)
        {
            ulong t = g_.e_[ep]; // next node
            if ( 0==qq_[t] )  // node free?
            {
                rv_[ns] = t;  // for cfunc()
                if ( cfunc_(*this, ns) )
                {
                    ++fct;
                    qq_[p] = fct; // mark position as seen: record turns
                    next_cond_path(ns, t);
                }
            }
        }
        qq_[p] = 0; // unmark position
    }
}
```

The free node under consideration is written to the end of the record of visited nodes so `cfunc()` does not need it as an explicit argument.

7.3.1 Modular adjacent changes (MAC) Gray codes

We search for Gray paths that have the *modular adjacent changes* (MAC) property: the values of successive elements of the delta sequence shall change by ± 1 modulo n. Two examples are show in figure 7.1 on the facing page. The sequence on the right side even has the stated property if the term 'modular' is omitted: It has the *adjacent changes* (AC) property.

As bitwise cyclic shifts and reflections of MAC Gray codes are again MAC Gray codes we consider paths starting $0 \to 1 \to 3$ as canonical paths.

```
 0:  .... 0    0    ...1  0          0:  .... 0    0    ...1  0
 1:  ...1 1    1    ..1.  1          1:  ...1 1    1    ..1.  1
 2:  ..11 2    3    .1..  2          2:  ..11 2    3    .1..  2
 3:  .111 3    7    1...  3          3:  .111 3    7    ..1.  1
 4:  1111 4   15    ...1  0          4:  .1.1 2    5    ...1  0
 5:  111. 3   14    ..1.  1          5:  .1.. 1    4    ..1.  1
 6:  11.. 2   12    ...1  0          6:  .11. 2    6    .1..  2
 7:  11.1 3   13    1...  3          7:  ..1. 1    2    1...  3
 8:  .1.1 2    5    ...1  0          8:  1.1. 2   10    .1..  2
 9:  .1.. 1    4    ..1.  1          9:  111. 3   14    ..1.  1
10:  .11. 2    6    .1..  2         10:  11.. 2   12    ...1  0
11:  ..1. 1    2    1...  3         11:  11.1 3   13    ..1.  1
12:  1.1. 2   10    ...1  0         12:  1111 4   15    .1..  2
13:  1.11 3   11    ..1.  1         13:  1.11 3   11    ..1.  1
14:  1..1 2    9    ...1  0         14:  1..1 2    9    ...1  0
15:  1... 1    8   [1...  3]        15:  1... 1    8   [1...  3]
```

Figure 7.1: Two four-bit modular adjacent changes (MAC) Gray codes. Both are cycles.

In the demo [FXT: demo graph/graph-macgray-demo.cc] the search is done as follows (shortened):

```
int main(int argc, char **argv)
{
    ulong n = 5;
    NXARG(n, "size in bits");
    cf_nb = n;

    digraph dg = make_gray_digraph(n, 0);
    digraph_paths dp(dg);

    ulong ns = 0, p = 0;
    // MAC: canonical paths start as 0-->1-->3
    {
        dp.mark(0, ns);
        dp.mark(1, ns);
        p = 3;
    }

    dp.all_cond_paths(pfunc, cfunc_mac, ns, p, maxnp);
    return 0;
}
```

The function used to impose the MAC condition is:

```
ulong cf_nb; // number of bits,  set in main()
bool cfunc_mac(digraph_paths &dp, ulong ns)
// Condition: difference of successive delta values (modulo n) == +-1
{
    // path initialized, we have ns>=2
    ulong p  = dp.rv_[ns],  p1 = dp.rv_[ns-1], p2 = dp.rv_[ns-2];
    ulong c = p ^ p1,  c1 = p1 ^ p2;
    if ( c & bit_rotate_left(c1,1,cf_nb) )  return  true;
    if ( c1 & bit_rotate_left(c,1,cf_nb) )  return  true;
    return false;
}
```

One finds paths for $n \leq 7$ ($n = 7$ takes about 15 minutes). Whether MAC Gray codes exist for $n \geq 8$ is unknown (none is found with a 40 hour search).

7.3.2 Adjacent changes (AC) Gray codes

For AC paths one can only discard track-reflected solutions, the canonical paths are those where the delta sequence starts with a values that is $\leq \lceil n/2 \rceil$. A function to impose the AC condition is

```
ulong cf_mt; // mid track < cf_mt,  set in main()
bool cfunc_ac(digraph_paths &dp, ulong ns)
// Condition: difference of successive delta values == +-1
{
    if ( ns<2 )  return  (dp.rv_[1] < cf_mt); // avoid track-reflected solutions
    ulong p  = dp.rv_[ns],  p1 = dp.rv_[ns-1], p2 = dp.rv_[ns-2];
    ulong c = p ^ p1, c1 = p1 ^ p2;
    if ( c & (c1<<1) )  return  true;
    if ( c1 & (c<<1) )  return  true;
```

```
    return false;
}
```

```
      0:  ..... 0    0    ..1..  2          0:  ....., 0    0    ....1  0
      1:  ..1.. 1    4    .1...  3          1:  ....1 1    1    ...1.  1
      2:  .11.. 2   12    1....  4          2:  ...11 2    3    ..1..  2
      3:  111.. 3   28    .1...  3          3:  ..111 3    7    .1...  3
      4:  1.1.. 2   20    ..1..  2          4:  .1111 4   15    ..1..  2
      5:  1.... 1   16    ...1.  1          5:  .1.11 3   11    ...1.  1
      6:  1..1. 2   18    ..1..  2          6:  .1..1 2    9    ..1..  2
      7:  1.11. 3   22    .1...  3          7:  .11.1 3   13    .1...  3
      8:  1111. 4   30    ..1..  2          8:  ..1.1 2    5    1....  4
      9:  11.1. 3   26    ...1.  1          9:  1.1.1 3   21    .1...  3
     10:  11... 2   24    ....1  0         10:  111.1 4   29    ..1..  2
     11:  11..1 3   25    ...1.  1         11:  11..1 3   25    ...1.  1
     12:  11.11 4   27    ..1..  2         12:  11.11 4   27    ..1..  2
     13:  11111 5   31    .1...  3         13:  11111 5   31    .1...  3
     14:  1.111 4   23    ..1..  2         14:  1.111 4   23    ..1..  2
     15:  1..11 3   19    ...1.  1         15:  1..11 3   19    ...1.  1
     16:  1...1 2   17    ..1..  2         16:  1...1 2   17    ....1  0
     17:  1.1.1 3   21    .1...  3         17:  1.... 1   16    ...1.  1
     18:  111.1 4   29    1....  4         18:  1..1. 2   18    ..1..  2
     19:  .11.1 3   13    .1...  3         19:  1.11. 3   22    .1...  3
     20:  ..1.1 2    5    ..1..  2         20:  1111. 4   30    ..1..  2
     21:  ....1 1    1    ...1.  1         21:  11.1. 3   26    ...1.  1
     22:  ...11 2    3    ..1..  2         22:  11... 2   24    ..1..  2
     23:  ..111 3    7    .1...  3         23:  111.. 3   28    .1...  3
     24:  .1111 4   15    ..1..  2         24:  1.1.. 2   20    1....  4
     25:  .1.11 3   11    ...1.  1         25:  ..1.. 1    4    .1...  3
     26:  .1..1 2    9    ....1  0         26:  .11.. 2   12    ..1..  2
     27:  .1... 1    8    ...1.  1         27:  .1... 1    8    ...1.  1
     28:  .1.1. 2   10    ..1..  2         28:  .1.1. 2   10    ..1..  2
     29:  .111. 3   14    .1...  3         29:  .111. 3   14    .1...  3
     30:  ..11. 2    6    ..1..  2         30:  ..11. 2    6    ..1..  2
     31:  ...1. 1    2   [...1.  1]       31:  ...1. 1    2   [...1.  1]
```

Figure 7.2: Two five-bit adjacent changes (AC) Gray codes that are cycles.

The program [FXT: demo graph/graph-acgray-demo.cc] allows searches for AC Gray codes. Two cycles for $n = 5$ is shown in figure 7.2. It turns out that such paths exist for $n \leq 6$ (the only path for $n = 6$ is shown in figure 7.3 on the facing page) but there is no AC Gray code for $n = 7$:

```
 time ./bin 7
arg 1: 7 == n  [size in bits]  default=5
arg 2: 0 == maxnp  [ stop after maxnp paths (0: never stop)]  default=0
n = 7   #pfct = 0
   #paths = 0   #cycles = 0
./bin 7  20.77s user 0.11s system 98% cpu 21.232 total
```

Nothing is known about the case $n \geq 8$.

Inspection of the AC Gray codes for different values of n result in a hand-woven algorithm. The function [FXT: `ac_gray_delta()` in comb/acgray.cc] computes the delta sequence for an AC Gray codes for $n \leq 6$:

```
void
ac_gray_delta(uchar *d, ulong ldn)
// Generate a delta sequence for an adjacent-changes (AC) Gray code
//  of length n=2**ldn where ldn<=6.
{
    if ( ldn<=2 )  // standard Gray code
    {
        d[0] = 0;
        if ( ldn==2 ) { d[1] = 1; d[2] = 0; }
        return;
    }

    ac_gray_delta(d, ldn-1);  // recursion

    ulong n = 1UL<<ldn;
    ulong nh = n/2;
    if ( 0==(ldn&1) )
    {
        if ( ldn>=6 )
        {
            reverse(d, nh-1);
            for (ulong k=0;  k<nh;  ++k)  d[k] = (ldn-2) - d[k];
        }
```

```
     0:  ...... 0    0    ...1..  2        32:  1.1... 2   40    .1....  4
     1:  ...1.. 1    4    ....1.  1        33:  111... 3   56    ..1...  3
     2:  ...11. 2    6    ...1..  2        34:  11.... 2   48    ...1..  2
     3:  ....1. 1    2    ..1...  3        35:  11.1.. 3   52    ....1.  1
     4:  ..1.1. 2   10    ...1..  2        36:  11.11. 4   54    ...1..  2
     5:  ..111. 3   14    ....1.  1        37:  11..1. 3   50    ..1...  3
     6:  ..11.. 2   12    .....1  0        38:  111.1. 4   58    ...1..  2
     7:  ..11.1 3   13    ....1.  1        39:  11111. 5   62    ....1.  1
     8:  ..1111 4   15    ...1..  2        40:  1111.. 4   60    .....1  0
     9:  ..1.11 3   11    ..1...  3        41:  1111.1 5   61    ....1.  1
    10:  ....11 2    3    ...1..  2        42:  111111 6   63    ...1..  2
    11:  ...111 3    7    ....1.  1        43:  111.11 5   59    ..1...  3
    12:  ...1.1 2    5    ...1..  2        44:  11..11 4   51    ...1..  2
    13:  .....1 1    1    ..1...  3        45:  11.111 5   55    ....1.  1
    14:  ..1..1 2    9    .1....  4        46:  11.1.1 4   53    ...1..  2
    15:  .11..1 3   25    ..1...  3        47:  11...1 3   49    ..1...  3
    16:  .1...1 2   17    ...1..  2        48:  111..1 4   57    .1....  4
    17:  .1.1.1 3   21    ....1.  1        49:  1.1..1 3   41    ..1...  3
    18:  .1.111 4   23    ...1..  2        50:  1....1 2   33    ...1..  2
    19:  .1..11 3   19    ..1...  3        51:  1..1.1 3   37    ....1.  1
    20:  .11.11 4   27    ...1..  2        52:  1..111 4   39    ...1..  2
    21:  .11111 5   31    ....1.  1        53:  1...11 3   35    ..1...  3
    22:  .111.1 4   29    .....1  0        54:  1.1.11 4   43    ...1..  2
    23:  .111.. 3   28    ....1.  1        55:  1.1111 5   47    ....1.  1
    24:  .1111. 4   30    ...1..  2        56:  1.11.1 4   45    .....1  0
    25:  .11.1. 3   26    ..1...  3        57:  1.11.. 3   44    ....1.  1
    26:  .1..1. 2   18    ...1..  2        58:  1.111. 4   46    ...1..  2
    27:  .1.11. 3   22    ....1.  1        59:  1.1.1. 3   42    ..1...  3
    28:  .1.1.. 2   20    ...1..  2        60:  1...1. 2   34    ...1..  2
    29:  .1.... 1   16    ..1...  3        61:  1..11. 3   38    ....1.  1
    30:  .11... 2   24    .1....  4        62:  1..1.. 2   36    ...1..  2
    31:  ..1... 1    8    1.....  5        63:  1..... 1   32   [1.....  5]
```

Figure 7.3: The (essentially unique) AC Gray code for $n = 6$. While the path is a cycle in the graph the AC condition does not hold for the transition from the last to the first word. No AC Gray code for $n = 7$ exists.

```
        for (ulong k=0,j=n-2;  k<j;  ++k,--j)  d[j] = d[k];
        d[nh-1] = ldn - 1;
    }
    else
    {
        for (ulong k=nh-2,j=nh-1;  0!=j;  --k,--j)  d[j] = d[k] + 1;
        for (ulong k=2,j=n-2;  k<j;  ++k,--j)  d[j] = d[k];
        d[0] = 0;
        d[nh] = 0;
    }
}
```

The Gray code is computed via

```
void
ac_gray(ulong *g, ulong ldn)
// Create an AC Gray code.
{
    ulong n = 1UL<<ldn;
    ALLOCA(uchar, d, n);
    ac_gray_delta(d, ldn);
    delta2gray(d, ldn, g);
}
```

where the routine `delta2gray()` is given in [FXT: file comb/delta2gray.cc]:

```
void
delta2gray(const unsigned char *d, ulong ldn, ulong *g, ulong g0/*=0*/)
// Fill into g[0..N-1] (N=2**ldn) the Gray path
//  corresponding to the delta sequence d[0..N-2].
{
    g[0] = g0;
    ulong n = 1UL << ldn;
    for (ulong k=0; k<n-1; ++k)  g[k+1] = g[k] ^ (1UL << d[k]);
}
```

The program [FXT: demo comb/acgray-demo.cc] can be used to create AC Gray codes for $n \le 6$. For $n \ge 7$ the algorithm produces near-AC Gray codes, where the number of non-AC transitions equals $2^{n-5} - 1$ for odd values of n and $2^{n-5} - 2$ for n even:

```
 # non-AC transitions:
   n =0..6  #non-ac = 0
   n =  7   #non-ac = 3
   n =  8   #non-ac = 6
   n =  9   #non-ac = 15
   n = 10   #non-ac = 30
   n = 11   #non-ac = 63
   n = 12   #non-ac = 126
   ...
```

It seems likely that near-AC Gray codes with fewer non-AC transitions exist.

7.4 Sorting the edges: lucky paths

The order of the nodes in the representation of the graph does not matter with finding paths as the algorithm does at no point refer to it. The order of the outgoing edges, however, *does* matter.

Edge sorting

Consider a large graph that has only a few paths. The calling tree of the recursive function `next_path()` obviously depends on the edge order. Thereby the first path can appear earlier or later in the search. 'Later' may well mean that the path is not found within any reasonable amount of time.

With a bit of luck one might find an ordering of the edges of the graph that will shorten the time span until the first path is found. The demo [FXT: demo graph/graph-monotonicgray-demo.cc] searches for monotonic Gray codes and allows to sort the edges of the graph. The method [FXT: `digraph::sort_edges()` in graph/digraph.cc] sorts the outgoing edges of each node according to a supplied comparison function.

The comparison function actually used imposes the lexicographic order shown in section 1.29 on page 59:

```
int my_cmp(const ulong &a, const ulong &b)
{
    if ( a==b )  return  0;
#define CODE(x) lexrev2negidx(x);
    ulong ca = CODE(a);
    ulong cb = CODE(b);
    return (ca<cb ? +1 : -1);
}
```

The choice was inspired by the observation that the bit-wise difference of successive elements in bit-lex order is either one or three.

We search until the first path for 8-bit words is found: For the unsorted graph this task takes 1.14 seconds, for the sorted it takes 0.03 seconds.

Sometimes, depending on the structure of the graph, paths tend to occur in 'clusters': bursts of many paths appear between longer periods of searching with no result. Finding the first F paths takes the following amounts of time (in seconds):

F	unsorted	sorted
1	1.14	0.03
10	5.76	0.07
100	14.1	0.22
1000	35.1	2.65

Lucky paths

The the first Gray code found in the hypercube graph with randomized edge order is shown in figure 7.4 on the facing page (left). The corresponding path (as reported by [FXT: `digraph_paths::print_turns()` in graph/digraphpaths.cc]) is described in the right column. Here `nn` is the number of neighbors of `node`, `xe` is the index of the neighbor (`next`) in the list of edges of `node`. Finally `xf` is the index among the

```
 0:  .... 0    0    1...  3          step: node -> next [xf xe / nn]
 1:  1... 1    8    ..1.  1             0:    0 ->   8  [ 0  0 /  4]
 2:  1.1. 2   10    .1..  2             1:    8 ->  10  [ 0  0 /  4]
 3:  111. 3   14    ...1  0             2:   10 ->  14  [ 0  0 /  4]
 4:  1111 4   15    1...  3             3:   14 ->  15  [ 0  0 /  4]
 5:  .111 3    7    .1..  2             4:   15 ->   7  [ 0  0 /  4]
 6:  ..11 2    3    ...1  0             5:    7 ->   3  [ 0  1 /  4]
 7:  ..1. 1    2    .1..  2             6:    3 ->   2  [ 1  2 /  4]
 8:  .11. 2    6    ..1.  1             7:    2 ->   6  [ 0  3 /  4]
 9:  .1.. 1    4    1...  3             8:    6 ->   4  [ 0  0 /  4]
10:  11.. 2   12    ...1  0             9:    4 ->  12  [ 1  3 /  4]
11:  11.1 3   13    1...  3            10:   12 ->  13  [ 0  0 /  4]
12:  .1.1 2    5    .1..  2            11:   13 ->   5  [ 0  1 /  4]
13:  ...1 1    1    1...  3            12:    5 ->   1  [ 0  3 /  4]
14:  1..1 2    9    ..1.  1            13:    1 ->   9  [ 0  2 /  4]
15:  1.11 3   11   [1.11  -]           14:    9 ->  11  [ 0  0 /  4]
                                    Path: #non-first-free turns = 2
```

Figure 7.4: A Gray code in the hypercube graph with randomized edge order (left) and the path description (right, see text).

free nodes in the list. The latter corresponds to the value `fct-1` in the function `next_path()` given on page 236.

If `xf` equals zero at some step then the first free neighbor was visited next. If `xf` is nonzero then a dead end was reached in the course of the search and there was a U-turn. If the path is not the first found then the U-turn might well correspond to a previous path.

If there was no U-turn then the number of *non-first-free* turns is zero (the number is given as the last line of the report). If it is zero we call the path found a *lucky path*. For each given ordering of the edges and each starting position of the search there is either at most one lucky path and if there is, it is the first path found.

When the first path is a lucky path then the search effectively 'falls through': The number of operations is a constant times the number of edges. That is, if a lucky path exists it is found almost immediately even for huge graphs.

7.5 Gray codes for Lyndon words

We search Gray codes for n-bit binary Lyndon words where n is a prime. Here is a Gray code for the 5-bit Lyndon words that is a cycle:

```
...1
..11
.111
.1111
..111
..1.1
```

An important application of such Gray codes is the construction of so-called *single track Gray codes* which can be obtained by appending rotated versions of the block. The following is a single track Gray code based on the block given. At each stage, the block is rotated by two positions (horizontal format):

```
######  --##--  -####-  ------  ---###
-####-  ------  ---###  ######  --##--
---###  ######  --##--  -####-  ------
--##--  -####-  ------  ---###  ######
------  ---###  ######  --##--  -####-
```

The *transition count* (the number of zero-one transitions) is by construction the same for each track. The all-zero and the all-one words are missing in the Gray code, its length equals $2^n - 2$.

```
       k :  [ node]  lyn_dec lyn_bin     diff  delta
       0 :  [    0]        1  ......1   ......1  0
       1 :  [    1]        3  .....11   .....1.  1
       2 :  [    3]        7  ....111   ....1..  2
       3 :  [    7]       15  ...1111   ...1...  3
       4 :  [   13]       31  ..11111   ..1....  4
       5 :  [   11]       27  ..11.11   ....1..  2
       6 :  [    8]       19  ..1..11   ...1...  3
       7 :  [   10]       23  ..1.111   ....1..  2
       8 :  [   16]       55  .11.111   .1.....  5
       9 :  [   17]       63  .111111   ...1...  3
      10 :  [   15]       47  .1.1111   ..1....  4
      11 :  [   14]       43  .1.1.11   ....1..  2
      12 :  [    5]       11  ...1.11   .1.....  5
      13 :  [    4]        9  ...1..1   .....1.  1
      14 :  [    6]       13  ...11.1   ....1..  2
      15 :  [   12]       29  ..111.1   ..1....  4
      16 :  [    9]       21  ..1.1.1   ...1...  3
      17 :  [    2]        5  ....1.1   ..1....  4
```

Figure 7.5: A Gray code through the length-7 binary Lyndon words.

7.5.1 Graph search with edge sorting

Gray codes for the 7-bit binary Lyndon words like the one shown in figure 7.5 can easily be found by a graph search. In fact, all of them can be generated in short time: For $n = 7$ there are 395 Gray codes (starting with the word `0000..001`) of which 112 are cycles.

```
       k :  [ node]  lyn_dec lyn_bin #rot rot(lyn)     diff  delta
       0 :  [    0]        1  ......1  0   ......1   ......1  0
       1 :  [    1]        3  .....11  0   .....11   .....1.  1
       2 :  [    3]        7  ....111  0   ....111   ....1..  2
       3 :  [    7]       15  ...1111  0   ...1111   ...1...  3
       4 :  [   13]       31  ..11111  0   ..11111   ..1....  4
       5 :  [   17]       63  .111111  0   .111111   .1.....  5
       6 :  [   15]       47  .1.1111  0   .1.1111   ..1....  4
       7 :  [   10]       23  ..1.111  1   .1.111.   ......1  0
       8 :  [   16]       55  .11.111  1   11.111.   1......  6
       9 :  [   11]       27  ..11.11  2   11.11..   .....1.  1
      10 :  [    5]       11  ...1.11  2   .1.11..   1.....  6
      11 :  [   14]       43  .1.1.11  2   .1.11.1   ......1  0
      12 :  [    6]       13  ...11.1  0   ...11.1   .1.....  5
      13 :  [   12]       29  ..111.1  0   ..111.1   ..1....  4
      14 :  [    8]       19  ..1..11  3   ..11..1   ....1..  2
      15 :  [    4]        9  ...1..1  0   ...1..1   ..1....  4
      16 :  [    9]       21  ..1.1.1  3   .1.1..1   .1.....  5
      17 :  [    2]        5  ....1.1  3   .1.1...   ......1  0
```

Figure 7.6: A Gray code through the length-7 binary Lyndon words found by searching through all 7-bit binary words.

The search for such a path for the next prime, $n = 11$, does not seem to give a result in reasonable time. If we do not insist on a Gray code through the cyclic minima but arbitrary rotations of the Lyndon words then more Gray codes exist. For that purpose nodes are declared adjacent if there is any cyclic rotation of the second node's value that differs in exactly one bit to the first node's value. The cyclic rotations can be recovered easily after a path is found. This is done in [FXT: demo graph/graph-lyndongray-demo.cc] whose output is shown in figure 7.6. Still, already for $n = 11$ we do not get a result. As the corresponding graph has 186 nodes and 1954 edges, this is not a surprise.

Let us try edge sorting! We sort the edges according to

```
int lyndon_cmp0(const ulong &a, const ulong &b)
{
    int bc = bit_count_cmp(a, b);
    if ( bc )  return  -bc;  // more bits first
    else
    {
        if ( a==b )  return 0;
        return  (a>b ?  +1 : -1);  // bigger numbers last
```

```
    }
}
```

where `bit_count_cmp()` is defined in [FXT: file bits/bitcount.h]:

```
static inline int bit_count_cmp(const ulong &a, const ulong &b)
{
    ulong ca = bit_count(a);
    ulong cb = bit_count(b);
    return  ( ca==cb ? 0 : (ca>cb ? +1 : -1) );
}
```

```
      k :  [ node]  lyn_dec       lyn_bin #rot      rot(lyn)           diff  delta
      0 :  [    0]        1  ...........1  0   ...........1   ...........1  0
      1 :  [    1]        3  ..........11  0   ..........11   ..........1.  1
      2 :  [    3]        7  .........111  0   .........111   .........1..  2
      3 :  [    7]       15  ........1111  0   ........1111   ........1...  3
      4 :  [   15]       31  .......11111  0   .......11111   .......1....  4
      5 :  [   31]       63  ......111111  0   ......111111   ......1.....  5
      6 :  [   63]      127  .....1111111  0   .....1111111   .....1......  6
      7 :  [  125]      255  ....11111111  0   ....11111111   ....1.......  7
      8 :  [  239]      511  ...111111111  0   ...111111111   ...1........  8
      9 :  [  417]     1023  ..1111111111  0   ..1111111111   ..1.........  9
     10 :  [  589]     2047  .11111111111  0   .11111111111   .1.......... 10
     11 :  [  629]     4095  .111111111111  0  .111111111111  .1.......... 11
     12 :  [  618]     3071  .1.1111111111  0  .1.1111111111  ..1......... 10
     13 :  [  514]     1535  ..1.111111111  1  .1.111111111.  ...........1  0
     14 :  [  624]     3583  .11.11111111  1   11.11111111.   1........... 12
     15 :  [  550]     1791  ..11.11111111  2   11.11111111..  ..........1.  1
     16 :  [  626]     3839  .111.11111111  2   11.11111111.1  ...........1  0
     17 :  [  567]     1919  ..111.1111111  3   11.1111111..1  .........1..  2
     18 :  [  627]     3967  .1111.1111111  3   11.1111111.11  ..........1.  1
     19 :  [  576]     1983  ..1111.111111  4   11.111111..11  ........1...  3
     20 :  [  628]     4031  .11111.111111  4   11.111111.111  .........1..  2
     21 :  [  581]     2015  ..11111.11111  5   11.11111..111  .......1....  4
     22 :  [  404]      991  ...1111.11111  5   11.11111...11  .........1..  2
     23 :  [  614]     3039  .1.1111.11111  5   11.11111.1.11  ........1...  3
     24 :  [  508]     1519  ..1.1111.1111  6   11.1111..1.11  ......1.....  5
     25 :  [  584]     2031  ..111111.1111  6   11.1111..1111  .........1..  2
      [--snip--]
    615 :  [    4]        9  .........1..1  5   ....1..1.....  ..1.......... 10
    616 :  [   36]       73  ......1..1..1  2   ....1..1..1..  .........1..  2
    617 :  [   32]       65  ......1.....1  2   ....1.....1..  .......1.....  5
    618 :  [   33]       67  ......1....11  2   ....1....11..  ........1...  3
    619 :  [  153]      323  ....1.1....11  2   ..1.1....11..  ..1.......... 10
    620 :  [   65]      133  .....1....1.1  8   ..1.1.....1..  ........1...  3
    621 :  [  154]      325  ....1.1...1.1  2   ..1.1...1.1..  .......1....  4
    622 :  [   79]      161  .....1.1....1 10   ..1.....1.1..  ....1.......  8
    623 :  [   16]       33  .......1....1 10   ..1.......1..  .......1....  4
    624 :  [  126]      265  ....1....1..1  2   ..1....1..1..  ......1.....  5
    625 :  [  145]      305  ....1..11...1 10   ..1....1..11.  ..........1.  1
    626 :  [  130]      273  ....1...1...1 10   ..1....1...1.  .........1..  2
    627 :  [  188]      401  ....11..1...1 10   ..1....11..1.  .......1....  4
    628 :  [   71]      145  .....1..1...1 10   ..1.....1..1.  ......1.....  5
    629 :  [    8]       17  ........1...1 10   ..1........1.  .......1....  4
```

Figure 7.7: Begin and end of a Gray cycle through the 13-bit binary Lyndon words.

We find a Gray code (which also is a cycle) for $n = 11$ immediately. And also for $n = 13$, again a cycle. The graph for $n = 13$ has 630 nodes and 8,056 edges, so finding a path is quite unexpected. The cycle found starts and ends as shown in figure 7.7. Stopping only after the tenth path needs essentially the same time, clustering.

For next candidate ($n = 17$) we do no find a Gray code within many hours of search. No surprise for a graph with 7,710 nodes and 130,828 edges.

We try another edge sorting scheme, an order based on the binary Gray code:

```
int lyndon_cmp2(const ulong &a, const ulong &b)
{
    if ( a==b )  return 0;
#define CODE(x) gray_code(x)
    ulong ta = CODE(a), tb = CODE(b);
    return  ( ta<tb ? +1 : -1);
#undef CODE
```

```
}
```

There we go, we find a cycle for $n = 17$ and all smaller primes. All are cycles and all paths are lucky paths.

The following edge sorting scheme also leads to Gray codes for all prime n where $3 \le n \le 17$:

```
int lyndon_cmp3(const ulong &a, const ulong &b)
{
    if ( a==b )  return 0;
#define CODE(x) inverse_gray_code(x)
    ulong ta = CODE(a), tb = CODE(b);
    return  ( ta<tb ? +1 : -1);
#undef CODE
}
```

And also for $n = 19$: 27,594 nodes, 523,978 edges, great. The result appears within a split second. Printing it on screen takes quite a bit longer. We obviously cannot expect to get a result for $n = 23$ (364,722 nodes, 8,388,142 edges). But we do. Surprise. The sorting scheme leads to cycles for all odd $n \le 25$, all paths are lucky paths.

The next candidate ($n = 29$) is out of reach, the graph alone (18,512,790 nodes) would need approximately 2 Gigabytes of RAM. Really? All cycles produced with `lyndon_cmp3()` are lucky paths and we can exploit that fact.

7.5.2 An optimized algorithm

With edge sorting functions that lead to a lucky path we can discard most of of the data used with graph searching. We only need to keep track of whether a node has been visited so far. A tag-array (bit-array, see section 4.6 on page 145) suffices.

With n-bit Lyndon words the amount of tag-bits needed is 2^n. Find an implementation of the algorithm as [FXT: `class lyndon_gray` in graph/lyndongray.h].

If only the cyclical minima of the values are tagged then only $2^n/2$ bits are needed if the access to the only necklace consisting of all ones is treated separately. This variant of the algorithm is activated by uncommenting the line `#define ALT_ALGORITM`. Noting that the lowest bit in a necklace is always one we need only $2^n/4$ bits: Simply shift the words right by one before testing or writing to the tag array. This can be achieved by additionally uncommenting the line `#define ALTALT` in the file.

When a node is visited the algorithm creates a table of neighbors and selects the minimum among the free nodes with respect to the edge sorting function used. The table of neighbors is discarded then in order to minimize memory usage.

When no neighbor is found the number of nodes visited so far is returned. If this number equals the number of $n-$bit Lyndon words a lucky path was found. With composite n the algorithm allows to find Gray codes through all $n-$bit necklaces with the exception of the all-ones and the all-zeros word.

Four flavors of the algorithm have been found so far, corresponding to edge sorting with the 3rd, 5th, 21th and 29th power of the Gray code. All of these lead to cycles for all primes $n \le 31$. The resources needed with larger values of n are shown in figure 7.8 on the next page.

Using a 64-bit machine equipped with more than 4 Gigabyte of RAM it can be verified that three of the edge sorting functions lead to a Gray cycles also for $n = 37$, the 3rd power version fails. One of the sorting functions *might* lead to a Gray code for $n = 41$.

A demo is [FXT: demo graph/lyndongray-demo.cc], four arguments can be given:

```
arg 1: 13 == n  [ a prime < BITS_PER_LONG ]  default=17
arg 2: 1 == wh  [printing: 0==>none, 1==>delta seq., 2==>full output]  default=1
arg 3: 3 == ncmp  [use comparison function (0,1,2,3)]  default=2
arg 4: 0 == testall  [special: test all primes <= value]  default=0
```

n	number of nodes	tag-size	approx. time
23	364,722	0.25 MB	1 sec
(25)	1,342,182	1 MB	3 sec
(27)	4,971,066	4 MB	12 sec
29	18,512,790	16 MB	1 min
31	69,273,666	64 MB	4 min
(33)	260,301,174	256 MB	16 min
(35)	981,706,830	16 GB	1 h
37	3,714,566,310	4 GB	7 h
(39)	14,096,303,342	16 GB	2 d
41	53,634,713,550	64 GB	10 d
43	204,560,302,842	256 GB	>20 d

Figure 7.8: Memory and (approximate) time needed for computing Gray codes with n-bit Lyndon words. The number of nodes equals the number of length-n necklaces minus two. The size of the tag array equals $2^n/4$ bits or $2^n/32$ bytes.

```
% ./bin 7 2 0  # 7 bits, full output, comparison function 0
n = 7   #lyn = 18
   1:    ......1   0  ......1   ......1  0
   2:    ...1..1   0  ...1..1   ...1...  3
   3:    ..1..11   3  ..11..1   ..1....  4
   4:    ..1.111   3  .111..1   .1.....  5
   5:    .1.1111   2  .1111.1   ....1..  2
   6:    .1.1.11   2  .1.11.1   ..1....  4
   7:    .11.111   5  11.11.1   1......  6
   8:    .111111   2  11111.1   ..1....  4
   9:    ..11111   2  11111..   ......1  0
  10:    ..111.1   2  111.1..   ...1...  3
  11:    ..1.1.1   2  1.1.1..   .1.....  5
  12:    ....1.1   2  ..1.1..   1......  6
  13:    ...1.11   1  ..1.11.   .....1.  1
  14:    ..11.11   1  .11.11.   .1.....  5
  15:    ...11.1   2  .11.1..   .....1.  1
  16:    ...1111   2  .1111..   ...1...  3
  17:    ....111   2  ..111..   .1.....  5
  18:    .....11   2  ...11..   ..1....  4
last = .....11     crc=0b14a5846c41d57f
n = 7   #lyn = 18   #= 18
```

Figure 7.9: A Gray code for 7-bit Lyndon words.

An example with full output is given in figure 7.9 on the preceding page. A 64-bit CRC [FXT: `class crc64` in bits/crc64.h] (see section 1.33 on page 69) is computed from the delta sequence (rightmost column) and printed with the last word.

```
 % ./bin 13 1 2  # 13 bits, delta seq. output, comparison function 2
  n = 13   #lyn = 630
  06B57458354645962546436734A74684A106C0145120825747A745247AC8564567018A7654647484A756A546457CA1ACBC1C
  856BA9A64B974565486456596452194252155315BC82BC75BA02926256354267A462475A3ACB9761560C37412583758CA5624
  B8C6A6C6A87A9C20CBA45340420145405231290756976515631602042 30A7BA31C1485C6105201510490BCA891BA9B1B9AC0
  A9A89B898A565B878574586574784 5A9546702305A41275315458767465747A8457845470379A8586B0A76985787679767 59
  A976567686A567656A576B86581305A20AB0ACB0AB5352343823546532524756 3A432532A372354657643572373624634642
  45323974234352356532364232632352343275323423253969268532342325826424368236323463623584232423832423 27
  523242325323432642324235323423
  last = ..........11     crc=568dab04b55aa2fb
  n = 13   #lyn = 630   #= 630


% ./bin 13 1 3  # 13 bits, delta seq. output, comparison function 3
n = 13   #lyn = 630
  06B57458354645962546436735371CA8B1587BA7610635285A0C2484B9713476B689A897AC98768968B9A106326016261050
  1424B8979A78987B97898C98921941315313698314281687BCB9469C489C6210205B050A1A7A4568A9BC5CB79AB647B74812
  0AB30BC1A131ACB120B0164CA1CABA121ABACA2B0BACAB184578678498958486764 6A84561916546947457875458654901 37
  4020103101210427017121650745 7B854606C16BC523801365164130164BC7987A09872CBA9A87A20B787AC9B7CBA834C0C1
  3C341C1042010C14C01C414587854645A854C95035A6A9570A9756586B9B5969580A0872C3123B0CB316BC6C0B21B2C0C2C0
  5301C0530CB1C1530C01CB0BC20CBC0CB1C875657568 65A75A65A40898A898B91CA898A8B898A81BC8A9ACA989AB817A9BC1
  BA9ABA9CA9AB918A1CACBAC9BCB0BC
  last = ..........11     crc=745def277b1fbed0
  n = 13   #lyn = 630   #= 630
```

Figure 7.10: Delta sequences for two different Gray codes for 13-bit Lyndon words. Note that both start identically.

```
    % ./bin 29 0 0  # 29 bits, output=progress, comparison function 0
    n = 29   #lyn = 18512790
    .............    1048576 ( 5.66406 % )   crc=ceabc5f2056be699
    .............    2097152 ( 11.3281 % )   crc=76dd94f1a554b50d
    .............    3145728 ( 16.9922 % )   crc=6b39957f1e141f4d
    .............    4194304 ( 22.6563 % )   crc=53419af1f1185dc0
    .............    5242880 ( 28.3203 % )   crc=45d45b193f8ee566
    .............    6291456 ( 33.9844 % )   crc=95a24c824f56e196
    .............    7340032 ( 39.6484 % )   crc=003ee5af5b248e34
    .............    8388608 ( 45.3125 % )   crc=23cb74d3ea0c4587
    .............    9437184 ( 50.9766 % )   crc=896fd04c87dd0d43
    .............   10485760 ( 56.6406 % )   crc=b00d8c899f0fc791
    .............   11534336 ( 62.3047 % )   crc=d148f1b95b23eeab
    .............   12582912 ( 67.9688 % )   crc=82971e2ed4863050
    .............   13631488 ( 73.6328 % )   crc=f249ad5b4fed252d
    .............   14680064 ( 79.2969 % )   crc=909821d0c7246a98
    .............   15728640 ( 84.9609 % )   crc=1c5d68e38e55b3ca
    .............   16777216 ( 90.625 % )    crc=0e64f82c67c79cf1
    .............   17825792 ( 96.2891 % )   crc=62c17b9f3c644396
    .........
    last = ........................11     crc=5736fc9365da927e
    n = 29   #lyn = 18512790   #= 18512790
```

Figure 7.11: Computation of a Gray code through the 29-bit Lyndon words. Most output is suppressed, only the CRC is printed at certain checkpoints.

For larger n one might want to print only the delta sequence, this is shown in figure 7.10. The CRC allows to conveniently determine whether two delta sequences are different. Different sequences sometimes start identically.

For still larger values of n even the delta sequence tends to get huge (e.g., with $n = 37$ the sequence would be approximately 3.7 GB). One can suppress all output except for a progress indication, shown in figure 7.11. Here the CRC checksum is updated only with every (cyclically unadjusted) 2^{16}-th Lyndon word.

Part II

Fast orthogonal transforms

Chapter 8

The Fourier transform

This chapter introduces the discrete Fourier transform and algorithms for it's fast computation. Implementations and optimization considerations for complex and real valued transforms are given. The fast Fourier transforms (FFTs) are the basis of the algorithms for fast convolution (given in chapter 9) which are in turn the core of the fast high precision multiplication routines described in chapter 14. The number theoretic Fourier transforms (NTTs) are treated separately in chapter 13.

8.1 The discrete Fourier transform

The *discrete Fourier transform* (DFT or simply FT) of a complex sequence $a = [a_0, a_1, \ldots, a_{n-1}]$ of length n is the complex sequence $c = [c_0, c_1, \ldots, c_{n-1}]$ defined by

$$c = \mathcal{F}\big[\, a \,\big] \tag{8.1a}$$

$$c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \, z^{+x\,k} \quad \text{where} \quad z = e^{2\,\pi\,i/n} \tag{8.1b}$$

z is an n-th root of unity: $z^n = 1$.

Back-transform (or *inverse discrete Fourier transform*, IDFT or simply IFT) is then

$$a = \mathcal{F}^{-1}\big[\, c \,\big] \tag{8.2a}$$

$$a_x := \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} c_k \, z^{-x\,k} \tag{8.2b}$$

To see this, consider element y of the IFT of the FT of a:

$$\mathcal{F}^{-1}\big[\, \mathcal{F}\big[\, a \,\big] \,\big]_y = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} (a_x \, z^{x\,k}) \, z^{-y\,k} \tag{8.3a}$$

$$= \frac{1}{n} \sum_x a_x \sum_k (z^{x-y})^k \tag{8.3b}$$

Now $\sum_k (z^{x-y})^k = n$ for $x = y$ and zero else. This is because z is an n-th root of unity: with $x = y$ the sum consists of n times $z^0 = 1$, with $x \neq y$ the summands lie on the unit circle (on the vertices of an equilateral polygon with center zero) and add up to zero. Therefore the whole expression is equal to

$$\frac{1}{n}\, n \sum_x a_x \, \delta_{x,y} = a_y \tag{8.4a}$$

where

$$\delta_{x,y} \quad = \quad \begin{cases} 1 & (x = y) \\ 0 & (x \neq y) \end{cases} \tag{8.4b}$$

Here we will call the FT with the plus in the exponent the forward transform. The choice is actually arbitrary[1]. The sign in the exponent is called the *sign of the transform*.

The FT is a linear transform. That is, for $\alpha, \beta \in \mathbb{C}$

$$\mathcal{F}\big[\,\alpha\, a + \beta\, b\,\big] \quad = \quad \alpha\, \mathcal{F}\big[\,a\,\big] + \beta\, \mathcal{F}\big[\,b\,\big] \tag{8.5}$$

Further *Parseval's equation* holds, the sum of squares of the absolute values is identical for a sequence and its Fourier transform:

$$\sum_{x=0}^{n-1} |a_x|^2 \quad = \quad \sum_{k=0}^{n-1} |c_k|^2 \tag{8.6}$$

A straightforward implementation of the discrete Fourier transform, that is, the computation of n sums each of length n, requires $\sim n^2$ operations.

```
void slow_ft(Complex *f, long n, int is)
{
    Complex h[n];
    const double ph0 = is*2.0*M_PI/n;
    for (long w=0; w<n; ++w)
    {
        Complex t = 0.0;
        for (long k=0; k<n; ++k)
        {
            t +=  f[k] * SinCos(ph0*k*w);
        }
        h[w] = t;
    }
    copy(h, f, n);
}
```

This is [FXT: `slow_ft()` in fft/slowft.cc]. The variable `is`$= \pm 1$ is the sign of the transform, the function `SinCos(x)` returns a `Complex(cos(x), sin(x))`.

Note that the normalization factor $\frac{1}{\sqrt{n}}$ in front of the FT sums has been left out. The inverse of the transform with one sign is the transform with the opposite sign followed by a multiplication of each element by $\frac{1}{n}$. One has to keep in mind if that the sum of squares of the original sequence and its transform are equal up to a factor $1/\sqrt{n}$.

A *fast Fourier transform* (FFT) algorithm is an algorithm that improves the operation count to proportional $n \sum_{k=1}^{m}(p_k - 1)$, where $n = p_1 p_2 \cdots p_m$ is a factorization of n. In case of a power $n = p^m$ the value computes to $n\,(p-1)\,\log_p(n)$. In the special case $p = 2$ even $n/2\,\log_2(n)$ (complex) multiplications suffice. There are several different FFT algorithms with many variants.

8.2 Summary of definitions of Fourier transforms *

This section summarizes the definitions of the continuous, semi-continuous and the discrete Fourier transform.

[1] Engineers seem to prefer the minus for the forward transform, mathematicians the plus.

The continuous Fourier transform

The (continuous) *Fourier transform* (FT) of a function $f : \mathbb{C}^n \to \mathbb{C}^n, \quad \vec{x} \mapsto f(\vec{x})$ is defined by

$$F(\vec{\omega}) \quad := \quad \frac{1}{\left(\sqrt{2\,\pi}\right)^n} \int_{\mathbb{C}^n} f(\vec{x})\, e^{+\sigma\, i\, \vec{x}\, \vec{\omega}} d^n x \tag{8.7}$$

where $\sigma = \pm 1$ is the sign of the transform. The FT is a unitary transform. Its inverse is the complex conjugate transform

$$f(\vec{x}) \quad = \quad \frac{1}{\left(\sqrt{2\,\pi}\right)^n} \int_{\mathbb{C}^n} F(\vec{\omega})\, e^{-\sigma\, i\, \vec{x}\, \vec{\omega}} d^n \omega \tag{8.8}$$

For the 1-dimensional case one has

$$F(\omega) \quad = \quad \frac{1}{\sqrt{2\,\pi}} \int_{-\infty}^{+\infty} f(x) e^{+\sigma\, i\, x\, \omega} dx \tag{8.9}$$

$$f(x) \quad = \quad \frac{1}{\sqrt{2\,\pi}} \int_{-\infty}^{+\infty} F(\omega)\, e^{-\sigma\, i\, x\, \omega} d\omega \tag{8.10}$$

The 'frequency-form' is

$$F(\nu) \quad = \quad \int_{-\infty}^{+\infty} f(x) e^{+2\,\pi\, \sigma\, i\, x\, \nu} dx \tag{8.11}$$

$$f(x) \quad = \quad \int_{-\infty}^{+\infty} F(\nu)\, e^{-2\,\pi\, \sigma\, i\, x\, \nu} d\nu \tag{8.12}$$

The semi-continuous Fourier transform

For periodic functions defined on a interval $L \in \mathbb{R}$, $f : L \to \mathbb{R},\ x \mapsto f(x)$ one defines the *semi-continuous Fourier transform* as:

$$c_k \quad := \quad \frac{1}{\sqrt{L}} \int_L f(x)\, e^{+\sigma\, 2\,\pi\, i\, k\, x/L} dx \tag{8.13}$$

Then

$$\frac{1}{\sqrt{L}} \sum_{k=-\infty}^{k=+\infty} c_k\, e^{-\sigma\, 2\,\pi\, i\, k\, x/L} \quad = \quad \begin{cases} f(x) & \text{if } f \text{ continuous at } x \\ \frac{f(x+0)+f(x-0)}{2} & \text{else} \end{cases} \tag{8.14}$$

An equivalent form is given by

$$a_k \quad := \quad \frac{1}{\sqrt{L}} \int_L f(x)\, \cos \frac{2\,\pi\, k\, x}{L} dx, \qquad k = 0, 1, 2, \ldots \tag{8.15a}$$

$$b_k \quad := \quad \frac{1}{\sqrt{L}} \int_L f(x)\, \sin \frac{2\,\pi\, k\, x}{L} dx, \qquad k = 1, 2, \ldots \tag{8.15b}$$

$$f(x) \quad = \quad \frac{1}{\sqrt{L}} \left[\frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k\, \cos \frac{2\,\pi\, k\, x}{L} + b_k\, \sin \frac{2\,\pi\, k\, x}{L} \right) \right] \tag{8.15c}$$

The connection between the two forms is given by:

$$c_k \quad = \quad \begin{cases} \frac{a_0}{2} & (k = 0) \\ \frac{1}{2}(a_k - i b_k) & (k > 0) \\ \frac{1}{2}(a_k + i b_k) & (k < 0) \end{cases} \tag{8.16}$$

The discrete Fourier transform

The *discrete Fourier transform* (DFT) of a sequence f of length n with elements f_x is defined by

$$c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} f_x \, e^{+\sigma \, 2 \, \pi \, i \, x \, k/n} \tag{8.17}$$

The inverse transform is

$$f_x = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} c_k \, e^{-\sigma \, 2 \, \pi \, i \, x \, k/n} \tag{8.18}$$

8.3 Radix-2 FFT algorithms

A little bit of notation

In what follows let a be a length-n sequence with n a power of two.

- Let $a^{(even)}$ and $a^{(odd)}$ denote the length-$n/2$ subsequences of those elements of a that have even and odd indices, respectively. That is, $a^{(even)} = [a_0, a_2, a_4, a_6, \ldots, a_{n-2}]$ and $a^{(odd)} = [a_1, a_3, \ldots, a_{n-1}]$.
- Let $a^{(left)}$ and $a^{(right)}$ denote the left and right subsequences, respectively. That is, $a^{(left)} = [a_0, a_1, \ldots, a_{n/2-1}$ and $a^{(right)} = [a_{n/2}, a_{n/2+1}, \ldots, a_{n-1}]$.
- Let $c = \mathcal{S}^k a$ denote the sequence with elements $c_x = a_x \, e^{k \, 2 \, \pi \, i \, x/n}$. The symbol $\mathcal{S}$ shall suggest a shift operator. With radix-2 FFT algorithms only $\mathcal{S}^{1/2}$ is needed.
- In relations between sequences we sometimes emphasize the length of the sequences on both sides as in $a^{(even)} \stackrel{n/2}{=} b^{(odd)} + c^{(odd)}$. In these relations the operators plus and minus are to be understood as element-wise.

8.3.1 Decimation in time (DIT) FFT

The following observation is the key to the (radix-2) *decimation in time* (DIT) FFT algorithm, also called *Cooley-Tukey* FFT algorithm:
For even values of n the k-th element of the Fourier transform is

$$\mathcal{F}\big[\, a \,\big]_k = \sum_{x=0}^{n-1} a_x \, z^{x \, k} = \sum_{x=0}^{n/2-1} a_{2\,x} \, z^{2\,x\,k} + \sum_{x=0}^{n/2-1} a_{2\,x+1} \, z^{(2\,x+1)\,k} \tag{8.19a}$$

$$= \sum_{x=0}^{n/2-1} a_{2\,x} \, z^{2\,x\,k} + z^k \sum_{x=0}^{n/2-1} a_{2\,x+1} \, z^{2\,x\,k} \tag{8.19b}$$

where $z = e^{\sigma \, i \, 2 \, \pi/n}$, $\sigma = \pm 1$ is the sign of the transform and $k \in \{0, 1, \ldots, n-1\}$.

The identity tells us how to compute the k-th element of the length-n Fourier transform from the length-$n/2$ Fourier transforms of the even and odd indexed subsequences.

To actually rewrite the length-n FT in terms of length-$n/2$ FTs one has to distinguish whether $0 \le k < n/2$ or $n/2 \le k < n$. In the expressions we rewrite $k \in \{0, 1, 2, \ldots, n-1\}$ as $k = j + \delta \, \frac{n}{2}$

where $j \in \{0, 1, 2, \ldots, n/2-1\}$ and $\delta \in \{0, 1\}$:

$$\sum_{x=0}^{n-1} a_x \, z^{x\,(j+\delta\,\frac{n}{2})} = \sum_{x=0}^{n/2-1} a_x^{(even)} \, z^{2\,x\,(j+\delta\,\frac{n}{2})} + z^{j+\delta\,\frac{n}{2}} \sum_{x=0}^{n/2-1} a_x^{(odd)} \, z^{2\,x\,(j+\delta\,\frac{n}{2})} \tag{8.20a}$$

$$= \begin{cases} \displaystyle\sum_{x=0}^{n/2-1} a_x^{(even)} \, z^{2\,x\,j} + z^j \sum_{x=0}^{n/2-1} a_x^{(odd)} \, z^{2\,x\,j} & \text{for} \quad \delta = 0 \\ \displaystyle\sum_{x=0}^{n/2-1} a_x^{(even)} \, z^{2\,x\,j} - z^j \sum_{x=0}^{n/2-1} a_x^{(odd)} \, z^{2\,x\,j} & \text{for} \quad \delta = 1 \end{cases} \tag{8.20b}$$

Observing that z^2 is just the root of unity that appears in a length-$n/2$ transform one can rewrite the last two equations to obtain the *radix-2 DIT FFT step*:

$$\mathcal{F}\big[\,a\,\big]^{(left)} \overset{n/2}{=} \mathcal{F}\big[\,a^{(even)}\,\big] + \mathcal{S}^{1/2}\mathcal{F}\big[\,a^{(odd)}\,\big] \tag{8.21a}$$

$$\mathcal{F}\big[\,a\,\big]^{(right)} \overset{n/2}{=} \mathcal{F}\big[\,a^{(even)}\,\big] - \mathcal{S}^{1/2}\mathcal{F}\big[\,a^{(odd)}\,\big] \tag{8.21b}$$

The length-n transform has been replaced by two transforms of length $n/2$. If n is a power of 2 this scheme can be applied recursively until length-one transforms are reached which are identity ('do nothing') operations.

Thereby the operation count is improved to proportional $n \cdot \log_2(n)$: There are $\log_2(n)$ splitting steps, the work in each step is proportional to n.

Recursive implementation

A recursive implementation of radix-2 DIT FFT given as pseudo code (C++ version in [FXT: file fft/recfft2.cc]) is

```
procedure rec_fft_dit2(a[], n, x[], is)
// complex a[0..n-1] input
// complex x[0..n-1] result
{
    complex b[0..n/2-1], c[0..n/2-1]   // workspace
    complex s[0..n/2-1], t[0..n/2-1]   // workspace

    if n == 1 then  // end of recursion
    {
        x[0] := a[0]
        return
    }

    nh := n/2

    for k:=0 to nh-1  // copy to workspace
    {
        s[k] := a[2*k]    // even indexed elements
        t[k] := a[2*k+1]  // odd  indexed elements
    }

    // recursion: call two half-length FFTs:
    rec_fft_dit2(s[],nh,b[],is)
    rec_fft_dit2(t[],nh,c[],is)

    fourier_shift(c[],nh,is*1/2)

    for k:=0 to nh-1  // copy back from workspace
    {
        x[k]    := b[k] + c[k];
        x[k+nh] := b[k] - c[k];
    }
}
```

The parameter `is` $= \sigma = \pm 1$ is the sign of the transform. The data length `n` must be a power of 2. The result is returned in the array `x[]`. Note that normalization (multiplication of each element of `x[]` by $1/\sqrt{n}$) is not included here.

The procedure uses the subroutine `fourier_shift` which modifies the array `c[]` according to the operation $\mathcal{S}^v$: each element `c[k]` is multiplied by $e^{v\,2\,\pi\,i\,k/n}$. It is called with $v = \pm 1/2$ for the Fourier transform. The pseudo code (C++ equivalent in [FXT: file fft/fouriershift.cc]) is

```
procedure fourier_shift(c[], n, v)
{
    for k:=0 to n-1
    {
        c[k] := c[k] * exp(v*2.0*PI*I*k/n)
    }
}
```

The recursive FFT-procedure involves $n \log_2(n)$ function calls, which can be avoided by rewriting it in a non-recursive way. One can even do all operations *in-place*, no temporary workspace is needed at all. The price is the necessity of an additional data reordering: The procedure `revbin_permute(a[],n)` rearranges the array `a[]` in a way that each element a_x is swapped with $a_{\tilde{x}}$, where $\tilde{x}$ is obtained from x by reversing its binary digits. Methods to do this are discussed in section 2.1.

Iterative implementation

Pseudo code for a non-recursive procedure of the radix-2 DIT algorithm (C++ implementation given in [FXT: file fft/fftdit2.cc]):

```
procedure fft_depth_first_dit2(a[], ldn, is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn  // length of a[] is a power of 2

    revbin_permute(a[],n)

    for ldm:=1 to ldn  // log_2(n) iterations
    {
        m  := 2**ldm
        mh := m/2

        for r:=0 to n-m step m  // n/m iterations
        {
            for j:=0 to mh-1  // m/2 iterations
            {
                e := exp(is*2*PI*I*j/m)  // log_2(n)*n/m*m/2 = log_2(n)*n/2 computations

                u := a[r+j]
                v := a[r+j+mh] * e

                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
        }
    }
}
```

This version of a non-recursive FFT procedure already avoids the calling overhead and it works in-place. But it is a bit wasteful. The (expensive) computation `e := exp(is*2*PI*I*j/m)` is done $n/2 \cdot \log_2(n)$ times.

Saving trigonometric computations

To reduce the number of trigonometric computations, one can simply swap the two inner loops, leading to the first 'real world' FFT procedure presented here. Pseudo code for a non-recursive procedure of the radix-2 DIT algorithm (C++ version in [FXT: file fft/fftdit2.cc]):

```
procedure fft_dit2(a[], ldn, is)
// complex a[0..2**ldn-1] input, result
{
```

```
    n := 2**ldn
    revbin_permute(a[],n)
    for ldm:=1 to ldn  // log_2(n) iterations
    {
        m  := 2**ldm
        mh := m/2
        for j:=0 to mh-1  // m/2 iterations
        {
            e := exp(is*2*PI*I*j/m)  // 1 + 2 + ... + n/8 + n/4 + n/2 == n-1 computations
            for r:=0 to n-m step m
            {
                u := a[r+j]
                v := a[r+j+mh] * e
                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
        }
    }
}
```

Swapping the two inner loops reduces the number of trigonometric computations to `n` but leads to a feature that many FFT implementations share: Memory access is highly nonlocal. For each recursion stage (value of `ldm`) the array is traversed `mh` times with `n/m` accesses in strides[2] of `mh`. As `mh` is a power of 2 this can (on computers that use memory cache) have a very negative performance impact for large values of `n`. On computers where the memory access is very slow compared to the CPU the naive version can actually outperform the other.

It is a good idea to extract the `ldm==1` stage of the outermost loop, this avoids complex multiplications with the trivial factors $1 + 0\,i$ (and the computations of these quantities as trigonometric functions). Replace

```
    for ldm:=1 to ldn
    {
```

by

```
    for r:=0 to n-1 step 2
    {
        {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
    }
    for ldm:=2 to ldn
    {
```

8.3.2 Decimation in frequency (DIF) FFT

Splitting of the Fourier sum into a left and right half leads to the *decimation in frequency* (DIF) FFT algorithm, also called *Sande-Tukey* FFT algorithm.
For even values of n the k-th element of the Fourier transform is

$$\mathcal{F}\big[\,a\,\big]_k = \sum_{x=0}^{n-1} a_x\, z^{x\,k} = \sum_{x=0}^{n/2-1} a_x\, z^{x\,k} + \sum_{x=n/2}^{n} a_x\, z^{x\,k} \tag{8.22a}$$

$$= \sum_{x=0}^{n/2-1} a_x\, z^{x\,k} + \sum_{x=0}^{n/2-1} a_{x+n/2}\, z^{(x+n/2)\,k} \tag{8.22b}$$

$$= \sum_{x=0}^{n/2-1} \left(a_x^{(left)} + z^{k\,n/2}\, a_x^{(right)}\right) z^{x\,k} \tag{8.22c}$$

where $z = e^{\sigma\, i\, 2\,\pi/n}$, $\sigma = \pm 1$ is the sign of the transform and $k \in \{0, 1, \ldots, n-1\}$.

[2] Stride-n memory access: consecutive accesses to addresses that are n units apart.

Here one has to distinguish whether k is even or odd. Therefore we rewrite $k \in \{0, 1, 2, \ldots, n-1\}$ as $k = 2\,j + \delta$ where $j \in \{0, 1, 2, \ldots, n/2-1\}$ and $\delta \in \{0, 1\}$:

$$\sum_{x=0}^{n-1} a_x\, z^{x\,(2\,j+\delta)} = \sum_{x=0}^{n/2-1} (a_x^{(left)} + z^{(2\,j+\delta)\,n/2}\, a_x^{(right)})\, z^{x\,(2\,j+\delta)} \tag{8.23a}$$

$$= \begin{cases} \sum_{x=0}^{n/2-1} (a_x^{(left)} + a_x^{(right)})\, z^{2\,x\,j} & \text{for} \quad \delta = 0 \\ \sum_{x=0}^{n/2-1} z^x (a_x^{(left)} - a_x^{(right)})\, z^{2\,x\,j} & \text{for} \quad \delta = 1 \end{cases} \tag{8.23b}$$

$z^{(2\,j+\delta)\,n/2} = e^{\pm\pi\,i\,\delta}$ is equal to plus or minus one for $\delta = 0$ or $\delta = 1$ corresponding to k even or odd. The last two equations are, more compactly written, the key to the *radix-2 DIF FFT step*:

$$\mathcal{F}\big[\,a\,\big]^{(even)} \stackrel{n/2}{=} \mathcal{F}\big[\,a^{(left)} + a^{(right)}\,\big] \tag{8.24a}$$

$$\mathcal{F}\big[\,a\,\big]^{(odd)} \stackrel{n/2}{=} \mathcal{F}\big[\,\mathcal{S}^{1/2}\left(a^{(left)} - a^{(right)}\right)\big] \tag{8.24b}$$

A recursive implementation of radix-2 DIF FFT given as pseudo code (C++ version given in [FXT: file fft/recfft2.cc]) is

```
procedure rec_fft_dif2(a[], n, x[], is)
// complex a[0..n-1] input
// complex x[0..n-1] result
{
    complex b[0..n/2-1], c[0..n/2-1]   // workspace
    complex s[0..n/2-1], t[0..n/2-1]   // workspace

    if n == 1 then
    {
        x[0] := a[0]
        return
    }

    nh := n/2

    for k:=0 to nh-1
    {
        s[k] := a[k]     // 'left'  elements
        t[k] := a[k+nh]  // 'right' elements
    }

    for k:=0 to nh-1
    {
        {s[k], t[k]} := {(s[k]+t[k]), (s[k]-t[k])}
    }

    fourier_shift(t[],nh,is*0.5)

    rec_fft_dif2(s[],nh,b[],is)
    rec_fft_dif2(t[],nh,c[],is)

    j := 0
    for k:=0 to nh-1
    {
        x[j]   := b[k]
        x[j+1] := c[k]
        j := j+2
    }
}
```

The parameter `is` $= \sigma = \pm 1$ is the sign of the transform. The data length `n` must be a power of 2. The result is returned in the array `x[]`. Again, the routine does no normalization.

Pseudo code for a non-recursive procedure (C++ implementation given in [FXT: file fft/fftdif2.cc]):

```
procedure fft_dif2(a[],ldn,is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn
```

```
    for ldm:=ldn to 1 step -1
    {
        m  := 2**ldm
        mh := m/2
        for j:=0 to mh-1
        {
            e := exp(is*2*PI*I*j/m)
            for r:=0 to n-m step m
            {
                u := a[r+j]
                v := a[r+j+mh]
                a[r+j]    := (u + v)
                a[r+j+mh] := (u - v) * e
            }
        }
    }
    revbin_permute(a[], n)
}
```

In DIF FFTs the `revbin_permute()`-procedure is called after the main loop, in the DIT code it was called before the main loop. As in the procedure for the DIT FFT (page 258) the inner loops where swapped to save trigonometric computations.

Extracting the `ldm==1` stage of the outermost loop is again a good idea. Replace the line

```
    for  ldm:=ldn to 1 step -1
```

by

```
    for  ldm:=ldn to 2 step -1
```

and insert

```
    for r:=0 to n-1 step 2
    {
        {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
    }
```

before the call of `revbin_permute(a[], n)`.

8.4 Saving trigonometric computations

The trigonometric computations are an expensive part of any FFT. There are two apparent ways for saving the involved CPU cycles, the use of lookup-tables and recursive methods.

8.4.1 Using lookup tables

The idea is to save all necessary sin/cos-values in an array and later looking up the values needed. This is a good idea if one wants to compute many FFTs of the same (small) length. For FFTs of large sequences one gets large lookup tables that can introduce a high cache-miss rate. Thereby one is likely experiencing little or no speed gain, even a notable slowdown is possible. However, for a length-n FFT one does not need to store all the (n complex or $2\,n$ real) sine/cosine values $\exp(2\,\pi\,i\,k/n) = \cos(2\,\pi\,k/n) + i\,\sin(2\,\pi\,k/n)$ where $k = 0, 1, 2, 3, \ldots, n-1$. For the lookups one can use the symmetry relations

$$\cos(\pi + x) = -\cos(x) \tag{8.25a}$$
$$\sin(\pi + x) = -\sin(x) \tag{8.25b}$$

To reduce the interval from $0 \ldots 2\pi$ to $0 \ldots \pi$. Exploiting the relations

$$\cos(\pi/2 + x) = -\sin(x) \tag{8.26a}$$
$$\sin(\pi/2 + x) = +\cos(x) \tag{8.26b}$$

further reduces the interval to $0 \ldots \pi/2$. Finally, the relation

$$\sin(x) \quad = \quad \cos(\pi/2 - x) \tag{8.27}$$

shows that only the table of cosines is needed. That is, already a table of the $n/4$ real values $\cos(2\,\pi\,i\,k/n)$ for $k = 0, 1, 2, 3, \ldots, n/4 - 1$ suffices for a length-n FFT computation. The size of the table is thereby cut by a factor of 8.

8.4.2 Recursive generation

In FFT computations one typically needs the values

$$[\,\exp(i\,\varphi\,0) = 1, \quad \exp(i\,\varphi\,\gamma), \quad \exp(i\,\varphi\,2\,\gamma), \quad \exp(i\,\varphi\,3\,\gamma), \quad \ldots\,] \quad \text{where} \quad \varphi \in \mathbb{R}$$

in sequence. The naive idea for a recursive computation of these values is to precompute $d = \exp(i\,\varphi\,\gamma)$ and then compute the following value using the identity $\exp(i\,\varphi\,k\,\gamma) = d \cdot \exp(i\,\varphi\,(k-1)\,\gamma)$. This method, however, is of no practical value because the numerical error grows exponentially in the process.

A stable version of a trigonometric recursion for the computation of the sequence can be stated as follows. Precompute

$$\begin{aligned}
c &= \cos\varphi, && (8.28a)\\
s &= \sin\varphi, && (8.28b)\\
\alpha &= 1 - \cos\gamma \qquad \text{[Cancellation!]} && (8.28c)\\
&= 2\left(\sin\frac{\gamma}{2}\right)^2 \qquad \text{[OK.]} && (8.28d)\\
\beta &= \sin\gamma && (8.28e)
\end{aligned}$$

Then compute the next value from the previous one via

$$\begin{aligned}
c_{next} &= c - (\alpha\,c + \beta\,s); && (8.29a)\\
s_{next} &= s - (\alpha\,s - \beta\,c); && (8.29b)
\end{aligned}$$

The underlying idea is to use the relation $E(\varphi + \gamma) = E(\varphi) - E(\varphi) \cdot z$ where $E(x) := \exp(2\,\pi\,i\,x)$. This leads to $z = 1 - \cos\gamma - i\,\sin\gamma = 2\,\left(\sin\frac{\gamma}{2}\right)^2 - i\,\sin\gamma$.

Do not expect to get all the precision you would get with calls of the sine and cosine functions, but even for very long FFTs less than 3 bits of precision are lost. When working with (C-type) `double`s it might be a good idea to use the type `long double` with the trigonometric recursion: the generated values will then always be accurate within the `double`-precision, provided `long double`s are actually more precise than `double`s. With high precision multiplication routines (that is, with exact integer convolution) this can be mandatory.

A real-world example from [FXT: `fht_dif_core()` in fht/fhtdif.cc]:

```
[--snip--]
double tt = M_PI_4/kh;   // the angle increment
double s1 = 0.0,  c1 = 1.0;  // start at angle zero
double al = sin(0.5*tt);
al *= (2.0*al);
double be = sin(tt);

for (ulong i=1; i<kh; i++)
{
    double t1 = c1;
    c1 -= (al*t1+be*s1);
    s1 -= (al*s1-be*t1);

    // here c1 = cos(tt*i) and s1 = sin(tt*i)
[--snip--]
```

The variable `tt` equals γ in relations 8.28d and 8.28e.

We finally mention the CORDIC algorithms for sine and cosine shown in section 18.2.1 on page 456.

8.5 Higher radix FFT algorithms

Higher radix FFT algorithms save trigonometric computations. The radix-4 FFT algorithms presented in what follows replace all multiplications with complex factors $(0, \pm i)$ by the obvious simpler operations. Radix-8 algorithms also simplify the special cases where the sines and cosines equal $\pm\sqrt{1/2}$.

Further the bookkeeping overhead is reduced due to the more unrolled structure. Moreover, the number of loads and stores is reduced.

More notation

Let a be a length-n sequence where n is a multiple of m.

- Let $a^{(r\%m)}$ denote the subsequence of the elements with index x where $x \equiv r \bmod m$. For example, $a^{(0\%2)} = a^{(even)}$ and $a^{(3\%4)} = [a_3, a_7, a_{11}, a_{15}, \dots]$. The length of $a^{(r\%m)}$ is n/m.
- Let $a^{(r/m)}$ denote the subsequence of elements with indices $\frac{r\,n}{m} \dots \frac{(r+1)\,n}{m} - 1$. For example $a^{(1/2)} = a^{(right)}$ and $a^{(2/3)}$ is the last third of a. The length of $a^{(r/m)}$ is also n/m.

8.5.1 Decimation in time

First reformulate the radix-2 DIT step (relations 8.21a and 8.21b on page 257) in the new notation:

$$\mathcal{F}\big[\,a\,\big]^{(0/2)} \overset{n/2}{=} \mathcal{S}^{0/2}\mathcal{F}\big[\,a^{(0\%2)}\,\big] + \mathcal{S}^{1/2}\mathcal{F}\big[\,a^{(1\%2)}\,\big] \tag{8.30a}$$

$$\mathcal{F}\big[\,a\,\big]^{(1/2)} \overset{n/2}{=} \mathcal{S}^{0/2}\mathcal{F}\big[\,a^{(0\%2)}\,\big] - \mathcal{S}^{1/2}\mathcal{F}\big[\,a^{(1\%2)}\,\big] \tag{8.30b}$$

Note that $\mathcal{S}^{0/2} = \mathcal{S}^0$ is the identity operator.

The derivation of the radix-4 step is analogous to the radix-2 step, it just involves more writing and does not give additional insights. So we just state the *radix-4 DIT FFT step* which can be applied when n is divisible by 4:

$$\mathcal{F}\big[\,a\,\big]^{(0/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\big[\,a^{(0\%4)}\,\big] + \mathcal{S}^{1/4}\mathcal{F}\big[\,a^{(1\%4)}\,\big] + \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big] + \mathcal{S}^{3/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big] \tag{8.31a}$$

$$\mathcal{F}\big[\,a\,\big]^{(1/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\big[\,a^{(0\%4)}\,\big] + i\sigma\mathcal{S}^{1/4}\mathcal{F}\big[\,a^{(1\%4)}\,\big] - \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big] - i\sigma\mathcal{S}^{3/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big] \tag{8.31b}$$

$$\mathcal{F}\big[\,a\,\big]^{(2/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\big[\,a^{(0\%4)}\,\big] - \mathcal{S}^{1/4}\mathcal{F}\big[\,a^{(1\%4)}\,\big] + \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big] - \mathcal{S}^{3/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big] \tag{8.31c}$$

$$\mathcal{F}\big[\,a\,\big]^{(3/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\big[\,a^{(0\%4)}\,\big] - i\sigma\mathcal{S}^{1/4}\mathcal{F}\big[\,a^{(1\%4)}\,\big] - \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big] + i\sigma\mathcal{S}^{3/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big] \tag{8.31d}$$

In contrast to the radix-2 step that happens to be identical for forward and backward transform the sign of the transform $\sigma = \pm 1$ appears explicitly. The relations, written more compactly:

$$\begin{aligned}\mathcal{F}\big[\,a\,\big]^{(j/4)} \overset{n/4}{=}\ & +e^{\sigma\,2\,i\,\pi\,0\,j/4}\cdot\mathcal{S}^{0/4}\mathcal{F}\big[\,a^{(0\%4)}\,\big] + e^{\sigma\,2\,i\,\pi\,1\,j/4}\cdot\mathcal{S}^{1/4}\mathcal{F}\big[\,a^{(1\%4)}\,\big] \\ & +e^{\sigma\,2\,i\,\pi\,2\,j/4}\cdot\mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big] + e^{\sigma\,2\,i\,\pi\,3\,j/4}\cdot\mathcal{S}^{3/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big]\end{aligned} \tag{8.32}$$

where $j = 0, 1, 2, 3$ and n is a multiple of 4.

Still more compactly:

$$\mathcal{F}\big[\,a\,\big]^{(j/4)} \overset{n/4}{=} \sum_{k=0}^{3} e^{\sigma 2\,i\,\pi\,k\,j/4}\cdot\mathcal{S}^{\sigma k/4}\mathcal{F}\big[\,a^{(k\%4)}\,\big] \qquad j = 0, 1, 2, 3 \tag{8.33}$$

where the summation symbol denotes *element-wise* summation of the sequences. The dot indicates multiplication of all elements of the sequence by the exponential.

The general *radix-r DIT FFT step*, applicable when n is a multiple of r, is:

$$\mathcal{F}\big[\,a\,\big]^{(j/r)} \quad \overset{n/r}{=} \quad \sum_{k=0}^{r-1} e^{\sigma\,2\,i\,\pi\,k\,j/r} \cdot \mathcal{S}^{\sigma\,k/r} \mathcal{F}\big[\,a^{(k\%r)}\,\big] \qquad j = 0, 1, 2, \ldots, r-1 \tag{8.34}$$

Our notation turned out to be useful indeed.

8.5.2 Decimation in frequency

The radix-2 DIF step (relations 8.24a and 8.24b on page 260) in the new notation:

$$\mathcal{F}\big[\,a\,\big]^{(0\%2)} \quad \overset{n/2}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{0/2}\Big(a^{(0/2)} + a^{(1/2)}\Big)\,\big] \tag{8.35a}$$

$$\mathcal{F}\big[\,a\,\big]^{(1\%2)} \quad \overset{n/2}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{1/2}\Big(a^{(0/2)} - a^{(1/2)}\Big)\,\big] \tag{8.35b}$$

The *radix-4 DIF FFT step*, applicable for n divisible by 4, is

$$\mathcal{F}\big[\,a\,\big]^{(0\%4)} \quad \overset{n/4}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{0/4}\Big(a^{(0/4)} + \quad a^{(1/4)} + a^{(2/4)} + \quad a^{(3/4)}\Big)\,\big] \tag{8.36a}$$

$$\mathcal{F}\big[\,a\,\big]^{(1\%4)} \quad \overset{n/4}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{1/4}\Big(a^{(0/4)} + i\,\sigma\,a^{(1/4)} - a^{(2/4)} - i\,\sigma\,a^{(3/4)}\Big)\,\big] \tag{8.36b}$$

$$\mathcal{F}\big[\,a\,\big]^{(2\%4)} \quad \overset{n/4}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{2/4}\Big(a^{(0/4)} - \quad a^{(1/4)} + a^{(2/4)} - \quad a^{(3/4)}\Big)\,\big] \tag{8.36c}$$

$$\mathcal{F}\big[\,a\,\big]^{(3\%4)} \quad \overset{n/4}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{3/4}\Big(a^{(0/4)} - i\,\sigma\,a^{(1/4)} - a^{(2/4)} + i\,\sigma\,a^{(3/4)}\Big)\,\big] \tag{8.36d}$$

Again, $\sigma = \pm 1$ is the sign of the transform. More compactly:

$$\mathcal{F}\big[\,a\,\big]^{(j\%4)} \quad \overset{n/4}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{\sigma\,j/4} \sum_{k=0}^{3} e^{\sigma\,2\,i\,\pi\,k\,j/4} \cdot a^{(k/4)}\,\big] \qquad j = 0, 1, 2, 3 \tag{8.37}$$

The general *radix-r DIF FFT step* is

$$\mathcal{F}\big[\,a\,\big]^{(j\%r)} \quad \overset{n/r}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{\sigma\,j/r} \sum_{k=0}^{r-1} e^{\sigma\,2\,i\,\pi\,k\,j/r} \cdot a^{(k/r)}\,\big] \qquad j = 0, 1, 2, \ldots, r-1 \tag{8.38}$$

8.5.3 Implementation of radix-r FFTs

For the implementation of a radix-r FFT with $r \neq 2$ the `revbin_permute` routine has to be replaced by its radix-r version `radix_permute`. The reordering now swaps elements a_x with $a_{\tilde{x}}$ where $\tilde{x}$ is obtained from x by reversing its radix-r expansion (see section 2.2 on page 83). In most practical cases one considers $r = p^x$ where p is prime. Pseudo code for a radix $r = p^x$ DIT FFT:

```
procedure fftdit_r(a[], n, is)
// complex a[0..n-1] input, result
// p (hardcoded)
// r == power of p (hardcoded)
// n == power of p (not necessarily a power of r)
{
    radix_permute(a[], n, p)

    lx := log(r) / log(p)  // r == p ** lx
    ln := log(n) / log(p)
    ldm := (log(n)/log(p)) % lx

    if ( ldm != 0 )  // n is not a power of p
    {
        xx := p**lx
        for z:=0 to n-xx step xx
```

```
        {
            fft_dit_xx(a[z..z+xx-1], is)  // inlined length-xx dit fft
        }
    }
    for ldm:=ldm+lx to ln step lx
    {
        m  := p**ldm
        mr := m/r
        for j := 0 to mr-1
        {
            e := exp(is*2*PI*I*j/m)
            for k:=0 to n-m step m
            {
                // all code in this block should be
                // inlined, unrolled and combined:
                // temporary  u[0..r-1]
                for z:=0 to r-1
                {
                    u[z] := a[k+j+mr*z]
                }
                radix_permute(u[], r, p)
                for z:=1 to r-1  // e**0 == 1
                {
                    u[z] := u[z] * e**z
                }
                r_point_fft(u[], is)
                for z:=0 to r-1
                {
                    a[k+j+mr*z] := u[z]
                }
            }
        }
    }
}
```

Of course the loops that use the variable `z` have to be unrolled, the (length-p^x) scratch space `u[]` has to be replaced by explicit variables (for example, `u0, u1, ...`) and the `r_point_fft(u[],is)` shall be an inlined p^x-point FFT.

There is one pitfall: If one uses the radix-p permutation instead of a radix-p^x permutation (for example, the radix-2 `revbin_permute()` for a radix-4 FFT), then some additional reordering is necessary in the innermost loop. In the given pseudo code this is indicated by the `radix_permute(u[],p)` just before the `p_point_fft(u[],is)` line.

8.5.4 Radix-4 DIT FFT codes

C++ code for a radix-4 DIT FFT is given in [FXT: file fft/fftdit4l.cc]:

```
static const ulong RX = 4;  // == r
static const ulong LX = 2;  // == log(r)/log(p) == log_2(r)

void
fft_dit4l(Complex *f, ulong ldn, int is)
// Decimation in time radix-4 FFT.
{
    double s2pi = ( is>0 ? 2.0*M_PI : -2.0*M_PI );
    const ulong n = (1UL<<ldn);
    revbin_permute(f, n);
    ulong ldm = (ldn&1);
    if ( ldm!=0 )  // n is not a power of 4, need a radix-2 step
    {
        for (ulong r=0; r<n; r+=2)
        {
            Complex a0 = f[r];
            Complex a1 = f[r+1];
```

```
            f[r]   = a0 + a1;
            f[r+1] = a0 - a1;
        }
    }
    ldm += LX;
    for ( ; ldm<=ldn ; ldm+=LX)
    {
        ulong m = (1UL<<ldm);
        ulong m4 = (m>>LX);
        double ph0 = s2pi/m;
        for (ulong j=0; j<m4; j++)
        {
            double phi = j*ph0;
            Complex e  = SinCos(phi);
            Complex e2 = SinCos(2.0*phi);
            Complex e3 = SinCos(3.0*phi);
            for (ulong r=0; r<n; r+=m)
            {
                ulong i0 = j + r;
                ulong i1 = i0 + m4;
                ulong i2 = i1 + m4;
                ulong i3 = i2 + m4;
                Complex a0 = f[i0];
                Complex a1 = f[i2]; // (!)
                Complex a2 = f[i1]; // (!)
                Complex a3 = f[i3];
                a1 *= e;
                a2 *= e2;
                a3 *= e3;
                Complex t0 = (a0+a2) + (a1+a3);
                Complex t2 = (a0+a2) - (a1+a3);
                Complex t1 = (a0-a2) + Complex(0,is) * (a1-a3);
                Complex t3 = (a0-a2) - Complex(0,is) * (a1-a3);
                f[i0] = t0;
                f[i1] = t1;
                f[i2] = t2;
                f[i3] = t3;
            }
        }
    }
}
```

For reasonable performance the call to the procedure `radix_permute(u[],p)` of the pseudo code has been replaced by changing indices in the loops where the `a[z]` are read. The respective lines are marked with the comment '`// (!)`'.

In order not to restrict the possible array sizes to powers of $p^x = 4$ but only to powers of $p = 2$ an additional radix-2 step has been prepended that is used when n is an odd power of two.

The routine [FXT: `fft_dit4_core_p1()` in fft/fftdit4.cc] is a reasonably optimized radix-4 DIT FFT implementation. It starts with an radix-2 or radix-8 step for the initial pass with trivial exp()-values. The core routine is hardcoded for $\sigma = +1$ and called with swapped real and imaginary part for the the inverse transform as explained in section 8.8 on page 274.

Using separate arrays for real and imaginary parts is suboptimal for large transforms due to cache problem. A routine that uses the C++ type `complex` is given in [FXT: file fft/cfftdit4.cc]. The core routine is hardcoded for $\sigma = -1$:

```
void
fft_dit4_core_m1(Complex *f, ulong ldn)
// Auxiliary routine for fft_dit4()
// Radix-4 decimation in frequency FFT
// ldn := base-2 logarithm of the array length
// Fixed isign = -1
// Input data must be in revbin_permuted order
{
    const ulong n = (1UL<<ldn);
```

```
    if ( n<=2 )
    {
        if ( n==2 )  sumdiff(f[0], f[1]);
        return;
    }
    ulong ldm = ldn & 1;
    if ( ldm!=0 )  // n is not a power of 4, need a radix-8 step
    {
        for (ulong i0=0; i0<n; i0+=8)  fft8_dit_core_m1(f+i0);  // isign
    }
    else
    {
        for (ulong i0=0; i0<n; i0+=4)
        {
            ulong i1 = i0 + 1;
            ulong i2 = i1 + 1;
            ulong i3 = i2 + 1;

            Complex x, y, u, v;
            sumdiff(f[i0], f[i1], x, u);
            sumdiff(f[i2], f[i3], y, v);
            v *= Complex(0, -1);  // isign
            sumdiff(u, v, f[i1], f[i3]);
            sumdiff(x, y, f[i0], f[i2]);
        }
    }
    ldm += 2 * LX;

    for ( ; ldm<=ldn; ldm+=LX)
    {
        ulong m = (1UL<<ldm);
        ulong m4 = (m>>LX);
        const double ph0 = -2.0*M_PI/m;  // isign

        for (ulong j=0; j<m4; j++)
        {
            double phi = j * ph0;
            Complex e  = SinCos(phi);
            Complex e2 = e * e;
            Complex e3 = e2 * e;

            for (ulong r=0; r<n; r+=m)
            {
                ulong i0 = j + r;
                ulong i1 = i0 + m4;
                ulong i2 = i1 + m4;
                ulong i3 = i2 + m4;

                Complex x = f[i1] * e2;
                Complex u;
                sumdiff3_r(x, f[i0], u);

                Complex v = f[i3] * e3;
                Complex y = f[i2] * e;
                sumdiff(y, v);
                v *= Complex(0, -1);  // isign

                sumdiff(u, v, f[i1], f[i3]);
                sumdiff(x, y, f[i0], f[i2]);
            }
        }
    }
}
```

The routine `fft8_dit_core_m1` called therein is an unrolled size-8 DIT FFT given in [FXT: file fft/fft8ditcore.cc]. We further need a version of the routine for the positive sign. The following changes need to be made:

```
void
fft_dit4_core_p1(Complex *f, ulong ldn)
// Fixed isign = +1
{
    [-snip-]
        for (ulong i0=0; i0<n; i0+=8)  fft8_dit_core_p1(f+i0);  // isign
    [-snip-]
            v *= Complex(0, +1);  // isign
    [-snip-]
```

```
        const double ph0 = +2.0*M_PI/m;  // isign
    [-snip-]
                v *= Complex(0, +1);  // isign
    [-snip-]
}
```

The routine to be called by the user is

```
void
fft_dit4(Complex *f, ulong ldn, int is)
// Fast Fourier Transform
// ldn := base-2 logarithm of the array length
// is := sign of the transform (+1 or -1)
// Radix-4 decimation in time algorithm
{
    revbin_permute(f, 1UL<<ld);
    if ( is>0 )  fft_dit4_core_p1(f, ldn);
    else         fft_dit4_core_m1(f, ldn);
}
```

8.5.5 Radix-4 DIF FFT codes

Pseudo code for a radix-4 DIF FFT:

```
procedure fftdif4(a[],ldn,is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn
    for ldm := ldn to 2 step -2
    {
        m  := 2**ldm
        mr := m/4
        for j := 0 to mr-1
        {
            e  := exp(is*2*PI*I*j/m)
            e2 := e * e
            e3 := e2 * e
            for r := 0 to n-m step m
            {
                u0 := a[r+j]
                u1 := a[r+j+mr]
                u2 := a[r+j+mr*2]
                u3 := a[r+j+mr*3]
                x := u0 + u2
                y := u1 + u3
                t0 := x + y  // == (u0+u2) + (u1+u3)
                t2 := x - y  // == (u0+u2) - (u1+u3)
                x := u0 - u2
                y := (u1 - u3)*I*is
                t1 := x + y  // == (u0-u2) + (u1-u3)*I*is
                t3 := x - y  // == (u0-u2) - (u1-u3)*I*is
                t1 := t1 * e
                t2 := t2 * e2
                t3 := t3 * e3
                a[r+j]      := t0
                a[r+j+mr]   := t2  // (!)
                a[r+j+mr*2] := t1  // (!)
                a[r+j+mr*3] := t3
            }
        }
    }
    if is_odd(ldn) then  // n not a power of 4
    {
        for r:=0 to n-2 step 2
        {
            {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
        }
    }
    revbin_permute(a[],n)
}
```

The C++ equivalent is [FXT: `fft_dif4l()` in fft/fftdif4l.cc]. A reasonably optimized implementation is given in [FXT: file fft/fftdif4.cc], it is hardcoded for $\sigma = +1$:

```
void
fft_dif4_core_p1(double *fr, double *fi, ulong ldn)
// Auxiliary routine for fft_dif4().
// Decimation in frequency (DIF) radix-4 FFT.
// Output data is in revbin_permuted order.
// ldn := base-2 logarithm of the array length.
// Fixed isign = +1
{
    const ulong n = (1UL<<ldn);

    if ( n<=2 )
    {
        if ( n==2 )
        {
            sumdiff(fr[0], fr[1]);
            sumdiff(fi[0], fi[1]);
        }
        return;
    }

    for (ulong ldm=ldn; ldm>=(LX<<1); ldm-=LX)
    {
        ulong m = (1UL<<ldm);
        ulong m4 = (m>>LX);

        const double ph0 = 2.0*M_PI/m;

        for (ulong j=0; j<m4; j++)
        {
            double c,s, c2,s2, c3,s3;
            SinCos(j*ph0, &s, &c);
            csqr(c, s, c2, s2);
            cmult(c, s, c2, s2, c3, s3);

            for (ulong r=0; r<n; r+=m)
            {
                ulong i0 = j + r;
                ulong i1 = i0 + m4;
                ulong i2 = i1 + m4;
                ulong i3 = i2 + m4;

                double xr,yr,ur,vr, xi,yi,ui,vi;

                // {x,u} = {f[i0]+f[i2], f[i0]-f[i2]}:
                sumdiff(fr[i0], fr[i2], xr, ur);
                sumdiff(fi[i0], fi[i2], xi, ui);

                // {y,v} = {f[i1]+f[i3], (f[i1]-f[i3])*(0,is)}:
                sumdiff(fi[i3], fi[i1], yi, vr);
                sumdiff(fr[i1], fr[i3], yr, vi);

                diffsum3(xr, yr, fr[i0]);
                diffsum3(xi, yi, fi[i0]);

                cmult(c2, s2, yr, yi, fr[i1], fi[i1]);

                sumdiff(ur, vr, xr, yr);
                sumdiff(ui, vi, xi, yi);

                cmult(c3, s3, yr, yi, fr[i3], fi[i3]);
                cmult(c,  s,  xr, xi, fr[i2], fi[i2]);
            }
        }
    }

    if ( (ldn&1)!=0 )  // n is not a power of 4, need a radix-8 step
    {
        for (ulong i0=0; i0<n; i0+=8)  fft8_dif_core_p1(fr+i0, fi+i0);
    }
    else
    {
        for (ulong i0=0; i0<n; i0+=4)
        {
            double xr,yr,ur,vr, xi,yi,ui,vi;
            ulong i1 = i0 + 1;
            ulong i2 = i1 + 1;
            ulong i3 = i2 + 1;
```

```
            sumdiff(fr[i0], fr[i2], xr, ur);
            sumdiff(fr[i1], fr[i3], yr, vi);
            sumdiff(fi[i0], fi[i2], xi, ui);
            sumdiff(fi[i3], fi[i1], yi, vr);

            sumdiff(xi, yi, fi[i0], fi[i1]);
            sumdiff(ui, vi, fi[i2], fi[i3]);
            sumdiff(xr, yr, fr[i0], fr[i1]);
            sumdiff(ur, vr, fr[i2], fr[i3]);
        }
    }
}
```

The `sumdiff()` function is defined in [FXT: file aux0/sumdiff.h].

The routine to be called by the user is

```
void
fft_dif4(double *fr, double *fi, ulong ldn, int is)
// Fast Fourier Transform
// ldn := base-2 logarithm of the array length
// is := sign of the transform (+1 or -1)
// Radix-4 decimation in frequency algorithm
{
    if ( is>0 )  fft_dif4_core_p1(fr, fi, ldn);
    else         fft_dif4_core_p1(fi, fr, ldn);

    revbin_permute(fr, 1UL<<ldn);
    revbin_permute(fi, 1UL<<ldn);
}
```

A version that uses the C++ type `complex` is given in [FXT: file fft/cfftdif4.cc]. The routine to be called by the user is

```
void
fft_dif4(Complex *f, ulong ldn, int is)
// Fast Fourier Transform
// ldn := base-2 logarithm of the array length
// is := sign of the transform (+1 or -1)
// radix-4 decimation in frequency algorithm
{
    if ( is>0 )  fft_dif4_core_p1(f, ldn);
    else         fft_dif4_core_m1(f, ldn);
    revbin_permute(f, 1UL<<ldn);
}
```

8.6 Split-radix Fourier transforms (SRFT)

The idea underlying the *split-radix FFT* is to use both radix-2 and radix-4 decompositions at the same time.

From the radix-2 (DIF) decomposition (relations 8.24a and 8.24b on page 260) we use the first, the one for the even indices. For the odd indices we use the radix-4 splitting (relations 8.36b and 8.36d on page 264, slightly reordered).

The radix-4 decimation in frequency (DIF) step for the split-radix FFT:

$$\mathcal{F}\big[\,a\,\big]^{(0\%2)} \stackrel{n/2}{=} \mathcal{F}\big[\,\Big(a^{(0/2)} + a^{(1/2)}\Big)\,\big] \tag{8.39a}$$

$$\mathcal{F}\big[\,a\,\big]^{(1\%4)} \stackrel{n/4}{=} \mathcal{F}\big[\,\mathcal{S}^{1/4}\Big(\Big(a^{(0/4)} - a^{(2/4)}\Big) + i\,\sigma\,\Big(a^{(1/4)} - a^{(3/4)}\Big)\Big)\,\big] \tag{8.39b}$$

$$\mathcal{F}\big[\,a\,\big]^{(3\%4)} \stackrel{n/4}{=} \mathcal{F}\big[\,\mathcal{S}^{3/4}\Big(\Big(a^{(0/4)} - a^{(2/4)}\Big) - i\,\sigma\,\Big(a^{(1/4)} - a^{(3/4)}\Big)\Big)\,\big] \tag{8.39c}$$

Now we have expressed the length-$N = 2^n$ FFT as one length-$N/2$ and two length-$N/4$ FFTs. A nice feature is that the operation count of the split-radix FFT is actually lower than that of the radix-4 FFT. Using the introduced notation it is almost trivial to write down the DIT version of the algorithm:

The radix-4 decimation in time (DIT) step for the split-radix FFT:

$$\mathcal{F}\big[\,a\,\big]^{(0/2)} \stackrel{n/2}{=} \Big(\mathcal{F}\big[\,a^{(0\%2)}\,\big] + \mathcal{S}^{1/2}\mathcal{F}\big[\,a^{(1\%2)}\,\big]\Big) \tag{8.40a}$$

$$\mathcal{F}\big[\,a\,\big]^{(1/4)} \stackrel{n/4}{=} \Big(\mathcal{F}\big[\,a^{(0\%4)}\,\big] - \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big]\Big) + i\sigma\mathcal{S}^{1/4}\Big(\mathcal{F}\big[\,a^{(1\%4)}\,\big] - \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big]\Big) \tag{8.40b}$$

$$\mathcal{F}\big[\,a\,\big]^{(3/4)} \stackrel{n/4}{=} \Big(\mathcal{F}\big[\,a^{(0\%4)}\,\big] - \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big]\Big) - i\sigma\mathcal{S}^{1/4}\Big(\mathcal{F}\big[\,a^{(1\%4)}\,\big] - \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big]\Big) \tag{8.40c}$$

Pseudo code for the split-radix DIF algorithm:

```
procedure fft_splitradix_dif(x[],y[],ldn,is)
{
    n := 2**ldn
    if n<=1  return
    n2 := 2*n
    for k:=1 to ldn
    {
        n2 := n2 / 2
        n4 := n2 / 4
        e := 2 * PI / n2
        for j:=0 to n4-1
        {
            a := j * e
            cc1 := cos(a)
            ss1 := sin(a)
            cc3 := cos(3*a)  // == 4*cc1*(cc1*cc1-0.75)
            ss3 := sin(3*a)  // == 4*ss1*(0.75-ss1*ss1)
            ix := j
            id := 2*n2
            while ix<n-1
            {
                i0 := ix
                while i0 < n
                {
                    i1 := i0 + n4
                    i2 := i1 + n4
                    i3 := i2 + n4
                    {x[i0], r1} := {x[i0] + x[i2], x[i0] - x[i2]}
                    {x[i1], r2} := {x[i1] + x[i3], x[i1] - x[i3]}
                    {y[i0], s1} := {y[i0] + y[i2], y[i0] - y[i2]}
                    {y[i1], s2} := {y[i1] + y[i3], y[i1] - y[i3]}
                    {r1, s3} := {r1+s2, r1-s2}
                    {r2, s2} := {r2+s1, r2-s1}
                    // complex mult: (x[i2],y[i2]) := -(s2,r1) * (ss1,cc1)
                    x[i2] :=  r1*cc1 - s2*ss1
                    y[i2] := -s2*cc1 - r1*ss1
                    // complex mult: (y[i3],x[i3]) := (r2,s3) * (cc3,ss3)
                    x[i3] :=  s3*cc3 + r2*ss3
                    y[i3] :=  r2*cc3 - s3*ss3
                    i0 := i0 + id
                }
                ix := 2 * id - n2 + j
                id := 4 * id
            }
        }
    }
    ix := 1
    id := 4
    while ix<n
    {
        for i0:=ix-1 to n-id step id
        {
            i1 := i0 + 1
            {x[i0], x[i1]} := {x[i0]+x[i1], x[i0]-x[i1]}
            {y[i0], y[i1]} := {y[i0]+y[i1], y[i0]-y[i1]}
        }
        ix := 2 * id - 1
```

```
        id := 4 * id
    }
    revbin_permute(x[],n)
    revbin_permute(y[],n)
    if is>0
    {
        for j:=1 to n/2-1
        {
            swap(x[j], x[n-j])
        }
        for j:=1 to n/2-1
        {
            swap(y[j], y[n-j])
        }
    }
}
```

The C++ implementation given in [FXT: file fft/fftsplitradix.cc] uses a DIF core as above which was given in [41]. The C++ type `complex` version of the SRFT given in [FXT: file fft/cfftsplitradix.cc] uses an DIF or DIT core, depending on the sign of the transform. Here we just give the DIF version:

```
void
split_radix_dif_fft_core(Complex *f, ulong ldn)
// Split-radix decimation in frequency (DIF) FFT.
// ldn := base-2 logarithm of the array length.
// Fixed isign = +1
// Output data is in revbin_permuted order.
{
    if ( ldn==0 )  return;
    const ulong n = (1UL<<ldn);
    double s2pi = 2.0*M_PI;  // pi*2*isign
    ulong n2 = 2*n;
    for (ulong k=1; k<ldn; k++)
    {
        n2 >>= 1;  // == n>>(k-1) == n, n/2, n/4, ..., 4
        const ulong n4 = n2 >> 2;  // == n/4, n/8, ..., 1
        const double e = s2pi / n2;
        {  // j==0:
            const ulong j = 0;
            ulong ix = j;
            ulong id = (n2<<1);
            while ( ix<n )
            {
                for (ulong i0=ix; i0<n; i0+=id)
                {
                    ulong i1 = i0 + n4;
                    ulong i2 = i1 + n4;
                    ulong i3 = i2 + n4;
                    Complex t0, t1;
                    sumdiff3(f[i0], f[i2], t0);
                    sumdiff3(f[i1], f[i3], t1);
                    // t1 *= Complex(0, 1);  // +isign
                    t1 = Complex(-t1.imag(), t1.real());
                    sumdiff(t0, t1);
                    f[i2] = t0; // * Complex(cc1, ss1);
                    f[i3] = t1; // * Complex(cc3, ss3);
                }
                ix = (id<<1) - n2 + j;
                id <<= 2;
            }
        }
        for (ulong j=1; j<n4; j++)
        {
            double a = j * e;
            double cc1,ss1, cc3,ss3;
            SinCos(a, &ss1, &cc1);
            SinCos(3.0*a, &ss3, &cc3);
            ulong ix = j;
```

```
            ulong id = (n2<<1);
            while ( ix<n )
            {
                for (ulong i0=ix; i0<n; i0+=id)
                {
                    ulong i1 = i0 + n4;
                    ulong i2 = i1 + n4;
                    ulong i3 = i2 + n4;
                    Complex t0, t1;
                    sumdiff3(f[i0], f[i2], t0);
                    sumdiff3(f[i1], f[i3], t1);
                    t1 = Complex(-t1.imag(), t1.real());
                    sumdiff(t0, t1);
                    f[i2] = t0 * Complex(cc1, ss1);
                    f[i3] = t1 * Complex(cc3, ss3);
                }
                ix = (id<<1) - n2 + j;
                id <<= 2;
            }
        }
    }
    for (ulong ix=0, id=4;  ix<n;  id*=4)
    {
        for (ulong i0=ix; i0<n; i0+=id)  sumdiff(f[i0], f[i0+1]);
        ix = 2*(id-1);
    }
}
```

8.7 Symmetries of the Fourier transform

A bit of notation. Let $\overline{a}$ be the length-n sequence a reversed around the element with index 0:

$$\overline{a}_0 := a_0 \tag{8.41a}$$

$$\overline{a}_{n/2} := a_{n/2} \qquad \text{if } n \text{ even} \tag{8.41b}$$

$$\overline{a}_k := a_{n-k} = a_{-k} \tag{8.41c}$$

That is, we consider the indices modulo n and $\overline{a}$ is the sequence a with negated indices. Element zero stays in its place and for even n there is also an element with index $n/2$ that stays in place.
Example one, length-4: $a := [0, 1, 2, 3]$ then $\overline{a} = [0, 3, 2, 1]$ (zero and two stay).
Example two, length-5: $a := [0, 1, 2, 3, 4]$ then $\overline{a} = [0, 4, 3, 2, 1]$ (only zero stays).

Let a_S and a_A denote the symmetric and antisymmetric part of the sequence a, respectively:

$$a_S := \frac{1}{2}\,(a + \overline{a}) \tag{8.42a}$$

$$a_A := \frac{1}{2}\,(a - \overline{a}) \tag{8.42b}$$

The elements with indices 0 and (for even n) $n/2$ of a_A are zero. One has

$$a = a_S + a_A \tag{8.43a}$$

$$\overline{a} = a_S - a_A \tag{8.43b}$$

Let $c + i\,d$ the FT of the sequence $a + i\,b$. Then

$$\mathcal{F}\big[\,(a_S + a_A) + i\,(b_S + b_A)\,\big] = (c_S + c_A) + i\,(d_S + d_A) \quad \text{where} \tag{8.44a}$$

$$\mathcal{F}\big[\,a_S\,\big] = c_S \in \mathbb{R} \tag{8.44b}$$

$$\mathcal{F}\big[\,a_A\,\big] = i\,d_A \in i\,\mathbb{R} \tag{8.44c}$$

$$\mathcal{F}\big[\,i\,b_S\,\big] = i\,d_S \in i\,\mathbb{R} \tag{8.44d}$$

$$\mathcal{F}\big[\,i\,b_A\,\big] = c_A \in \mathbb{R} \tag{8.44e}$$

Where we write $a \in \mathbb{R}$ as a short form for a purely real sequence a. Equivalently, write $a \in i\,\mathbb{R}$ for purely imaginary sequences. Thereby the FT of a complex symmetric or antisymmetric sequence is symmetric or antisymmetric, respectively:

$$\mathcal{F}\big[\,a_S + i\,b_S\,\big] \;=\; c_S + i\,d_S \tag{8.45a}$$
$$\mathcal{F}\big[\,a_A + i\,b_A\,\big] \;=\; c_A + i\,d_A \tag{8.45b}$$

The real and imaginary part of the transform of a symmetric sequence correspond to the real and imaginary part of the original sequence. With an antisymmetric sequence the transform of the real and imaginary part correspond to the imaginary and real part of the original sequence.

$$\mathcal{F}\big[\,(a_S + a_A)\,\big] \;=\; c_S + i\,d_A \tag{8.46a}$$
$$\mathcal{F}\big[\,i\,(b_S + b_A)\,\big] \;=\; c_A + i\,d_S \tag{8.46b}$$

Now let the sequence a be purely real. Then

$$\mathcal{F}\big[\,a_S\,\big] \;=\; +\overline{\mathcal{F}\big[\,a_S\,\big]} \qquad \in\ \mathbb{R} \tag{8.47a}$$
$$\mathcal{F}\big[\,a_A\,\big] \;=\; -\overline{\mathcal{F}\big[\,a_A\,\big]} \qquad \in i\,\mathbb{R} \tag{8.47b}$$

That is, the FT of a real symmetric sequence is real and symmetric and the FT of a real antisymmetric sequence is purely imaginary and antisymmetric. Thereby the FT of a general real sequence is the complex conjugate of its reversed:

$$\mathcal{F}\big[\,a\,\big] \;=\; \overline{\mathcal{F}\big[\,a\,\big]}^{\,*} \qquad for \quad a \in \mathbb{R} \tag{8.48}$$

Similarly, for a purely imaginary sequence $b \in i\mathbb{R}$:

$$\mathcal{F}\big[\,b_S\,\big] \;=\; +\overline{\mathcal{F}\big[\,b_S\,\big]} \qquad \in i\,\mathbb{R} \tag{8.49a}$$
$$\mathcal{F}\big[\,b_A\,\big] \;=\; -\overline{\mathcal{F}\big[\,b_A\,\big]} \qquad \in\ \mathbb{R} \tag{8.49b}$$

We compare the results of the Fourier transform and its inverse (the transform with negated sign σ) by symbolically writing the transforms as a complex multiplication with the trigonometric term (using C for cosine, S for sine):

$$\mathcal{F}\big[\,a + i\,b\,\big] \;:\; (a + i\,b)\,(C + i\,S) \;=\; (a\,C - b\,S) + i\,(b\,C + a\,S) \tag{8.50a}$$
$$\mathcal{F}^{-1}\big[\,a + i\,b\,\big] \;:\; (a + i\,b)\,(C - i\,S) \;=\; (a\,C + b\,S) + i\,(b\,C - a\,S) \tag{8.50b}$$

The terms on the rhs. can be identified with those in relation 8.44a on the preceding page. We see that changing the sign of the transform leads to a result where the components due to the asymmetric components of the input are negated.

8.8 Inverse FFT for free

Suppose you programmed some FFT algorithm just for one value of `is`, the sign in the exponent. There is a nice trick that gives the inverse transform for free, if your implementation uses separate arrays for real and imaginary part of the complex sequences to be transformed. If your procedure is something like

```
procedure my_fft(ar[], ai[], ldn)  // only for is==+1 !
// real ar[0..2**ldn-1] input, result, real part
// real ai[0..2**ldn-1] input, result, imaginary part
{
    // incredibly complicated code
    // that you cannot see how to modify
    // for is==-1
}
```

Then you *don't* need to modify this procedure at all in order to get the inverse transform. If you want the inverse transform somewhere then just, instead of

```
my_fft(ar[], ai[], ldn)  // forward fft
```

type

```
my_fft(ai[], ar[], ldn)  // backward fft
```

Note the swapped real- and imaginary parts! The same trick works for a procedure coded for fixed `is`$= -1$.

To see, why this works, we first note that

$$\begin{aligned} \mathcal{F}\big[\,a + i\,b\,\big] &= \mathcal{F}\big[\,a_S\,\big] + i\,\sigma\,\mathcal{F}\big[\,a_A\,\big] + i\,\mathcal{F}\big[\,b_S\,\big] + \sigma\,\mathcal{F}\big[\,b_A\,\big] && (8.51a)\\ &= \mathcal{F}\big[\,a_S\,\big] + i\,\mathcal{F}\big[\,b_S\,\big] + i\,\sigma\,\big(\mathcal{F}\big[\,a_A\,\big] - i\,\mathcal{F}\big[\,b_A\,\big]\big) && (8.51b)\end{aligned}$$

and the computation with swapped real- and imaginary parts gives

$$\mathcal{F}\big[\,b + i\,a\,\big] = \mathcal{F}\big[\,b_S\,\big] + i\,\mathcal{F}\big[\,a_S\,\big] + i\,\sigma\,\big(\mathcal{F}\big[\,b_A\,\big] - i\,\mathcal{F}\big[\,a_A\,\big]\big) \qquad (8.52a)$$

... but the real and imaginary parts are implicitly swapped at the end of the computation, giving

$$\mathcal{F}\big[\,a_S\,\big] + i\,\mathcal{F}\big[\,b_S\,\big] - i\,\sigma\,\big(\mathcal{F}\big[\,a_A\,\big] - i\,\mathcal{F}\big[\,b_A\,\big]\big) = \mathcal{F}^{-1}\big[\,a + i\,b\,\big] \qquad (8.52b)$$

When the type `Complex` is used then the best way to achieve the inverse transform may be to reverse the sequence according to the symmetry of the FT according to relations 8.50a and 8.50b on the preceding page: The transform with negated sign can be computed by reversing the order of the result (use [FXT: `reverse_0()` in perm/reverse.h]). The reversal can also happen with the input data before the transform. This can be advantageous if the has to be copied anyway (use [FXT: `copy_reverse_0()` in aux1/copy.h]). While not really 'free' the additional work shouldn't matter in most cases.

8.9 Real valued Fourier transforms

The Fourier transform of a purely real sequence $c = \mathcal{F}\big[\,a\,\big]$ where $a \in \mathbb{R}$ has a symmetric real part ($\mathrm{Re}\,\overline{c} = \mathrm{Re}\,c$, relation 8.48) and an antisymmetric imaginary part ($\mathrm{Im}\,\overline{c} = -\,\mathrm{Im}\,c$). The symmetric and antisymmetric part of the original sequence correspond to the symmetric (and purely real) and antisymmetric (and purely imaginary) part of the transform, respectively:

$$\mathcal{F}\big[\,a\,\big] = \mathcal{F}\big[\,a_S\,\big] + i\,\sigma\,\mathcal{F}\big[\,a_A\,\big] \qquad (8.53)$$

Simply using a complex FFT for real input is basically a waste by a factor 2 of memory and CPU cycles. There are several ways out:

- wrapper routines for complex FFTs (section 8.9.1 on page 277)
- usage of the fast Hartley transform (section 12.5 on page 365)
- special versions of the split-radix algorithm (section 8.9.2 on page 278)

All techniques have in common that they store only half of the complex result to avoid the redundancy due to the symmetries of a complex FT of purely real input. The result of a real to (half-) complex FT (abbreviated R2CFT) contains the purely real components c_0 (the 'DC-part' of the input signal) and, in case n is even, $c_{n/2}$ (the Nyquist frequency part). The inverse procedure, the (half-) complex to real transform (abbreviated C2RFT) must be compatible to the ordering of the R2CFT.

Sign of the transforms

The sign of the transform can be chosen arbitrarily to be either $+1$ or -1. Note that transform with the 'other sign' is *not* the inverse transform. The R2CFT and its inverse C2RFT must use the same sign.

Some R2CFT and C2RFT implementations are hardcoded for a fixed sign. In order to obtain the R2CFT with the other sign the trick (in the spirit of section 8.8) is to negate the imaginary part after the transform. This of course is not much of a trick at all. In case one has to copy the data anyway before the transform one can exploit the relation

$$\mathcal{F}\big[\,\overline{a}\,\big] \;=\; \mathcal{F}\big[\,a_S\,\big] - i\,\sigma\,\mathcal{F}\big[\,a_A\,\big] \tag{8.54}$$

That is, copy the real data in reversed order to get the transform with the other sign. This technique does not involve an extra pass and should be virtually for free.

Obviously for the complex-to-real FTs (C2RFT) one has to negate the imaginary part before the transform in order to obtain the inverse transform for the other sign.

Data ordering

All procedures presented here use the following scheme for the real part of the transformed sequence c in the output array `a[]`:

$$\begin{aligned} \texttt{a[0]} &= \operatorname{Re} c_0 \\ \texttt{a[1]} &= \operatorname{Re} c_1 \\ \texttt{a[2]} &= \operatorname{Re} c_2 \\ &\vdots \\ \texttt{a[n/2]} &= \operatorname{Re} c_{n/2} \end{aligned} \tag{8.55}$$

For the imaginary part of the result there are two schemes:
Scheme 1 ('parallel ordering') is

$$\begin{aligned} \texttt{a[n/2+1]} &= \operatorname{Im} c_1 \\ \texttt{a[n/2+2]} &= \operatorname{Im} c_2 \\ \texttt{a[n/2+3]} &= \operatorname{Im} c_3 \\ &\vdots \\ \texttt{a[n-1]} &= \operatorname{Im} c_{n/2-1} \end{aligned} \tag{8.56}$$

Note the absence of the elements $\operatorname{Im} c_0$ and $\operatorname{Im} c_{n/2}$ which are always zero.

Scheme 2 ('antiparallel ordering') is

$$\begin{aligned} \texttt{a[n/2+1]} &= \operatorname{Im} c_{n/2-1} \\ \texttt{a[n/2+2]} &= \operatorname{Im} c_{n/2-2} \\ \texttt{a[n/2+3]} &= \operatorname{Im} c_{n/2-3} \\ &\vdots \\ \texttt{a[n-1]} &= \operatorname{Im} c_1 \end{aligned} \tag{8.57}$$

8.9.1 Real valued FT via wrapper routines

A simple way to use a complex length-$n/2$ FFT for a real length-n FFT (n even) is to use some post- and preprocessing routines. For a real sequence a one feeds the (half length) complex sequence $f = a^{(even)} + i\,a^{(odd)}$ into a complex FFT. Some post-processing is necessary. This is not the most elegant real FFT available, but it is directly usable to turn complex FFTs of any (even) length into a real-valued FFT.

A C++ implementation of the real to complex FFT (R2CFT) is given in [FXT: file realfft/realfftwrap.cc]:

```
void
wrap_real_complex_fft(double *f, ulong ldn, int is/*=+1*/)
// Real to complex FFT (R2CFT)
// Ordering of output:
// f[0]     = re[0]   (DC part, purely real)
// f[1]     = re[n/2] (nyquist freq, purely real)
// f[2]     = re[1]
// f[3]     = im[1]
// f[4]     = re[2]
// f[5]     = im[2]
//          ...
// f[2*i]   = re[i]
// f[2*i+1] = im[i]
//          ...
// f[n-2]   = re[n/2-1]
// f[n-1]   = im[n/2-1]
//
{
    if ( ldn==0 )  return;

    fht_fft((Complex *)f, ldn-1, +1);

    const ulong n = 1UL<<ldn;
    const ulong nh = n/2, n4 = n/4;
    const double phi0 = M_PI / nh;
    for(ulong i=1; i<n4; i++)
    {
        ulong i1 = 2 * i;   // re low [2, 4, ..., n/2-2]
        ulong i2 = i1 + 1;  // im low [3, 5, ..., n/2-1]

        ulong i3 = n - i1;  // re hi  [n-2, n-4, ..., n/2+2]
        ulong i4 = i3 + 1;  // im hi  [n-1, n-3, ..., n/2+3]

        double f1r, f2i;
        sumdiff05(f[i3], f[i1], f1r, f2i);

        double f2r, f1i;
        sumdiff05(f[i2], f[i4], f2r, f1i);

        double c, s;
        double phi = i*phi0;
        SinCos(phi, &s, &c);

        double tr, ti;
        cmult(c, s, f2r, f2i, tr, ti);

        // f[i1] = f1r + tr;  // re low
        // f[i3] = f1r - tr;  // re hi
        // =^=
        sumdiff(f1r, tr, f[i1], f[i3]);


        // f[i4] = is * (ti + f1i);  // im hi
        // f[i2] = is * (ti - f1i);  // im low
        // =^=
        if ( is>0 )  sumdiff( ti,  f1i, f[i4], f[i2]);
        else         sumdiff(-ti,  f1i, f[i2], f[i4]);
    }
    sumdiff(f[0], f[1]);

    if ( nh>=2 )  f[nh+1] *= is;
}
```

The inverse routine, the complex to real FFT (C2RFT), is [FXT: file realfft/realfftwrap.cc]:

```
void
wrap_complex_real_fft(double *f, ulong ldn, int is/*=+1*/)
// Complex to real FFT (C2RFT).
// Inverse of wrap_real_complex_fft().
```

```
// Ordering of input is same as for
//   the output of wrap_real_complex_fft()
{
    if ( ldn==0 )  return;
    const ulong n = 1UL<<ldn;
    const ulong nh = n/2, n4 = n/4;
    const double phi0 = -M_PI / nh;
    for(ulong i=1; i<n4; i++)
    {
        ulong i1 = 2 * i;   // re low [2, 4, ..., n/2-2]
        ulong i2 = i1 + 1;  // im low [3, 5, ..., n/2-1]
        ulong i3 = n - i1;  // re hi  [n-2, n-4, ..., n/2+2]
        ulong i4 = i3 + 1;  // im hi  [n-1, n-3, ..., n/2+3]
        double f1r, f2i;
        // double f1r =  f[i1] + f[i3];  // re symm
        // double f2i =  f[i1] - f[i3];  // re asymm
        // =^=
        sumdiff(f[i1], f[i3], f1r, f2i);
        double f2r, f1i;
        // double f2r = -f[i2] - f[i4];  // im symm
        // double f1i =  f[i2] - f[i4];  // im asymm
        // =^=
        sumdiff(-f[i4], f[i2], f1i, f2r);
        double c, s;
        double phi = i*phi0;
        SinCos(phi, &s, &c);
        double tr, ti;
        cmult(c, s, f2r, f2i, tr, ti);
        // f[i1] = f1r + tr;  // re low
        // f[i3] = f1r - tr;  // re hi
        // =^=
        sumdiff(f1r, tr, f[i1], f[i3]);
        // f[i2] = ti - f1i;  // im low
        // f[i4] = ti + f1i;  // im hi
        // =^=
        sumdiff(ti, f1i, f[i4], f[i2]);
    }
    sumdiff(f[0], f[1]);
    if ( nh>=2 )  { f[nh] *= 2.0; f[nh+1] *= 2.0; }
    fht_fft((Complex *)f, ldn-1, -1);
    if ( is<0 )  reverse_0(f, n);
}
```

8.9.2 Real valued split-radix Fourier transforms

We just give pseudo code for the split-radix real to complex FFT and its inverse. The C++ implementations are given in [FXT: file realfft/realfftsplitradix.cc]. The code given here follows [42].

Recall that the following pseudo code

```
{ a, b } := { c, d }
```

is a parallel assignment. For example,

```
{x0, x1} := {x0+x1, x0-x1 }
```

would translate to the C code

```
double s, d;
s = x0 + x1;  d = x0 - x1;
x0 = s;  x1 = d;
```

Use the function `sumdiff()` [FXT: file aux0/sumdiff.h] for the translation of the parallel assignments.

Real to complex split-radix FFT

Pseudo code for the split-radix R2CFT algorithm, the sign of the transform is hardcoded to $\sigma = -1$:

```
procedure r2cft_splitradix_dit(x[], ldn)
{
    n := 2**ldn

    ix := 1;
    id := 4;
    do
    {
        i0 := ix-1
        while i0<n
        {
            i1 := i0 + 1
            {x[i0], x[i1]} := {x[i0]+x[i1], x[i0]-x[i1]}
            i0 := i0 + id
        }
        ix := 2*id-1
        id := 4 * id
    }
    while ix<n

    n2 := 2
    nn := n/4
    while nn!=0
    {
        ix := 0
        n2 := 2*n2
        id := 2*n2
        n4 := n2/4
        n8 := n2/8
        do  // ix loop
        {
            i0 := ix
            while i0<n
            {
                i1 := i0
                i2 := i1 + n4
                i3 := i2 + n4
                i4 := i3 + n4

                {t1, x[i4]} := {x[i4]+x[i3], x[i4]-x[i3]}
                {x[i1], x[i3]} := {x[i1]+t1, x[i1]-t1}

                if  n4!=1
                {
                    i1 := i1 + n8
                    i2 := i2 + n8
                    i3 := i3 + n8
                    i4 := i4 + n8

                    t1 := (x[i3]+x[i4]) * sqrt(1/2)
                    t2 := (x[i3]-x[i4]) * sqrt(1/2)

                    {x[i4], x[i3]} := {x[i2]-t1, -x[i2]-t1}
                    {x[i1], x[i2]} := {x[i1]+t2,  x[i1]-t2}
                }

                i0 := i0 + id
            }

            ix := 2*id - n2
            id := 2*id
        }
        while ix<n

        e := 2.0*PI/n2
        a := e

        for j:=2 to n8
        {
            cc1 := cos(a)
            ss1 := sin(a)
            cc3 := cos(3*a)  // == 4*cc1*(cc1*cc1-0.75)
            ss3 := sin(3*a)  // == 4*ss1*(0.75-ss1*ss1)

            a := j*e

            ix := 0
            id := 2*n2

            do  // ix-loop
            {
                i0 := ix
                while  i0<n
                {
                    i1 := i0 + j - 1
                    i2 := i1 + n4
```

```
                    i3 := i2 + n4
                    i4 := i3 + n4
                    i5 := i0 + n4 - j + 1
                    i6 := i5 + n4
                    i7 := i6 + n4
                    i8 := i7 + n4

                    // complex mult: (t2,t1) := (x[i7],x[i3]) * (cc1,ss1)
                    t1 := x[i3]*cc1 + x[i7]*ss1
                    t2 := x[i7]*cc1 - x[i3]*ss1

                    // complex mult: (t4,t3) := (x[i8],x[i4]) * (cc3,ss3)
                    t3 := x[i4]*cc3 + x[i8]*ss3
                    t4 := x[i8]*cc3 - x[i4]*ss3

                    t5 := t1 + t3
                    t6 := t2 + t4
                    t3 := t1 - t3
                    t4 := t2 - t4

                    {t2, x[i3]} := {t6+x[i6], t6-x[i6]}
                    x[i8] := t2
                    {t2,x[i7]} := {x[i2]-t3, -x[i2]-t3}
                    x[i4] := t2
                    {t1, x[i6]} := {x[i1]+t5, x[i1]-t5}
                    x[i1] := t1
                    {t1, x[i5]} := {x[i5]+t4, x[i5]-t4}
                    x[i2] := t1

                    i0 := i0 + id
                }

                ix := 2*id - n2
                id := 2*id

            }
            while ix<n
        }
        nn := nn/2
    }
}
```

Complex to real split-radix FFT

Pseudo code for the split-radix C2RFT algorithm, the sign of the transform is hardcoded to $\sigma = -1$:

```
procedure c2rft_splitradix_dif(x[], ldn)
{
    n := 2**ldn

    n2 := n/2
    nn := n/4
    while nn!=0
    {
        ix := 0
        id := n2
        n2 := n2/2
        n4 := n2/4
        n8 := n2/8
        do  // ix loop
        {
            i0 := ix
            while i0<n
            {
                i1 := i0
                i2 := i1 + n4
                i3 := i2 + n4
                i4 := i3 + n4

                {x[i1], t1} := {x[i1]+x[i3], x[i1]-x[i3]}
                x[i2] := 2*x[i2]
                x[i4] := 2*x[i4]
                {x[i3], x[i4]} := {t1+x[i4], t1-x[i4]}

                if  n4!=1
                {
                    i1 := i1 + n8
                    i2 := i2 + n8
                    i3 := i3 + n8
                    i4 := i4 + n8

                    {x[i1], t1} := {x[i2]+x[i1], x[i2]-x[i1]}
                    {t2, x[i2]} := {x[i4]+x[i3], x[i4]-x[i3]}
                    x[i3] := -sqrt(2)*(t2+t1)
                    x[i4] :=  sqrt(2)*(t1-t2)
                }
```

```
                i0 := i0 + id
            }

            ix := 2*id - n2
            id := 2*id
        }
        while ix<n

        e := 2.0*PI/n2
        a := e

        for j:=2 to n8
        {
            cc1 := cos(a)
            ss1 := sin(a)
            cc3 := cos(3*a)  // == 4*cc1*(cc1*cc1-0.75)
            ss3 := sin(3*a)  // == 4*ss1*(0.75-ss1*ss1)
            a := j*e

            ix := 0
            id := 2*n2
            do  // ix-loop
            {
                i0 := ix
                while  i0<n
                {
                    i1 := i0 + j - 1
                    i2 := i1 + n4
                    i3 := i2 + n4
                    i4 := i3 + n4
                    i5 := i0 + n4 - j + 1
                    i6 := i5 + n4
                    i7 := i6 + n4
                    i8 := i7 + n4

                    {x[i1], t1} := {x[i1]+x[i6], x[i1]-x[i6]}
                    {x[i5], t2} := {x[i5]+x[i2], x[i5]-x[i2]}
                    {t3, x[i6]} := {x[i8]+x[i3], x[i8]-x[i3]}
                    {t4, x[i2]} := {x[i4]+x[i7], x[i4]-x[i7]}
                    {t1, t5} := {t1+t4, t1-t4}
                    {t2, t4} := {t2+t3, t2-t3}

                    // complex mult: (x[i7],x[i3]) := (t5,t4)  * (ss1,cc1)
                    x[i3] := t5*cc1 + t4*ss1
                    x[i7] := -t4*cc1 + t5*ss1

                    // complex mult: (x[i4],x[i8]) := (t1,t2)  * (cc3,ss3)
                    x[i4] := t1*cc3 - t2*ss3
                    x[i8] := t2*cc3 + t1*ss3

                    i0 := i0 + id
                }

                ix := 2*id - n2
                id := 2*id
            }
            while ix<n
        }

        nn := nn/2
    }

    ix := 1;
    id := 4;
    do
    {
        i0 := ix-1
        while i0<n
        {
            i1 := i0 + 1
            {x[i0], x[i1]} := {x[i0]+x[i1], x[i0]-x[i1]}
            i0 := i0 + id
        }
        ix := 2*id-1
        id := 4 * id
    }
    while ix<n
}
```

8.10 Multidimensional Fourier transforms

Definition

Let $a_{x,y}$ ($x = 0, 1, 2, \ldots, C-1$ and $y = 0, 1, 2, \ldots, R-1$) be a 2-dimensional array of data. That is, a $R \times C$ 'matrix' of R rows (of length C) and C columns (of length R). Its 2-dimensional Fourier transform is defined by:

$$c = \mathcal{F}\big[\, a \,\big] \tag{8.58a}$$

$$c_{k,h} := \frac{1}{\sqrt{n}} \sum_{x=0}^{C-1} \sum_{y=0}^{R-1} a_{x,y}\, z^{+(x\,k/C + y\,h/R)} \quad \text{where} \quad z = e^{\sigma\, 2\,\pi\, i} \tag{8.58b}$$

where $k \in \{0, 1, 2, \ldots, C-1\}$ and $h \in \{0, 1, 2, \ldots, R-1\}$. The inverse transform is

$$a = \mathcal{F}^{-1}\big[\, c \,\big] \tag{8.59a}$$

$$a_{x,y} = \frac{1}{\sqrt{n}} \sum_{k=0}^{C-1} \sum_{h=0}^{R-1} c_{k,h}\, z^{-(x\,k/C + y\,h/R)} \tag{8.59b}$$

For a m-dimensional array $a_{\vec{x}}$ (where $\vec{x} = (x_1, x_2, x_3, \ldots, x_m)$ and $x_i \in 0, 1, 2, \ldots, S_i$) the m-dimensional Fourier transform $c_{\vec{k}}$ (where $\vec{k} = (k_1, k_2, k_3, \ldots, k_m)$ and $k_i \in 0, 1, 2, \ldots, S_i$) is defined as

$$c_{\vec{k}} := \frac{1}{\sqrt{n}} \sum_{x_1=0}^{S_1-1} \sum_{x_2=0}^{S_2-1} \ldots \sum_{x_m=0}^{S_m-1} a_{\vec{x}}\, \exp\left(\sigma\, 2\,\pi\, i (x_1\, k_1/S_1 + x_2\, k_2/S_2 + \ldots + x_m\, k_m/S_m)\right) \tag{8.60a}$$

The inverse transform is, like in the 1-dimensional case, the complex conjugate transform.

The row-column algorithm

The equation of the definition of the two dimensional FT (relation 8.58a) can be recast as

$$c_{k,h} = \frac{1}{\sqrt{n}} \sum_{y=0}^{R-1} \left[\exp\left(y\, h/R\right) \sum_{x=0}^{C-1} a_{x,y}\, \exp\left(x\, k/C\right) \right] \tag{8.61}$$

which shows that the 2-dimensional FT can be obtained by first applying 1-dimensional transforms on the rows and then applying 1-dimensional transforms on the columns. The same result is obtained when the columns are transformed first and then the rows.

This leads us directly to the *row-column algorithm* for 2-dimensional FFTs. Pseudo code to compute the two dimensional FT of `a[][]` using the row-column method:

```
procedure rowcol_ft(a[][], R, C, is)
{
    complex a[R][C]  // R (length-C) rows, C (length-R) columns

    for r:=0 to R-1  // FFT rows
    {
        fft(a[r][], C, is)
    }

    complex t[R]     // scratch array for columns
    for c:=0 to C-1  // FFT columns
    {
        copy a[0,1,...,R-1][c] to t[]  // get column
        fft(t[], R, is)
        copy t[] to a[0,1,...,R-1][c]  // write back column
    }
}
```

Here it is assumed that the rows lie in contiguous memory (as in the C language). The equivalent C++ code is given in [FXT: file fft/twodimfft.cc].

Transposing the array before the column pass avoids the notorious problem with memory-cache and will do good for the performance in most cases. That is:

```
procedure rowcol_ft(a[][], R, C, is)
{
    complex a[R][C]  // R (length-C) rows, C (length-R) columns
    for r:=0 to R-1  // FFT rows
    {
        fft(a[r][], C, is)
    }
    transpose( a[R][C] )  // in-place
    for c:=0 to C-1  // FFT columns (which are rows now)
    {
        fft(a[c][], R, is)
    }
    transpose( a[C][R] )  // transpose back (note swapped R,C)
}
```

The transposing back at the end of the routine can be avoided if a back-transform will follow immediately as typical for convolution. The back-transform must then be called with `R` and `C` swapped.

The generalization to higher dimensions is straightforward, C++ code is given in [FXT: file fft/ndimfft.cc]

8.11 The matrix Fourier algorithm (MFA)

The matrix Fourier algorithm (MFA) is an algorithm for 1-dimensional FFTs that works for data lengths $n = RC$. It is quite similar to the row-column algorithm (relation 8.61 on the facing page) for 2-dimensional FFTs. The only differences are n multiplications with trigonometric factors and a final matrix transposition.

Consider the input array as a $R \times C$-matrix (R rows, C columns), the rows shall be contiguous in memory. Then the *matrix Fourier algorithm* (MFA) can be stated as follows:

1. Apply a (length R) FFT on each column.
2. Multiply each matrix element (index r, c) by $\exp(\sigma\, 2\,\pi\, i\, r\, c/n)$
3. Apply a (length C) FFT on each row.
4. Transpose the matrix.

Note the elegance! A variant of the MFA is called *four step FFT* in [10]. A trivial modification is obtained if the steps are executed in reversed order. The *transposed matrix Fourier algorithm* (TMFA) for the FFT:

1. Transpose the matrix.
2. Apply a (length C) FFT on each row of the matrix.
3. Multiply each matrix element (index r, c) by $\exp(\sigma\, 2\,\pi\, i\, r\, c/n)$.
4. Apply a (length R) FFT on each column of the matrix.

A variant of the MFA that, apart from the transpositions, accesses the memory only in consecutive address ranges can be stated as

1. Transpose the matrix.
2. Apply a (length C) FFT on each row of the transposed matrix.
3. Multiply each matrix element (index r, c) by $\exp(\sigma\, 2\,\pi\, i\, r\, c/n)$.

4. Transpose the matrix back.
5. Apply a (length R) FFT on each row of the matrix.
6. Transpose the matrix (if the order of the transformed data matters).

The 'transposed' version of this algorithm is identical. The performance will depend critically on the performance of the transposition routine.

It is usually a good idea to use factors of the data length n that are close to $\sqrt{n}$. Of course one can apply the same algorithm for the row (or column) FFTs again: It can be an improvement to split n into 3 factors (as close to $n^{1/3}$ as possible) if a length-$n^{1/3}$ FFT fits completely into the cache. Especially for systems where CPU clock speed is much higher than memory clock speed the performance may increase drastically, a speedup by a factor of three (even when compared to otherwise very well optimized FFTs) can sometimes be observed.

Chapter 9

Algorithms for fast convolution

This chapter gives several FFT based algorithms for fast convolution. These are in practice the most important applications of the FFT. Algorithmic variants like the weighted convolution are described. An efficient algorithm for the convolution of arrays that do not fit into the main memory (mass storage convolution) is given for both complex and real data. Further, weighted convolutions and their algorithms are introduced.

Near the end of the chapter we describe how fast convolution can be used for computing the z-transform of sequences of arbitrary length. Another convolution based algorithm for the Fourier transform of arrays of prime length, Rader's algorithm, is described at the end of the chapter.

Convolution algorithms based on the fast Hartley transform are described in section 12.7. The dyadic convolution, which is computed via the Walsh transform is treated in section 10.6.

9.1 Definition and computation via FFT

The *cyclic convolution* (or *circular convolution*) of two length-n sequences $a = [a_0, a_1, \ldots, a_{n-1}]$ and $b = [b_0, b1, \ldots, b_{n-1}]$ is defined as the length-n sequence h with elements h_τ as:

$$h = a \circledast b \tag{9.1a}$$

$$h_\tau := \sum_{x+y \equiv \tau \pmod n} a_x \, b_y \tag{9.1b}$$

The last equation may be rewritten as

$$h_\tau := \sum_{x=0}^{n-1} a_x \, b_{(\tau - x) \mod n} \tag{9.2}$$

That is, indices $\tau - x$ wrap around, it is a cyclic convolution. A convenient way to illustrate the cyclic convolution of two sequences is shown in figure 9.1 on the next page.

A C++ implementation of the computation by definition is [FXT: `slow_convolution()` in convolution/slowcnvl.h]:

```
template <typename Type>
void slow_convolution(const Type *f, const Type *g, Type *h, ulong n)
// (cyclic) convolution:  h[] :=  f[] (*) g[]
// n := array length
{
    for (ulong tau=0;  tau<n;  ++tau)
```

```
          +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
          |
          0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
          1:   1  2  3  4   5  6  7  8   9 10 11 12  13 14 15  0
          2:   2  3  4  5   6  7  8  9  10 11 12 13  14 15  0  1
          3:   3  4  5  6   7  8  9 10  11 12 13 14  15  0  1  2

          4:   4  5  6  7   8  9 10 11  12 13 14 15   0  1  2  3
          5:   5  6  7  8   9 10 11 12  13 14 15  0   1  2  3  4
          6:   6  7  8  9  10 11 12 13  14 15  0  1   2  3  4  5
          7:   7  8  9 10  11 12 13 14  15  0  1  2   3  4  5  6

          8:   8  9 10 11  12 13 14 15   0  1  2  3   4  5  6  7
          9:   9 10 11 12  13 14 15  0   1  2  3  4   5  6  7  8
         10:  10 11 12 13  14 15  0  1   2  3  4  5   6  7  8  9
         11:  11 12 13 14  15  0  1  2   3  4  5  6   7  8  9 10

         12:  12 13 14 15   0  1  2  3   4  5  6  7   8  9 10 11
         13:  13 14 15  0   1  2  3  4   5  6  7  8   9 10 11 12
         14:  14 15  0  1   2  3  4  5   6  7  8  9  10 11 12 13
         15:  15  0  1  2   3  4  5  6   7  8  9 10  11 12 13 14


     +--   0   1   2   3   (a)
     |
     0:    0   1   2   4
     1:    1   3   5 <--= h[5] contains a[2]*b[1]
     2:    4   8   9 <--= h[9] contains a[2]*b[b]
     3:    ...
   (b):
```

Figure 9.1: Semi-symbolic table of the cyclic convolution of two sequences (top). The entries denote where in the convolution the products of the input elements can be found (bottom).

```
    {
        Type s = 0.0;
        for (ulong k=0; k<n; ++k)
        {
            ulong k2 = tau - k;
            if ( (long)k2<0 )  k2 += n;
            s += (f[k]*g[k2]);
        }
        h[tau] = s;
    }
}
```

The following version avoids the `if` statement in the inner loop:

```
    for (ulong tau=0;  tau<n;  ++tau)
    {
        Type s = 0.0;
        ulong k = 0;
        for (ulong k2=tau;  k<=tau;  ++k, --k2)  s += (f[k]*g[k2]);
        for (ulong k2=n-1;  k<n;     ++k, --k2)  s += (f[k]*g[k2]);  // wrapped around
        h[tau] = s;
    }
```

For length-n sequences this procedure involves proportional n^2 operations, therefore it is slow for large values of n. The Fourier transform provides us with a more efficient way to compute convolutions that only uses proportional $n \log(n)$ operations.

Computation via FFT

First we have to state the convolution property of the Fourier transform:

$$\mathcal{F}\big[\, a \circledast b \,\big] \quad = \quad \mathcal{F}\big[\, a \,\big]\, \mathcal{F}\big[\, b \,\big] \tag{9.3}$$

That is, convolution in original space is element-wise multiplication in Fourier space. The statement can be motivated as follows:

$$\mathcal{F}\big[\,a\,\big]_k\,\mathcal{F}\big[\,b\,\big]_k \;=\; \sum_x a_x\,z^{k\,x}\sum_y b_y\,z^{k\,y} \;=\; \sum_x a_x\,z^{k\,x}\sum_{\tau-x} b_{\tau-x}\,z^{k\,(\tau-x)} \quad \text{where} \quad y=\tau-x \tag{9.4a}$$

$$=\; \sum_x\sum_{\tau-x} a_x\,z^{k\,x}b_{\tau-x}\,z^{k\,(\tau-x)} \;=\; \sum_\tau\left(\sum_x a_x\,b_{\tau-x}\right)z^{k\,\tau} \tag{9.4b}$$

$$=\; \left(\mathcal{F}\big[\sum_x a_x\,b_{\tau-x}\,\big]\right)_k \;=\; \big(\mathcal{F}\big[\,a\circledast b\,\big]\big)_k \tag{9.4c}$$

Rewriting relation 9.3 on the facing page as

$$a\circledast b \;=\; \mathcal{F}^{-1}\big[\,\mathcal{F}\big[\,a\,\big]\,\mathcal{F}\big[\,b\,\big]\,\big] \tag{9.5}$$

tells us how to proceed. Pseudo code for the cyclic convolution of two complex valued sequences `x[]` and `y[]`, result is returned in `y[]`:

```
procedure fft_cyclic_convolution(x[], y[], n)
{
    complex x[0..n-1], y[0..n-1]

    // transform data:
    fft(x[], n, +1)
    fft(y[], n, +1)

    // convolution in transformed domain:
    for i:=0 to n-1
    {
        y[i] := y[i] * x[i]
    }

     // transform back:
    fft(y[], n, -1)

    // normalize:
    for i:=0 to n-1
    {
        y[i] := y[i] / n
    }
}
```

It is assumed that the procedure `fft()` does no normalization. In the normalization loop you precompute `1.0/n` and multiply as divisions are much slower than multiplications.

Avoiding the revbin permutations

One can save the revbin permutations by observing that any DIF FFT is of the form

```
    DIF_FFT_CORE(f, n);
    revbin_permute(f, n);
```

and any DIT FFT is of the form

```
    revbin_permute(f, n);
    DIT_FFT_CORE(f, n);
```

Thereby a convolution routine that uses DIF FFTs for the forward transform and DIT FFTs as backward transform can omit the revbin permutations as demonstrated in the C++ implementation for the cyclic convolution of complex sequences [FXT: `fft_complex_convolution()` in convolution/fftcocnvl.cc]:

```
#define  DIT_FFT_CORE  fft_dit4_core
#define  DIF_FFT_CORE  fft_dif4_core
void
fft_complex_convolution(Complex * restrict f, Complex * restrict g,
                        ulong ldn, double v/*=0.0*/)
//
// (complex, cyclic) convolution:  g[] :=  f[] (*) g[]
// (use zero padded data for usual convolution)
// ldn := base-2 logarithm of the array length
```

```
//
// Supply a value for v for a normalization factor != 1/n
{
    const ulong n = (1UL<<ldn);

    DIF_FFT_CORE(f, ldn);
    DIF_FFT_CORE(g, ldn);
    if ( v==0.0 )  v = 1.0/n;
    for (ulong i=0; i<n; ++i)
    {
        Complex t = g[i] * f[i];
        g[i] = t * v;
    }
    DIT_FFT_CORE(g, ldn);
}
```

Relation 9.3 on page 286 is also true for the more general z-transform (see section 9.8 on page 298), but there is no back-transform, so we cannot turn

$$\mathcal{Z}\,[\,a \circledast b\,] \quad = \quad \mathcal{Z}\,[\,a\,]\;\mathcal{Z}\,[\,b\,] \tag{9.6}$$

into a practical algorithm for convolution.

The so-called *auto convolution* (or *self convolution*) of a sequence is defined as the convolution of a sequence with itself: $h = a \circledast a$. The corresponding procedure needs only two instead of three FFTs.

Weighted convolutions

In the definition of the cyclic convolution (relations 9.1a and 9.1b) one can distinguish between those summands where the $x+y$ 'wrapped around' (i.e. $x+y = n+\tau$) and those where simply $x+y = \tau$ holds. These are (following the notation in [39]) denoted by $h^{(1)}$ and $h^{(0)}$ respectively. We have

$$h \quad = \quad h^{(0)} + h^{(1)} \qquad \text{where} \tag{9.7a}$$

$$h^{(0)} \quad = \quad \sum_{x \le \tau} a_x\, b_{\tau - x} \tag{9.7b}$$

$$h^{(1)} \quad = \quad \sum_{x > \tau} a_x\, b_{n+\tau - x} \tag{9.7c}$$

There is a simple way to separate $h^{(0)}$ and $h^{(1)}$ as the left and right half of a length-$2\,n$ sequence. This is just what the *acyclic convolution* (or *linear convolution*) does: Acyclic convolution of two (length-n) sequences a and b can be defined as that length-$2\,n$ sequence h which is the cyclic convolution of the *zero padded* sequences A and B:

$$A \quad := \quad [a_0, a_1, a_2, \ldots, a_{n-1}, 0, 0, \ldots, 0] \tag{9.8}$$

Same for B. Then the acyclic convolution is defined as

$$h \quad = \quad a \circledast_{ac} b \tag{9.9a}$$

$$h_\tau \quad := \quad \sum_{x=0}^{2\,n-1} A_x\, B_{\tau - x} \qquad \tau = 0, 1, 2, \ldots, 2\,n-1 \tag{9.9b}$$

As an illustration consider the convolution of the sequence $[1, 1, 1, 1]$ with itself: its linear self convolution is the length-8 sequence $[h_0][h_1] = [1, 2, 3, 4][3, 2, 1, 0]$, its cyclic self convolution is $[h_0 + h_1] = [4, 4, 4, 4]$.

The semi-symbolic table for the acyclic convolution is given in figure 9.2 on the next page. The elements in the lower right triangle do not 'wrap around' anymore, they go to extra buckets. Note there are 31 buckets labeled $0 \ldots 30$.

```
+--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
|
0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
1:   1  2  3  4   5  6  7  8   9 10 11 12  13 14 15 16
2:   2  3  4  5   6  7  8  9  10 11 12 13  14 15 16 17
3:   3  4  5  6   7  8  9 10  11 12 13 14  15 16 17 18

4:   4  5  6  7   8  9 10 11  12 13 14 15  16 17 18 19
5:   5  6  7  8   9 10 11 12  13 14 15 16  17 18 19 20
6:   6  7  8  9  10 11 12 13  14 15 16 17  18 19 20 21
7:   7  8  9 10  11 12 13 14  15 16 17 18  19 20 21 22

8:   8  9 10 11  12 13 14 15  16 17 18 19  20 21 22 23
9:   9 10 11 12  13 14 15 16  17 18 19 20  21 22 23 24
10: 10 11 12 13  14 15 16 17  18 19 20 21  22 23 24 25
11: 11 12 13 14  15 16 17 18  19 20 21 22  23 24 25 26

12: 12 13 14 15  16 17 18 19  20 21 22 23  24 25 26 27
13: 13 14 15 16  17 18 19 20  21 22 23 24  25 26 27 28
14: 14 15 16 17  18 19 20 21  22 23 24 25  26 27 28 29
15: 15 16 17 18  19 20 21 22  23 24 25 26  27 28 29 30
```

Figure 9.2: Semi-symbolic table for the (length-31) acyclic convolution of two length-16 sequences.

Linear convolution is *polynomial multiplication*: Let $A = a_0 + a_1\,x + a_2\,x^2 + \ldots$, $B = b_0 + b_1\,x + b_2\,x^2 + \ldots$ and $C = A\,B = c_0 + c_1\,x + c_2\,x^2 + \ldots$ then

$$c_k \quad = \quad \sum_{i+j=k} a_i\,b_j \tag{9.10}$$

Chapter 14 on page 389 explains how fast convolution algorithms can be used for fast multiplication of multiprecision numbers.

9.2 Correlation

```
+--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
|
0:   0 15 14 13  12 11 10  9   8  7  6  5   4  3  2  1
1:   1  0 15 14  13 12 11 10   9  8  7  6   5  4  3  2
2:   2  1  0 15  14 13 12 11  10  9  8  7   6  5  4  3
3:   3  2  1  0  15 14 13 12  11 10  9  8   7  6  5  4

4:   4  3  2  1   0 15 14 13  12 11 10  9   8  7  6  5
5:   5  4  3  2   1  0 15 14  13 12 11 10   9  8  7  6
6:   6  5  4  3   2  1  0 15  14 13 12 11  10  9  8  7
7:   7  6  5  4   3  2  1  0  15 14 13 12  11 10  9  8

8:   8  7  6  5   4  3  2  1   0 15 14 13  12 11 10  9
9:   9  8  7  6   5  4  3  2   1  0 15 14  13 12 11 10
10: 10  9  8  7   6  5  4  3   2  1  0 15  14 13 12 11
11: 11 10  9  8   7  6  5  4   3  2  1  0  15 14 13 12

12: 12 11 10  9   8  7  6  5   4  3  2  1   0 15 14 13
13: 13 12 11 10   9  8  7  6   5  4  3  2   1  0 15 14
14: 14 13 12 11  10  9  8  7   6  5  4  3   2  1  0 15
15: 15 14 13 12  11 10  9  8   7  6  5  4   3  2  1  0
```

Figure 9.3: Semi-symbolic table for the (cyclic) correlation of two length-16 sequences.

The *cyclic correlation* (or *circular correlation*) of two length-n sequences $a = [a_0, a_1, \ldots, a_{n-1}]$ and $b = [b_0, b1, \ldots, b_{n-1}]$ is defined as the length-n sequence h with elements h_τ as:

$$h \quad = \quad a * b \tag{9.11a}$$

$$h_\tau \quad := \quad \sum_{x-y \equiv \tau \pmod n} a_x\,b_y \tag{9.11b}$$

```
+--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
|
0:   0 31 30 29  28 27 26 25  24 23 22 21  20 19 18 17
1:   1  0 31 30  29 28 27 26  25 24 23 22  21 20 19 18
2:   2  1  0 31  30 29 28 27  26 25 24 23  22 21 20 19
3:   3  2  1  0  31 30 29 28  27 26 25 24  23 22 21 20

4:   4  3  2  1   0 31 30 29  28 27 26 25  24 23 22 21
5:   5  4  3  2   1  0 31 30  29 28 27 26  25 24 23 22
6:   6  5  4  3   2  1  0 31  30 29 28 27  26 25 24 23
7:   7  6  5  4   3  2  1  0  31 30 29 28  27 26 25 24

8:   8  7  6  5   4  3  2  1   0 31 30 29  28 27 26 25
9:   9  8  7  6   5  4  3  2   1  0 31 30  29 28 27 26
10: 10  9  8  7   6  5  4  3   2  1  0 31  30 29 28 27
11: 11 10  9  8   7  6  5  4   3  2  1  0  31 30 29 28

12: 12 11 10  9   8  7  6  5   4  3  2  1   0 31 30 29
13: 13 12 11 10   9  8  7  6   5  4  3  2   1  0 31 30
14: 14 13 12 11  10  9  8  7   6  5  4  3   2  1  0 31
15: 15 14 13 12  11 10  9  8   7  6  5  4   3  2  1  0
```

Figure 9.4: Semi-symbolic table for the linear (acyclic) correlation of two length-16 sequences.

The last relation can be recast as

$$h_\tau \quad := \quad \sum_{x=0}^{n-1} a_x \, b_{(\tau+x) \mod n} \tag{9.12}$$

Again, indices $\tau + x$ wrap around. The semi-symbolic table for the (cyclic) correlation is shown in figure 9.3 on the previous page. For the computation of the acyclic correlation the sequences have to be zero-padded as in the algorithm for the acyclic convolution. The semi-symbolic table is shown in figure 9.4.

A C++ implementation of the slow (that is, proportional n^2) algorithm is [FXT: file correlation/slowcorr.h]:

```
template <typename Type>
void slow_correlation(const Type *f, const Type *g, Type * restrict h, ulong n)
// compute (cyclic) correlation of f[], g[]
// n := array length
{
    for (ulong tau=0;  tau<n;  ++tau)
    {
        Type s = 0.0;
        for (ulong k=0; k<n; ++k)
        {
             ulong k2 = k + tau;
             if ( k2>=n )  k2 -= n;
             s += (g[k]*f[k2]);
        }
        h[tau] = s;
    }
}
```

The `if` statement in the inner loop is avoided by the following version:

```
    for (ulong tau=0;  tau<n;  ++tau)
    {
        Type s = 0.0;
        ulong k = 0;
        for (ulong k2=tau;  k2<n;  ++k, ++k2)  s += (g[k]*f[k2]);
        for (ulong k2=0;     k<n;  ++k, ++k2)  s += (g[k]*f[k2]);
        h[tau] = s;
    }
```

A simple algorithm for fast correlation follows from the relation

$$h_\tau \quad = \quad \mathcal{F}^{-1}\big[\, \mathcal{F}\big[\, a \,\big]\, \mathcal{F}\big[\, \bar{b} \,\big]\,\big] \tag{9.13}$$

That is, use a convolution algorithm with one of the input sequences reversed (indices negated modulo n). For purely real sequences the relation is equivalent to complex conjugation of one of the inner transforms:

$$h_\tau \quad = \quad \mathcal{F}^{-1}\big[\,\mathcal{F}\big[\,a\,\big]\,\mathcal{F}\big[\,b\,\big]^*\,\big] \tag{9.14}$$

For the computation of self-correlation the latter relation is the only reasonable way to go: First transform the input sequence, then multiply each element by its complex conjugate and finally transform back. A C++ implementation is [FXT: file correlation/fftcorr.cc]:

```
void
fft_correlation(double *f, double *g, ulong ldn)
// Compute (real, cyclic) correlation of f[], g[]
// Result is in g[]
// ldn := base-2 logarithm of the array length
{
    const ulong n=(1UL<<ldn);
    const ulong nh=(n>>1);

    fht_real_complex_fft(f, ldn);  // real, imag part in lower, upper half
    fht_real_complex_fft(g, ldn);

    const double v = 1.0/n;
    g[0]  *= f[0] * v;
    g[nh] *= f[nh] * v;
    for (ulong i=1,j=n-1; i<nh;  ++i,--j)  // real at index i, imag at index j
    {
        cmult_n(f[i], -f[j], g[i], g[j], v);
    }

    fht_complex_real_fft(g, ldn);
}
```

The function `cmult_n()` is defined in [FXT: file aux0/cmult.h]:

```
static inline void
cmult_n(double c, double s, double &u, double &v, double dn)
// {u,v} <--| {dn*(u*c-v*s), dn*(u*s+v*c)}
{ double t = u*s+v*c;  u *= c;  u -= v*s;  u *= dn;  v = t*dn; }
```

9.3 Mass storage convolution using the MFA

The matrix Fourier algorithm is an ideal candidate for a *mass storage FFT*, that is an FFT for a data set that does not fit into physical RAM. Algorithms on data sets that do not fit into physical RAM are sometimes called *external* or *out of core*. The naive idea to simply try such an FFT with the virtual memory mechanism of the operating system will end in eternal hard disk activity because of the non-locality of the FFT algorithms.

With convolution computations that use the matrix Fourier algorithm (MFA, see section 8.11 on page 283) for the forward transform and the transposed algorithm (TMFA) for the backward transform one can actually omit all transpositions. The elements of each row are assumed to lie contiguous in memory. For the sake of simplicity auto convolution is considered.

The matrix FFT convolution algorithm:

1. Apply a (length R) FFT on each column.
   (*memory access with C-skips*)
2. Multiply each matrix element (index r, c) by $\exp(+\sigma\, 2\,\pi\, i\, r\, c/n)$.
3. Apply a (length C) FFT on each row.
   (*memory access without skips*)
4. Complex square row (element-wise).
5. Apply a (length C) FFT on each row (of the transposed matrix).
   (*memory access is without skips*)

6. Multiply each matrix element (index r, c) by $\exp(-\sigma\, 2\,\pi\, i\, r\, c/n)$.
7. Apply a (length R) FFT on each column (of the transposed matrix).
   (*memory access with C-skips*)

Note that steps 3, 4 and 5 constitute a length-C convolution on each row.

C++ implementations of the cyclic and linear convolution using the described algorithm are given in [FXT: file convolution/matrixfftcnvl.cc]. The code for self-convolution can be found in [FXT: file convolution/matrixfftcnvla.cc].

The algorithm can be used for mass storage convolutions if care is taken to minimize the number of disk seeks.

Minimization of the number of disk seeks

The number of disk seeks has to be kept minimal because these are slow operations which, if occur too often, degrade performance unacceptably.

The crucial modification of the use of the MFA is *not* to choose R and C as close as possible to $\sqrt{n}$ as is usually done. Instead one chooses R to be minimal so that the row length C corresponds to the biggest data set that fits into the available RAM. We now analyze how the number of seeks depends on the choice of R and C: In what follows it is assumed that the data lies in memory as row_0, row_1, ..., row_{R-1}. In other words, the elements of each rows lie contiguous in memory. Further let $\alpha \geq 2$ be the number of times the data set exceeds the available RAM size.

In step 1 and 3 of the convolution algorithm given on page 296 one reads from disk (row by row, involving R seeks) the number of columns that just fit into RAM, does the (many, short) column-FFTs, writes back (again R seeks) and proceeds to the next block; this happens for α of these blocks, giving a total of $4\,\alpha\, R$ seeks for steps 1 and 3.

In step 2 one has to read (α times) blocks of one or more rows, which lie in contiguous portions of the disk, perform the FFT on the rows and write back to disk, leading to a total of $2\,\alpha$ seeks.

Thereby one has a number of $2\,\alpha + 4\,\alpha\, R$ seeks during the whole computation, which is minimized by the choice of maximal C. This means that one chooses a shape of the matrix so that the rows are as big as possible subject to the constraint that they have to fit into main memory, which in turn means there are $R = \alpha$ rows, leading to an optimal seek count of $K = 2\,\alpha + 4\,\alpha^2$.

Let S_D be the seek time of the hard disk in seconds, W_D be the disk transfer rate in megabyte per second. Further let A be the amount of available RAM in megabytes. Then the total time spent for disk operations is the sums of the time spent in seeks and the time for reading and writing:

$$T = K\, S_D + 6\, \frac{\alpha\, A}{W_D} \tag{9.15a}$$

$$= \left(2\,\alpha + 4\,\alpha^2\right)\, S_D + 6\, \frac{\alpha\, A}{R_D} \tag{9.15b}$$

Two examples for an FFT whose size that exceeds the available RAM by a factor of $\alpha = 16$ (so the number of seeks is $k = 1056$):

Not so fancy machine: Let the disk seek take 10 milliseconds ($S_D = 0.010$) then about 10 seconds are needed for the seeks. Further assume we have $A = 64$ Megabytes of RAM available so the transform size is 1 GB corresponding to 128 million (double precision) floats. The disk transfer rate shall be $W_D = 10$ MB/sec. Then the overhead for the read and write would amount to $6{\cdot}1024MB/(10MB/sec) \approx 615sec$ or approximately 10 minutes. So the total disk activity takes approximately 10.5 minutes.

Workstation machine with $S_D = 0.005$ sec (5 milliseconds), $W_d = 50$ MB/sec and 512 MB: The transform size is 8 GB (1 G doubles) and we get $T = 5.28 + 983.04 = 988.32$ or approximately 16.5 minutes.

In both cases the CPU will be busy for several minutes which is in the same order as time for the disk activity. Therefore, with a multi-threading OS one may want to use a *double buffer* variant: Choose the row length so that it fits twice into the RAM workspace; then let always one (CPU-intensive) thread do the FFTs in one of the scratch spaces and the other (hard disk intensive) thread write back the data from the other scratch space and read the next data to be processed. With not too small main memory (and not too slow hard disk) and some fine tuning this should allow to keep the CPU busy during much of the hard disk operations. Thereby most of the disk time can be 'hidden' as the CPU activity does not stall waiting for data.

The mass storage convolution as described was used for the calculation of the number

$$9^{9^9} \approx 0.4281247 \cdot 10^{369,693,100} \tag{9.16}$$

on a 32-bit machine in 1999. The computation used two files of size 2 GigaBytes each and took less than eight hours on a system with a AMD K6/2 CPU at 366 MHz with 66 MHz memory. The log-file of the computation is [hfloat: examples/run1-pow999.txt]. The computation did not use the double-buffer technique.

9.4 Weighted Fourier transforms and convolutions

Let us define a new kind of transform by slightly modifying the definition of the FT (formula 8.1a on page 253):

$$c = \mathcal{W}_v[a] \tag{9.17a}$$

$$c_k := \sum_{x=0}^{n-1} v_x\, a_x\, z^{x\,k} \qquad v_x \neq 0 \quad \forall x \tag{9.17b}$$

where $z := e^{\sigma\, 2\,\pi\, i/n}$. The sequence c shall be called (discrete) *weighted transform* of the sequence a with the weight (sequence) v. Note the v_x that entered: the weighted transform with $v_x = \frac{1}{\sqrt{n}}\ \forall x$ is just the usual Fourier transform. The inverse transform is

$$a = \mathcal{W}_v^{-1}[c] \tag{9.18a}$$

$$a_x = \frac{1}{n\, v_x} \sum_{k=0}^{n-1} c_k\, z^{-x\,k} \tag{9.18b}$$

This can be easily seen:

$$\mathcal{W}_v^{-1}\left[\mathcal{W}_v[a]\right]_y = \frac{1}{n\, v_y} \sum_{k=0}^{n-1} \sum_{x=0}^{n-1} v_x\, a_x\, z^{x\,k}\, z^{-y\,k} \tag{9.19a}$$

$$= \frac{1}{n} \sum_{k=0}^{n-1} \sum_{x=0}^{n-1} v_x\, \frac{1}{v_y}\, a_x\, z^{x\,k}\, z^{-y\,k} \tag{9.19b}$$

$$= \frac{1}{n} \sum_{x=0}^{n-1} v_x\, \frac{1}{v_y}\, a_x\, \delta_{x,y}\, n \quad = a_y \tag{9.19c}$$

Obviously all v_x have to be invertible. That $\mathcal{W}_v\left[\mathcal{W}_v^{-1}[a]\right]$ is also identity is apparent from the definitions.

Given an FFT routine it is trivial to set up a weighted Fourier transform. Pseudo code for the discrete weighted Fourier transform:

```
procedure weighted_ft(a[], v[], n, is)
{
    for x:=0 to n-1
```

```
    {
        a[x] := a[x] * v[x]
    }
    fft(a[], n, is)
}
```

The inverse is essentially identical. Pseudo code for the inverse discrete weighted Fourier transform:

```
procedure inverse_weighted_ft(a[], v[], n, is)
{
    fft(a[], n, -is)
    for x:=0 to n-1
    {
        a[x] := a[x] / v[x]
    }
}
```

The C++ implementations are given in [FXT: file fft/weightedfft.cc].

Define the *weighted (cyclic) convolution* h_v by

$$h_v = a \circledast_{\{v\}} b \tag{9.20a}$$
$$= \mathcal{W}_v^{-1}\left[\mathcal{W}_v\left[a\right]\,\mathcal{W}_v\left[b\right]\right] \tag{9.20b}$$

Then, for the special case $v_x = V^x$, one has

$$h_v = h^{(0)} + V^n\, h^{(1)} \tag{9.21}$$

Here $h^{(0)}$ and $h^{(1)}$ are defined as in relation 9.7a on page 288. It is not hard to see why this is: Up to the final division by the weight sequence, the weighted convolution is just the cyclic convolution of the two weighted sequences, which is for the element with index τ equal to

$$\sum_{x+y\equiv\tau \mod n} (a_x\, V^x)\,(b_y\, V^y) = \sum_{x\le\tau} a_x\, b_{\tau-x}\, V^\tau + \sum_{x>\tau} a_x\, b_{n+\tau-x}\, V^{n+\tau} \tag{9.22}$$

Final division of this element (by V^τ) gives $h^{(0)} + V^n\, h^{(1)}$ as stated.

The cases when V^n is some root of unity are particularly interesting: For $V^n = \pm i = \pm\sqrt{-1}$ one gets the so-called *right-angle convolution*:

$$h_v = h^{(0)} \mp i\, h^{(1)} \tag{9.23}$$

This gives a nice possibility to directly use complex FFTs for the computation of a linear (acyclic) convolution of two real sequences: For length-n sequences the elements of the linear convolution with indices $0, 1, \ldots, n-1$ are the real part of the result, the elements $n, n+1, \ldots, 2\,n-1$ are the imaginary part. Choosing $V^n = -1$ leads to the *negacyclic convolution* (or *skew circular convolution*):

$$h_v = h^{(0)} - h^{(1)} \tag{9.24}$$

Cyclic, negacyclic and right-angle convolution can be understood as polynomial products modulo the polynomials $z^n - 1$, $z^n + 1$ and $z^n \pm i$, respectively (see [Nussb]).

C++ implementations of the weighted-, negacyclic- and right-angle (self) convolution are given in [FXT: file convolution/weightedconv.cc].

The semi-symbolic table for the negacyclic convolution is shown in figure 9.5 on the facing page. With right-angle convolution the minuses have to be replaced by $i = \sqrt{-1}$ which means the wrap-around (i.e. $h^{(1)}$) elements go to the imaginary part. With real input one thereby effectively separates $h^{(0)}$ and $h^{(1)}$. Thereby the linear convolution of real sequences can be computed using the complex right-angle convolution.

With routines for both cyclic and negacyclic convolution the parts $h^{(0)}$ and $h^{(1)}$ can be computed as sum and difference, respectively. Thereby all expressions of the form $\alpha\, h^{(0)} + \beta\, h^{(1)}$ where $\alpha, \beta \in \mathbb{C}$ can be computed.

```
+--   0   1   2   3    4   5   6   7    8   9  10  11   12  13  14  15
|
0:    0   1   2   3    4   5   6   7    8   9  10  11   12  13  14  15
1:    1   2   3   4    5   6   7   8    9  10  11  12   13  14  15   0-
2:    2   3   4   5    6   7   8   9   10  11  12  13   14  15   0-  1-
3:    3   4   5   6    7   8   9  10   11  12  13  14   15   0-  1-  2-

4:    4   5   6   7    8   9  10  11   12  13  14  15    0-  1-  2-  3-
5:    5   6   7   8    9  10  11  12   13  14  15   0-   1-  2-  3-  4-
6:    6   7   8   9   10  11  12  13   14  15   0-  1-   2-  3-  4-  5-
7:    7   8   9  10   11  12  13  14   15   0-  1-  2-   3-  4-  5-  6-

8:    8   9  10  11   12  13  14  15    0-  1-  2-  3-   4-  5-  6-  7-
9:    9  10  11  12   13  14  15   0-   1-  2-  3-  4-   5-  6-  7-  8-
10:  10  11  12  13   14  15   0-  1-   2-  3-  4-  5-   6-  7-  8-  9-
11:  11  12  13  14   15   0-  1-  2-   3-  4-  5-  6-   7-  8-  9- 10-

12:  12  13  14  15    0-  1-  2-  3-   4-  5-  6-  7-   8-  9- 10- 11-
13:  13  14  15   0-   1-  2-  3-  4-   5-  6-  7-  8-   9- 10- 11- 12-
14:  14  15   0-  1-   2-  3-  4-  5-   6-  7-  8-  9-  10- 11- 12- 13-
15:  15   0-  1-  2-   3-  4-  5-  6-   7-  8-  9- 10-  11- 12- 13- 14-
```

Figure 9.5: Semi-symbolic table for the negacyclic convolution. The products that enter with negative sign are indicated with a postfix minus at the corresponding entry.

The direct (slow, proportional n^2) computation can be obtained by a minimal modification of the non-weighted convolution algorithm [FXT: file convolution/slowweightedcnvl.h]:

```
template <typename Type>
void slow_weighted_convolution(const Type *f, const Type *g, Type *h, ulong n, Type w)
// weighted (cyclic) convolution:  h[] :=  f[] (*)_w g[]
// n := array length
{
    for (ulong tau=0;  tau<n;  ++tau)
    {
        ulong k = 0;
        Type s0 = 0.0;
        for (ulong k2=tau;  k<=tau;  ++k, --k2)  s0 += (f[k]*g[k2]);
        Type s1 = 0.0;
        for (ulong k2=n-1;  k<n;     ++k, --k2)  s1 += (f[k]*g[k2]);  // wrapped around
        h[tau] = s0 + s1*w;
    }
}
```

9.5 Half cyclic convolution for half the price?

The computation of $h^{(0)}$ from formula 9.7a (without computing $h^{(1)}$) is called *half cyclic convolution*. Apparently, one asks for less information than one gets from the acyclic convolution. One might hope to find an algorithm that computes $h^{(0)}$ and uses only half the memory compared to the linear convolution or that needs half the work, possibly both. It may be a surprise that no such algorithm seems to be known.

Here is a clumsy attempt to find $h^{(0)}$ alone: Use the weighted transform with the weight sequence $v_x = V^x$ where V^n is very small. Then $h^{(1)}$ will in the result be multiplied with a small number and we hope to make it almost disappear. Indeed, using $V^n = 1000$ for the cyclic self convolution of the sequence $\{1, 1, 1, 1\}$ (where for the linear self convolution $h^{(0)} = \{1, 2, 3, 4\}$ and $h^{(1)} = \{3, 2, 1, 0\}$) one gets $\{1.003, 2.002, 3.001, 4.000\}$. At least for integer sequences one could choose V^n (more than two times) bigger than biggest possible value in $h^{(1)}$ and use rounding to nearest integer to isolate $h^{(0)}$. Alas, even for modest sized arrays numerical overflow and underflow give spurious results. Careful analysis shows that this idea leads to an algorithm far worse than simply using linear convolution.

The direct (slow) algorithm can of course be modified to compute $h^{(0)}$ or $h^{(1)}$. A C++ implementation is [FXT: file convolution/slowcnvlhalf.h]:

```
template <typename Type>
```

```
void slow_half_convolution(const Type *f, const Type *g, Type *h, ulong n, int h01)
// Half cyclic convolution.
// Part determined by h01 which must be 0 or 1.
// n := array length
{
    if ( 0==h01 )  // compute h0:
    {
        for (ulong tau=0;  tau<n;  ++tau)
        {
            Type s0 = 0.0;
            for (ulong k=0, k2=tau;  k<=tau;  ++k, --k2)  s0 += (f[k]*g[k2]);
            h[tau] = s0;
        }
    }
    else  // compute h1 (wrapped part):
    {
        for (ulong tau=0;  tau<n;  ++tau)
        {
            Type s1 = 0.0;
            for (ulong k2=n-1, k=tau+1;  k<n;  ++k, --k2)  s1 += (f[k]*g[k2]);
            h[tau] = s1;
        }
    }
}
```

9.6 Convolution using the MFA

With the weighted convolutions in mind we reformulate the matrix (self-) convolution algorithm given on page 291:

1. Apply a FFT on each column.
2. On each row apply the weighted convolution with $V^C = e^{2\pi i r/R} = 1^{r/R}$ where R is the total number of rows, $r = 0..R-1$ the index of the row, C the length of each row (equivalently, the total number columns)
3. Apply a FFT on each column (of the transposed matrix).

We first consider the special cases of two and three rows and then formulate an MFA-based algorithm for the convolution of real sequences.

9.6.1 The case $R = 2$

Define s and d as the sums and differences of the lower and higher halves of a given sequence x:

$$s := x^{(0/2)} + x^{(1/2)} \tag{9.25a}$$
$$d := x^{(0/2)} - x^{(1/2)} \tag{9.25b}$$

Then the cyclic auto convolution of the sequence x can be obtained by two half-length convolutions of s and d as

$$x \circledast x = \frac{1}{2}\left[s \circledast s + d \circledast_- d, \quad s \circledast s - d \circledast_- d\right] \tag{9.26}$$

where the symbols $\circledast$ and $\circledast_-$ stand for cyclic and negacyclic convolution, respectively (see section 9.4 on page 293). The equivalent formula for the cyclic convolution of two sequences x and y is

$$x \circledast y = \frac{1}{2}\left[s_x \circledast s_y + d_x \circledast_- d_y, \quad s_x \circledast s_y - d_x \circledast_- d_y\right] \tag{9.27}$$

where

$$\begin{aligned} s_x &:= x^{(0/2)} + x^{(1/2)} && (9.28a)\\ d_x &:= x^{(0/2)} - x^{(1/2)} && (9.28b)\\ s_y &:= y^{(0/2)} + y^{(1/2)} && (9.28c)\\ d_y &:= y^{(0/2)} - y^{(1/2)} && (9.28d) \end{aligned}$$

Now use the fact that an acyclic convolution is computed by a cyclic convolution of zero-padded sequences whose upper halves are simply zero, so $s_x = d_x = x$ and $s_y = d_y = y$. Then relation 9.27 reads:

$$x \circledast_{ac} y = \frac{1}{2}\left[x \circledast y + x \circledast_- y, \quad x \circledast y - x \circledast_- y\right] \tag{9.29}$$

And for the acyclic auto convolution:

$$x \circledast_{ac} x = \frac{1}{2}\left[x \circledast x + x \circledast_- x, \quad x \circledast x - x \circledast_- x\right] \tag{9.30}$$

The lower and upper halves of the acyclic convolution can be obtained from the sum and difference of the cyclic and the negacyclic convolution.

9.6.2 The case $R = 3$

Let $\omega = \frac{1}{2}(1 + i\sqrt{3})$ and define

$$\begin{aligned} A &:= x^{(0/3)} + x^{(1/3)} + x^{(2/3)} && (9.31a)\\ B &:= x^{(0/3)} + \omega\, x^{(1/3)} + \omega^2 x^{(2/3)} && (9.31b)\\ C &:= x^{(0/3)} + \omega^2 x^{(1/3)} + \omega\, x^{(2/3)} && (9.31c) \end{aligned}$$

Then, if $h := x \circledast x$, one has

$$\begin{aligned} h^{(0/3)} &= A \circledast A + B \circledast_{\{\omega\}} B + C \circledast_{\{\omega^2\}} C && (9.32a)\\ h^{(1/3)} &= A \circledast A + \omega^2 B \circledast_{\{\omega\}} B + \omega\, C \circledast_{\{\omega^2\}} C && (9.32b)\\ h^{(2/3)} &= A \circledast A + \omega\, B \circledast_{\{\omega\}} B + \omega^2 C \circledast_{\{\omega^2\}} C && (9.32c) \end{aligned}$$

For real valued data C is the complex conjugate (*cc.*) of B and (with $\omega^2 = cc.\omega$) $B \circledast_{\{\omega\}} B$ is the *cc.* of $C \circledast_{\{\omega^2\}} C$ and therefore every $B \circledast_{\{.\}} B$-term is the cc. of the $C \circledast_{\{.\}} C$-term in the same line. Is there a nice and general scheme for real valued convolutions based on the MFA? Read on for the positive answer.

9.7 Convolution of real valued data using the MFA

Consider the MFA-algorithm for the cyclic convolution as given on page 296 but with real input data:

For row 0 which is real after the column FFTs one needs to compute the usual cyclic convolution; for row $R/2$ which is also purely real after the column FFTs a negacyclic convolution is needed[1], the code for negacyclic convolution is given on page 370.

All other weighted convolutions involve complex computations, but it is easy to see how to reduce the work by 50 percent: As the result must be real the data in row number $R - r$ must, because of the symmetries of the real and imaginary part of the (inverse) Fourier transform of real data, be the complex conjugate of the data in row r. Therefore one can use real FFTs (R2CFTs) for all column-transforms for step 1 and half-complex to real FFTs (C2RFTs) for step 3.

Let the computational cost of a cyclic (real) convolution be q, then

[1] For odd values of R there is no such row and no negacyclic convolution is needed.

- For even values of R one must perform one cyclic (row 0), one negacyclic (row $R/2$) and $R/2-2$ complex weighted convolutions (rows $1, 2, \ldots, R/2-1$)
- For R odd one must perform 1 cyclic (row 0) and $(R-1)/2$ complex weighted convolutions (rows $1, 2, \ldots, (R-1)/2$)

Now assume, slightly simplifying, that the cyclic and the negacyclic real convolution involve the same number of computations and that the cost of a weighted complex convolution is twice that. Then in both cases above the total work is exactly half of that for the complex case, which is what one expects from a real world real valued convolution algorithm.

For acyclic convolution one may want to use the right angle convolution (and complex FFTs in the column passes), see section 9.4 on page 293.

9.8 The z-transform (ZT)

In this section we will learn a technique to compute the FT by a cyclic convolution. In fact, the transform computed is the z-transform, a more general transform that in a special case is identical to the FT.

Definition

The discrete z-transform (ZT) of a length-n sequence a is a length-n sequence c defined by

$$c = \mathcal{Z}[a] \tag{9.33a}$$

$$c_k := \sum_{x=0}^{n-1} a_x \, z^{k\,x} \tag{9.33b}$$

The z-transform is a linear transformation. It is not an orthogonal transformation unless z is a root of unity. For $z = e^{\pm 2\,\pi\,i/n}$ the z-transform specializes to the the discrete Fourier transform. An important property is the convolution property:

$$\mathcal{Z}[a \circledast b] = \mathcal{Z}[a]\;\mathcal{Z}[b] \tag{9.34}$$

Convolution in original space corresponds to ordinary (element-wise) multiplication in z-space. This can be turned into an efficient convolution algorithm for the special case of the Fourier transform but not in general because no efficient algorithm for the inverse transform is known.

Computation via convolution

In the definition of the (discrete) z-transform we rewrite the product $x\,k$ as

$$x\,k = \frac{1}{2}\left(x^2 + k^2 - (k-x)^2\right) \tag{9.35}$$

Thereby for element c_k of the FT of a one can write

$$c_k = \sum_{x=0}^{n-1} a_x \, z^{x\,k} = z^{k^2/2} \left[\sum_{x=0}^{n-1} \left(a_x \, z^{x^2/2}\right) \, z^{-(k-x)^2/2}\right] \tag{9.36}$$

The expression in brackets is a cyclic convolution of the sequence $a_x \, z^{x^2/2}$ with the sequence $z^{-x^2/2}$.

This leads to the algorithm for the *chirp z-transform*:

1. Multiply the sequence a element-wise with $z^{x^2/2}$.

2. Convolve the resulting sequence with the sequence $z^{-x^2/2}$.
3. Multiply element-wise with the sequence $z^{k^2/2}$.

The above algorithm constitutes a fast algorithm for the ZT because fast convolution is possible via FFT (see [Nussb]).

Arbitrary length FFT by ZT

The length (n) of the input sequence a for the fast z-transform is not limited to highly composite values: For values of n where a FFT is not feasible pad the sequence with zeros up to a length L with $L >= 2\,n$ such that a length-L FFT can be computed (highly composite L, for example a power of two).

As the FT is the special case $z = e^{\pm 2\,\pi\,i/n}$ of the ZT the chirp-ZT algorithm constitutes an FFT algorithm for sequences of arbitrary length.

The transform takes a few times more than an direct FFT would take. The worst case (if only FFTs for n a power of 2 are available) is $n = 2^p + 1$: One must perform three FFTs of length $L = 2^{p+2} \approx 4\,n$ for the computation of the convolution. So the total work amounts to about 12 times the work a FFT of length $n = 2^p$ would cost. It is of course possible to lower this 'worst case factor' to 6 by using highly composite L slightly greater than $2\,n$.

For multiple computations of z-transforms of the same length one may want to store the Fourier transform of the sequence $z^{k^2/2}$ as it does not change. Thereby the worst case is reduced to a factor 4 with highly composite FFTs and 8 if FFTs are available for powers of two only.

A C++ implementation of the Fourier transform for sequences of arbitrary length is given in [FXT: file chirpzt/fftarblen.cc].

Fractional Fourier transform by ZT

The z-transform with $z = e^{\alpha\,2\,\pi\,i/n}$ is called the *fractional Fourier transform*. For $\alpha = \pm 1$ one again obtains the usual Fourier transform. The fractional Fourier transform can be used for the computation of the FT of sequences with only few nonzero elements and for the exact detection of frequencies that are not integer multiples of the lowest frequency of the DFT. A discussion of the fractional Fourier transform can be found in [11].

A C++ implementation of the fractional Fourier transform for sequences of arbitrary length is given in [FXT: file chirpzt/fftfract.cc].

9.9 Prime length FFTs

For the computation of FFTs for sequences whose length is prime we can exploit the existence of primitive roots. We will be able to express the transform of all but the first element as a cyclic convolution of two sequences whose length is reduced by one.

Let p be prime, then an element g exists so that the least positive exponent e so that $g^e \equiv 1 \bmod p$ is $e = p - 1$. The element g is called a *generator* (or *primitive root*) modulo p (see section 21.6 on page 530). Every nonzero element modulo p can uniquely be expressed as a power g^e where $0 \le e < p-1$. For example, a generator modulo $p = 11$ is $g = 2$, its powers are:

$$g^0 \equiv 1 \quad g^1 \equiv 2 \quad g^2 \equiv 4 \quad g^3 \equiv 8 \quad g^4 \equiv 5 \quad g^5 \equiv 10 \equiv -1 \quad g^6 \equiv 9 \quad g^7 \equiv 7 \quad g^8 \equiv 3 \quad g^9 \equiv 6 \quad g^{p-1} \equiv 1$$

Likewise, we can express any nonzero element as a negative power of g. Let $h = g^{-1}$, then with our example, $h \equiv 6$ and

$$h^0 \equiv 1 \quad h^1 \equiv 6 \quad h^2 \equiv 3 \quad h^3 \equiv 7 \quad h^4 \equiv 9 \quad h^5 \equiv 10 \equiv -1 \quad h^6 \equiv 5 \quad h^7 \equiv 8 \quad h^8 \equiv 4 \quad h^9 \equiv 2 \quad h^{p-1} \equiv 1$$

This is just the reversed sequence of values.

Let the c be the Fourier transform of length-p sequence a:

$$c_k \quad = \quad \sum_{x=0}^{p-1} a_x \, z^{\sigma \, x \, k} \tag{9.37}$$

where $z = \exp(2\,i\,\pi/p)$ and $\sigma = \pm 1$ is the sign of the transform. We split the computation of the Fourier transform in two parts, we compute the first element of the transform as

$$c_0 \quad = \quad \sum_{x=0}^{p-1} a_x \tag{9.38}$$

Now it remains to compute, for $1 \le k \le p-1$,

$$c_k \quad = \quad a_0 + \sum_{x=1}^{p-1} a_x \, z^{\sigma \, x \, k} \tag{9.39}$$

Note the lower index of the sum.

We write $k \equiv g^e$ and $x \equiv g^{-f}$ (modulo p), so

$$c_{(g^e)} - a_0 \quad = \quad \sum_{f=0}^{p-2} a_{(g^{-f})} \, z^{\sigma \, (g^{-f}) \, (g^e)} \quad = \quad \sum_{f=0}^{p-2} a_{(g^{-f})} \, z^{\sigma \, (g^{e-f})} \tag{9.40}$$

The sum is a cyclic convolution of the sequences $z^{(g^w)}$ and $a_{(g^{-w})}$ where $0 \le w \le p-2$. That is, if we permute the sequences a_1, a_2, …, a_{p-1} and z^1, z^2, …, z^{p-1} and compute their cyclic convolution then we obtain a permutation of the sequence c_1, c_2, …, c_{p-1}.

We implement the method (which is called *Rader's algorithm*) in pari/gp:

```
ft_rader(a, is=+1)=
\\ Fourier transform for prime lengths (Rader's algorithm)
{
    local(n, a0, f0, g, w);
    local(f, ixp, ixm, pa, pw, t);
    n = length(a);
    a0 = a[1];  f0 = sum(j=1, n, a[j]);
    g = znprimroot(n);  ixp = vector(n, j, component( g^(j-1), 2) );
    w = is*2*I*Pi/n;  pw = vector(n-1, j, exp(w*ixp[j]) );
    pa = vector(n-1);  for (j=1, n-1, pa[j]=a[1+ixp[1+n-j]] );
    t = cconv(pa, pw);  \\ cyclic convolution
    f = vector(n);  f[1] = f0;  for (k=1, n-1, f[1+k]=t[k]+a0);
    t = vector(n);  t[1] = f[1];  for (k=2, n, t[1+ixp[k-1]]=f[k]);
    return( t );
}
```

With a (slow) implementation of the cyclic convolution and DFT we can check whether the method works by comparing the results:

```
cconv(a, b, w=+1)=
\\ weighted cyclic convolution (by definition, n^2 operations)
\\ w==+1 ==> usual cyclic convolution
{
    local(n, f, s, k, k2);
    n = length(a);
    f = vector(n);
    for (tau=0, n-1,  \\ tau = k + k2
        s0 = 0;  k = 0;  k2 = tau;
        while (k<=tau, s0 += (a[k+1]*b[k2+1]);  k++; k2--);
        s1 = 0;  k2 = n-1; \\ k=tau+1
        while (k<n,    s1 += (a[k+1]*b[k2+1]);  k++; k2--);
        f[tau+1] = s0 + w * s1;
    );
```

```
    return( f );
}
dft(a, is=+1)=
\\ Fourier transform (by definition, n^2 operations)
{
    local(n, f, s, ph0, ph);
    n = length(a);
    f = vector(n);
    ph0 = is*2*Pi*I/n;
    for (k=0, n-1,
        ph = ph0 * k;
        f[k+1] = sum (x=0, n-1,  a[x+1] * exp(ph*x) );
    );
    return( f );
}
```

(It indeed works, we omit the trivial code.)

In order to turn the algorithm into a fast Fourier transform we need to compute the convolution via length-$(p-1)$ (fast) transforms. This is trivially possible when $p-1=2^q$, for example when $p=5$ or $p=17$. As $p-1$ is always divisible by two, we can split at least once. For $p=11$ we have $(p-1)/2=5$ so we can again use Rader's algorithm and length-4 transforms.

The method can be used to generate code for short (prime) length FFTs. It is advisable to create the permuted and transformed sequence of the powers of z. Thereby only two length-$(p-1)$ FFTs will be needed for a length-p transform.

Chapter 10

The Walsh transform and its relatives

In this chapter we describe several variant of the Walsh transform. The Walsh transform has the same complexity as the Fourier transform but does not involve any multiplications. In fact, one can obtain a Walsh transform routine by removing all multiplications (with the sines and cosines) in a given FFT routine.

Related transforms like the slant transform and the Reed-Muller transform are described. The dyadic convolution that can be computed efficiently by the Walsh transform is introduced.

The $2^k \times 2^k$-matrices corresponding to the Walsh transforms are so-called Hadamard matrices. Such matrices also exist for certain sizes $n \times n$ for n not a power of two. We give two construction schemes for Hadamard matrices that come from the theory of finite fields.

10.1 The Walsh transform: Walsh-Kronecker basis

How to make a Walsh transform out of your FFT:
'*Replace exp(something) by* 1*, done.*'

Removing all *exp*(something) from the radix-2, decimation in time Fourier transform we get

```
void slow_walsh_wak_dit2(double *f, ulong ldn)
// (this routine has a problem)
{
    ulong n = (1<<(ulong)ldn);
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1<<ldm);
        const ulong mh = (m>>1);
        for (ulong j=0; j<mh; ++j)
        {
            for (ulong r=0; r<n; r+=m)
            {
                const ulong t1 = r+j;
                const ulong t2 = t1+mh;
                double v = f[t2];
                double u = f[t1];
                f[t1] = u+v;
                f[t2] = u-v;
            }
        }
    }
}
```

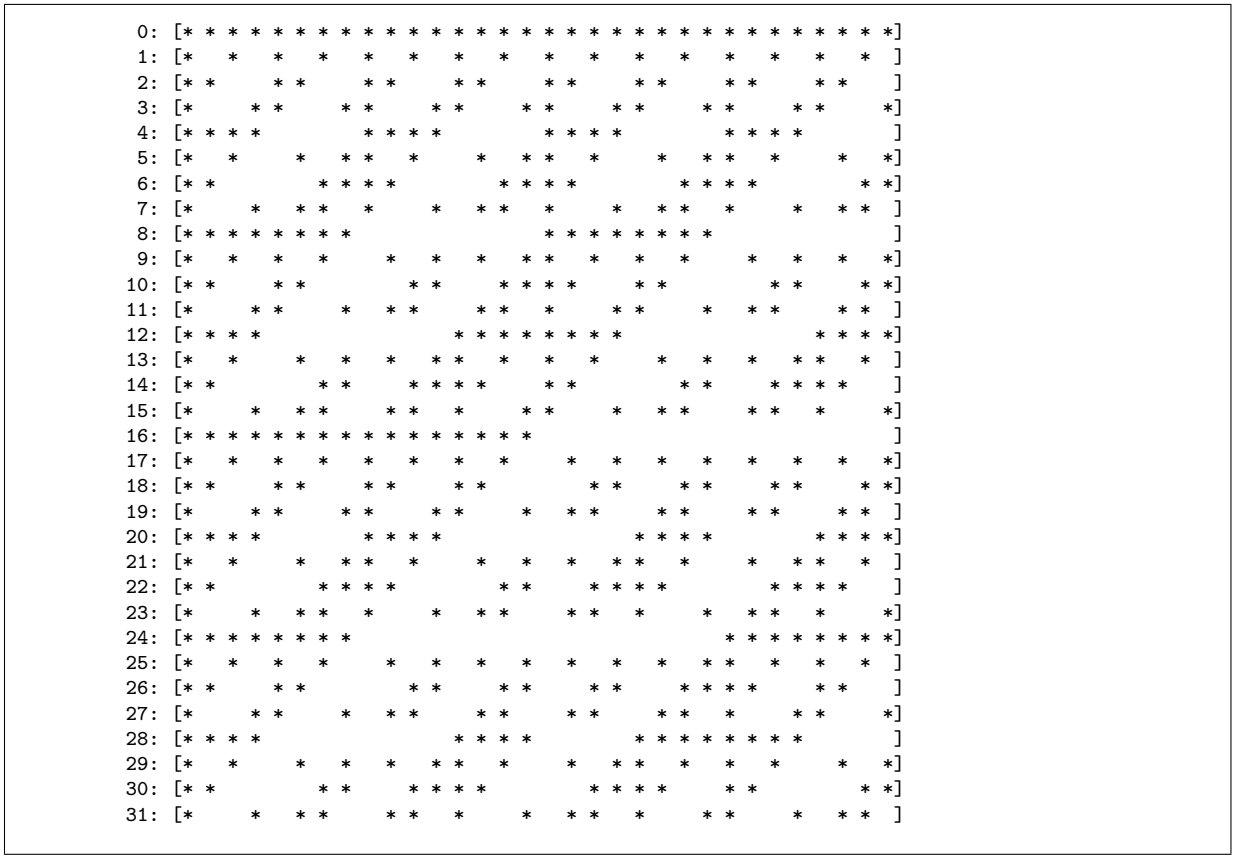


Figure 10.1: Basis functions for the Walsh transform (Walsh-Kronecker basis). Asterisks denote the value +1, blank entries denote −1.

The transform involves proportional $n \log_2(n)$ additions (and subtractions) and no multiplication at all. The transform, as given, is its own inverse up to a factor $1/n$. The Walsh transform of integer input is integral.

As the `slow` in the name shall suggest, the implementation has a problem as given. The memory access is highly non-local. Let's make a slight improvement: Here we just took the radix-2 DIT FFT code from page 258 and threw away all trigonometric computations (and multiplications). But the swapping of the inner loops, that we did for the FFT in order to save trigonometric computations is now of no advantage anymore. So we try this piece of code This is [FXT: `walsh_wak_dit2()` in walsh/walshwak2.h]:

```
template <typename Type>
void walsh_wak_dit2(Type *f, ulong ldn)
// Transform wrt. to Walsh-Kronecker basis (wak-functions).
// Radix-2 decimation in frequency (DIF) algorithm.
{
    ulong n = (1UL<<ldn);
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
```

```
        }
    }
}
```

Which performance impact can this innocent change in the code have? For $n = 2^{21}$ (and type double, 16 MByte of memory) it gives a speedup by a factor of about *eight*. For smaller lengths the ratio approaches one.

Code for the decimation in frequency algorithm is [FXT: `walsh_wak_dif2()` in walsh/walshwak2.h]:

```
template <typename Type>
void walsh_wak_dif2(Type *f, ulong ldn)
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

A function that computes the k-th base function of the transform is [FXT: `walsh_wak_basefunc()` in walsh/walshbasefunc.h]:

```
template <typename Type>
void walsh_wak_basefunc(Type *f, ulong n, ulong k)
{
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

The basis functions are shown in figure 10.1 on the facing page. Note that the lowest row is (the signed version of) the *Thue-Morse sequence*, see section 1.16.1 on page 34.

Multi-dimensional Walsh transform

If one applies the row-column algorithm (see section 8.10 on page 282) to compute a two-dimensional $n \times m$ Walsh transform then the result is exactly the same as with a 1-dimensional $n \cdot m$ transform. That is, algorithmically nothing needs to be done for multidimensional Walsh transforms: A k-dimensional $n_1 \times n_2 \times \ldots \times n_k$-transform is identical to a 1-dimensional $n_1 \cdot n_2 \cdot \ldots \cdot n_k$-transform. The length-$2^n$ Walsh transform is identical to a n-dimensional length-2 Fourier transform.

10.2 The Kronecker product

The length-2 Walsh transform is equivalent to the multiplication of a 2-component vector by the matrix

$$\mathbf{W}_2 \;=\; \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \tag{10.1}$$

The length-4 Walsh transform corresponds to

$$\mathbf{W}_4 = \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \end{bmatrix} \tag{10.2}$$

One might be tempted to write

$$\mathbf{W}_4 = \begin{bmatrix} +\mathbf{W}_2 & +\mathbf{W}_2 \\ +\mathbf{W}_2 & -\mathbf{W}_2 \end{bmatrix} \tag{10.3}$$

This idea can indeed be turned into a well-defined notation which turns out to be quite powerful when dealing with orthogonal transforms and their fast algorithms.

Let $\mathbf{A}$ be an $m \times n$ matrix

$$\mathbf{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \tag{10.4}$$

then the *Kronecker product* (or *tensor product*) with a matrix $\mathbf{B}$ is

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} a_{0,0}\mathbf{B} & a_{0,1}\mathbf{B} & \cdots & a_{0,n-1}\mathbf{B} \\ a_{1,0}\mathbf{B} & a_{1,1}\mathbf{B} & \cdots & a_{1,n-1}\mathbf{B} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0}\mathbf{B} & a_{m-1,1}\mathbf{B} & \cdots & a_{m-1,n-1}\mathbf{B} \end{bmatrix} \tag{10.5}$$

There is no restriction on the dimensions of $\mathbf{B}$. If $\mathbf{B}$ is a $r \times s$ matrix then the dimensions of the given Kronecker product is $mr \times ns$ and $c_{k+ir,l+js} = a_{i,j}b_{k,l}$.

For a scalar factor α the relations

$$(\alpha\mathbf{A}) \otimes \mathbf{B} = \alpha(\mathbf{A} \otimes \mathbf{B}) \tag{10.6a}$$
$$\mathbf{A} \otimes (\alpha\mathbf{B}) = \alpha(\mathbf{A} \otimes \mathbf{B}) \tag{10.6b}$$

should be immediate.

The Kronecker product is not commutative, that is, $\mathbf{A} \otimes \mathbf{B} \neq \mathbf{B} \otimes \mathbf{A}$ in general. The following relations are the same as for the ordinary matrix product. Bilinearity (the matrices left and right from a plus sign must be of the same dimensions):

$$(\mathbf{A} + \mathbf{B}) \otimes \mathbf{C} = \mathbf{A} \otimes \mathbf{C} + \mathbf{B} \otimes \mathbf{C} \tag{10.7a}$$
$$\mathbf{A} \otimes (\mathbf{B} + \mathbf{C}) = \mathbf{A} \otimes \mathbf{B} + \mathbf{A} \otimes \mathbf{C} \tag{10.7b}$$

Associativity:

$$\mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}) = (\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} \tag{10.8}$$

The matrix product (indicated by a dot) of Kronecker products can be rewritten as

$$(\mathbf{A} \otimes \mathbf{B}) \cdot (\mathbf{C} \otimes \mathbf{D}) = (\mathbf{A} \cdot \mathbf{C}) \otimes (\mathbf{B} \cdot \mathbf{D}) \tag{10.9a}$$
$$(\mathbf{L}_1 \otimes \mathbf{R}_1) \cdot (\mathbf{L}_2 \otimes \mathbf{R}_2) \cdot \ldots \cdot (\mathbf{L}_n \otimes \mathbf{R}_n) = (\mathbf{L}_1 \cdot \mathbf{L}_2 \cdot \ldots \cdot \mathbf{L}_n) \otimes (\mathbf{R}_1 \cdot \mathbf{R}_2 \cdot \ldots \cdot \mathbf{R}_n) \tag{10.9b}$$

The Kronecker product of matrix products can be rewritten as

$$(\mathbf{A} \cdot \mathbf{B}) \otimes (\mathbf{C} \cdot \mathbf{D}) = (\mathbf{A} \otimes \mathbf{C}) \cdot (\mathbf{B} \otimes \mathbf{D}) \tag{10.10a}$$
$$(\mathbf{L}_1 \cdot \mathbf{R}_1) \otimes (\mathbf{L}_2 \cdot \mathbf{R}_2) \otimes \ldots \otimes (\mathbf{L}_n \cdot \mathbf{R}_n) = (\mathbf{L}_1 \otimes \mathbf{L}_2 \otimes \ldots \otimes \mathbf{L}_n) \cdot (\mathbf{R}_1 \otimes \mathbf{R}_2 \otimes \ldots \otimes \mathbf{R}_n) \tag{10.10b}$$

Here the matrices left and right from a dot must be compatible for ordinary matrix multiplication.

One has

$$(\mathbf{A} \otimes \mathbf{B})^T = \mathbf{A}^T \otimes \mathbf{B}^T \tag{10.11a}$$
$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1} \tag{10.11b}$$

and, if $\mathbf{A}$ and $\mathbf{B}$ are $m \times n$ and $r \times s$ matrices, respectively

$$\mathbf{A} \otimes \mathbf{B} = (\mathbf{I}_m \otimes \mathbf{B}\,) \cdot (\mathbf{A} \otimes \mathbf{I}_s) \tag{10.12a}$$
$$= (\mathbf{A} \otimes \mathbf{I}_r) \cdot (\mathbf{I}_n \otimes \mathbf{B}\,) \tag{10.12b}$$

where $\mathbf{I}_n$ is the $n \times n$ identity matrix.

Back to the Walsh transform, we have $\mathbf{W}_1 = [1]$ and for $n = 2^k$, $n > 1$:

$$\mathbf{W}_n = \begin{bmatrix} +\mathbf{W}_{n/2} & +\mathbf{W}_{n/2} \\ +\mathbf{W}_{n/2} & -\mathbf{W}_{n/2} \end{bmatrix} = \mathbf{W}_2 \otimes \mathbf{W}_{n/2} \tag{10.13}$$

In order to see that this relation is the statement of a fast algorithm split the (to be transformed) vector x into halves

$$x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \tag{10.14}$$

and write out the matrix-vector product

$$\mathbf{W}_n\, x = \begin{bmatrix} \mathbf{W}_{n/2}\, x_0 + \mathbf{W}_{n/2}\, x_1 \\ \mathbf{W}_{n/2}\, x_0 - \mathbf{W}_{n/2}\, x_1 \end{bmatrix} = \begin{bmatrix} \mathbf{W}_{n/2}\, (x_0 + x_1) \\ \mathbf{W}_{n/2}\, (x_0 - x_1) \end{bmatrix} \tag{10.15}$$

That is, a length-n transform can be computed by two length-$n/2$ transforms of the sum and difference of the first and second half of x.

Now when $\mathbf{A} = \mathbf{B}$ in relation 10.11b we have $(\mathbf{A} \otimes \mathbf{A})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{A}^{-1}$, $(\mathbf{A} \otimes \mathbf{A} \otimes \mathbf{A})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{A}^{-1} \otimes \mathbf{A}^{-1}$ and so on. We define a notation equivalent to the product sign:

$$\left(\bigotimes_{k=1}^{n} \mathbf{A} \right)^{-1} = \bigotimes_{k=1}^{n} \mathbf{A}^{-1} \tag{10.16}$$

where the empty product equals $[1] = \mathbf{W}_1$.

For the Walsh transform:

$$\mathbf{W}_n = \bigotimes_{k=1}^{\log_2(n)} \mathbf{W}_2 \tag{10.17}$$

and

$$\mathbf{W}_n^{-1} = \bigotimes_{k=1}^{\log_2(n)} \mathbf{W}_2^{-1} \tag{10.18}$$

The latter relation isn't that exciting as $\mathbf{W}_2^{-1} = \mathbf{W}_2$ for the Walsh transform. However, it also holds when the inverse transform is different from the forward transform. Thereby, given a fast algorithm for some transform in form of a Kronecker product, the fast algorithm for the backward transform is immediate.

Computation of the The Kronecker product of two matrices is implemented in [FXT: `class matrix` in matrix/matrix.h]:

```
bool kronecker(const matrix<Type> A, const matrix<Type> B)
{
    ulong nra = A.nr_,  nca = A.nc_;
    ulong nrb = B.nr_,  ncb = B.nc_;
    if ( nra * nrb != nr_ )  return false;
    if ( nca * ncb != nc_ )  return false;
    for (ulong ra=0; ra<nra; ++ra)
    {
        ulong ro = ra * nrb;
        for (ulong ca=0; ca<nca; ++ca)
        {
            ulong co = ca * ncb;
            Type ea = A.get(ra,ca);
            for (ulong rb=0; rb<nrb; ++rb)
            {
                ulong r = ro + rb;
                for (ulong cb=0; cb<ncb; ++cb)
                {
                    Type eb = B.get(rb,cb);
                    ulong c = co + cb;
                    Type p = ea;  p *= eb;
                    set(r, c, p);
                }
            }
        }
    }
    return true;
}
```

The *direct sum* of two matrices is defined as

$$\mathbf{A} \oplus \mathbf{B} \quad := \quad \begin{bmatrix} \mathbf{A} & 0 \\ 0 & \mathbf{B} \end{bmatrix} \tag{10.19}$$

As an analog to the sum sign we have

$$\bigoplus_{k=1}^{n} \mathbf{A} \quad := \quad \mathbf{I}_n \otimes \mathbf{A} \tag{10.20}$$

where $\mathbf{I}_n$ is the $n \times n$ identity matrix. The matrix $\mathbf{I}_n \otimes \mathbf{A}$ consists of n copies of $\mathbf{A}$ that lie on the diagonal.

10.3 A variant of the Walsh transform *

All operations necessary for the Walsh transform are cheap: loads, stores, additions and subtractions. The memory access pattern is a major concern with direct mapped cache, as we have verified comparing the first two implementations in this chapter. Even the one found to be superior due to its more localized access is guaranteed to have a performance problem as soon as the array is long enough: all accesses are separated by a power-of-two distance and cache misses will occur beyond a certain limit. Rather bizarre attempts like inserting 'pad data' have been reported in order to mitigate the problem. The Gray code permutation described in section 2.8 on page 91 suggests an interesting solution where the sub-arrays are always accessed in mutually reversed order [FXT: file walsh/walshgray.h]:

```
template <typename Type>
void walsh_gray(Type *f, ulong ldn)
// Gray variant of the Walsh transform.
// Radix-2 decimation in frequency (DIF) algorithm
{
    const ulong n = (1UL<<ldn);

    for (ulong ldm=ldn; ldm>0; --ldm) // dif
    {
        const ulong m = (1UL<<ldm);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r + m - 1;
            for (  ;  t1<t2;  ++t1,--t2)
            {
```

```
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

The transform is not self-inverse, however, the inverse transform can be implemented easily be reversing the steps:

```
template <typename Type>
void inverse_walsh_gray(Type *f, ulong ldn)
// Inverse of walsh_gray().
// Radix-2 decimation in time (DIT) algorithm.
{
    const ulong n = (1UL<<ldn);

    for (ulong ldm=1; ldm<=ldn; ++ldm) // dit
    {
        const ulong m = (1UL<<ldm);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r + m - 1;
            for (  ;  t1<t2;  ++t1,--t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

Using Q for the `grs_negate()` routine (described below), W_k for `walsh_wak()`, W_g for `walsh_gray()` and G for `gray_permute()` then

$$W_k \;=\; Q\,W_g\,G^{-1} \;=\; G\,W_g^{-1}\,Q \tag{10.21}$$

That is, the following two sequences of statements

```
    inverse_gray_permute(f, n);  walsh_gray(f, ldn);  grs_negate(f, n);

    grs_negate(f, n);  walsh_gray(f, ldn);  gray_permute(f, n);
```

are both equivalent to the call `walsh_wak(f, ldn)`. The third line is a necessary correction for certain elements that have the wrong sign if uncorrected [FXT: `grs_negate()` in aux1/grsnegate.h]:

```
template <typename Type>
void grs_negate(Type *f, ulong n)
// Negate elements at indices where the Golay-Rudin-Shapiro is negative.
{
    for (ulong k=0; k<n; ++k)
    {
        if ( grs_negative_q(k) )  f[k] = -f[k];
    }
}
```

The function `grs_negative_q()` is described in section 1.16.4 on page 37.

It turns out that the Gray-variant only wins on machines where the memory clock speed is significantly lower than the CPU. While a call to `walsh_gray()` alone is never slower than the `walsh_wak()` routine the additional steps often cause too much overhead.

10.4 Higher radix Walsh transforms

A generator for short-length Walsh (wak) transforms is given as [FXT: demo fft/gen-walsh-demo.cc]. It allows to create code for DIF and DIT transforms. For example, the code for the 4-point DIF transforms

is

```
template <typename Type>
inline void
short_walsh_wak_dif_4(Type *f)
{
 Type t0, t1, t2, t3;
 t0 = f[0];
 t1 = f[1];
 t2 = f[2];
 t3 = f[3];
 sumdiff( t0, t2 );
 sumdiff( t1, t3 );
 sumdiff( t0, t1 );
 sumdiff( t2, t3 );
 f[0] = t0;
 f[1] = t1;
 f[2] = t2;
 f[3] = t3;
}
```

To keep the code readable, we use the `sumdiff()` function from [FXT: file aux0/sumdiff.h]:

```
template <typename Type>
static inline void sumdiff(Type &a, Type &b)
// {a, b}  <--| {a+b, a-b}
{ Type t=a-b; a+=b; b=t; }
```

We further need a variant that transforms elements which are not contiguous but lie apart by a distance s:

```
template <typename Type>
inline void
short_walsh_wak_dif_4(Type *f, ulong s)
{
 Type t0, t1, t2, t3;
 {
 ulong x = 0;
 t0 = f[x];  x += s;
 t1 = f[x];  x += s;
 t2 = f[x];  x += s;
 t3 = f[x];
 }
 sumdiff( t0, t2 );
 sumdiff( t1, t3 );
 sumdiff( t0, t1 );
 sumdiff( t2, t3 );
 {
 ulong x = 0;
 f[x] = t0;  x += s;
 f[x] = t1;  x += s;
 f[x] = t2;  x += s;
 f[x] = t3;
 }
}
```

The short DIF transforms are given in [FXT: file walsh/shortwalshwakdif.h], DIT variants can be found in [FXT: file walsh/shortwalshwakdit.h].

A radix-4 DIF transform using these ingredients is [FXT: file walsh/walshwak4.h]:

```
template <typename Type>
void walsh_wak_dif4(Type *f, ulong ldn)
// Transform wrt. to Walsh-Kronecker basis (wak-functions).
// Radix-4 decimation in frequency (DIF) algorithm.
// Self-inverse.
{
    const ulong n = (1UL<<ldn);

    if ( n<=2 )
    {
        if ( n==2 )  short_walsh_wak_dif_2(f);
        return;
    }

    for (ulong ldm=ldn; ldm>3; ldm-=2)
    {
        ulong m = (1UL<<ldm);
        ulong m4 = (m>>2);
```

```
        for (ulong r=0;  r<n;  r+=m)
        {
            for (ulong j=0;  j<m4;  j++)  short_walsh_wak_dif_4(f+j+r, m4);
        }
    }

    if ( ldn & 1 )  // n is not a power of 4, need a radix-8 step
    {
        for (ulong i0=0; i0<n; i0+=8)  short_walsh_wak_dif_8(f+i0);
    }
    else
    {
        for (ulong i0=0; i0<n; i0+=4)  short_walsh_wak_dif_4(f+i0);
    }
}
```

With the implementation radix-8 DIF transform some care must be taken to choose the correct final step size [FXT: file walsh/walshwak8.h]:

```
template <typename Type>
void walsh_wak_dif8(Type *f, ulong ldn)
// Transform wrt. to Walsh-Kronecker basis (wak-functions).
// Radix-8 decimation in frequency (DIF) algorithm.
// Self-inverse.
{
    const ulong n = (1UL<<ldn);

    if ( n<=4 )
    {
        switch (n )
        {
        case 4:  short_walsh_wak_dif_4(f);  break;
        case 2:  short_walsh_wak_dif_2(f);  break;
        }
        return;
    }

    const ulong xx = 4;
    ulong ldm;
    for (ldm=ldn; ldm>xx; ldm-=3)
    {
        ulong m = (1UL<<ldm);
        ulong m8 = (m>>3);
        for (ulong r=0;  r<n;  r+=m)
        {
            for (ulong j=0;  j<m8;  j++)  short_walsh_wak_dif_8(f+j+r, m8);
        }
    }

    switch ( ldm )
    {
    case 4:
        for (ulong i0=0; i0<n; i0+=16)  short_walsh_wak_dif_16(f+i0);
        break;
    case 3:
        for (ulong i0=0; i0<n; i0+=8)  short_walsh_wak_dif_8(f+i0);
        break;
    case 2:
        for (ulong i0=0; i0<n; i0+=4)  short_walsh_wak_dif_4(f+i0);
        break;
    }
}
```

Performance

For the performance comparison we include a matrix variant of the Walsh transform [FXT: file walsh/walshwakmatrix.h]:

```
template <typename Type>
void walsh_wak_matrix(Type *f, ulong ldn)
{
    ulong ldc = (ldn>>1);
    ulong ldr = ldn-ldc;  // ldr>=ldc
    ulong nc = (1UL<<ldc);
    ulong nr = (1UL<<ldr);  // nrow >= ncol

    for (ulong r=0; r<nr; ++r)  walsh_wak_dif4(f+r*nc, ldc);
```

```
    transpose2(f, nr, nc);
    for (ulong c=0; c<nc; ++c)  walsh_wak_dif4(f+c*nr, ldr);
    transpose2(f, nc, nr);
}
```

The transposition routine is given in [FXT: file aux2/transpose2.h]. We only use even powers of two so the transposition is that of a square matrix.

As for dyadic convolutions we do not need the data in a particular order so we also include a version of the matrix algorithm that omits the final transposition:

```
template <typename Type>
void walsh_wak_matrix_1(Type *f, ulong ldn, int is)
{
    ulong ldc = (ldn>>1);
    ulong ldr = ldn-ldc;  // ldr>=ldc
    if ( is<0 )  swap2(ldr, ldc);  // inverse
    ulong nc = (1UL<<ldc);
    ulong nr = (1UL<<ldr);  // nrow >= ncol

    for (ulong r=0; r<nr; ++r)  walsh_wak_dif4(f+r*nc, ldc);
    transpose2(f, nr, nc);
    for (ulong c=0; c<nc; ++c)  walsh_wak_dif4(f+c*nr, ldr);
}
```

The following calls give (up to normalization) the mutually inverse transforms:

```
    walsh_wak_matrix_1(f, ldn, +1);
    walsh_wak_matrix_1(f, ldn, -1);
```

We omit the range of 'tiny' lengths $n < 128$ were unrolled routines and the radix-4 algorithm consistently win. Figure 10.2 on the next page shows a comparison of the routines given so far. There are clearly two regions to distinguish: Firstly, the region were the transforms fit into the first-level data cache (which is 64 kilobyte, corresponding to '`ldn`'$= 13$). Secondly, the region where '`ldn`'> 13 and the performance becomes more and more memory bound.

In the first region the radix-4 routine is the fastest. The radix-8 routine comes close but, somewhat surprisingly, never wins.

In the second region the matrix version is the best. However, for very large sizes its performance could be better. Note that with odd '`ldn`' (not shown) its performance drops significantly due to the more expensive transposition operation. The transposition is clearly the bottleneck. One can use (machine specific) optimizations for the transposition to further improve the performance.

In the next section we give an algorithm that avoids the transposition completely and consistently outperforms the matrix algorithm.

10.5 Localized Walsh transforms

A decimation in time (DIT) algorithm combines the two halves of the array, then the halves of the two halves, the the halves of each quarter, and so on. With each step the whole array is accessed which leads to the performance drop as soon as the array doesn't fit into the cache.

One can reorganize the algorithm as follows: combine the two halves of the array and postpone further processing of the upper half, then combine the halves of the lowers half and again postpone processing of its upper half. Repeat until size two is reached. Then use the algorithm at the postponed parts, starting with the smallest (last postponed). The scheme can be sketched for size 16, as follows:

```
hhhhhhhhhhhhhhhh
hhhhhhhh44444444
hhhh333344444444
hh22333344444444
```

```
 8 == ldn;   MemSize ==     2 kB  ==  256 doubles;   rep == 976563
              walsh_wak_dif2(f,ldn);    dt=    2.49551    MB/s= 6114   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    1.56806    MB/s= 9731   rel=   0.628352 *
              walsh_wak_dif8(f,ldn);    dt=    1.57419    MB/s= 9693   rel=    0.63081
            walsh_wak_matrix(f,ldn);    dt=    2.28047    MB/s= 6691   rel=    0.91383
      walsh_wak_matrix_1(f,ldn,+1);     dt=    1.94357    MB/s= 7851   rel=   0.778827
                 walsh_gray(f,ldn);     dt=    2.39791    MB/s= 6363   rel=   0.960888

10 == ldn;   MemSize ==     8 kB  ==  1024 doubles;   rep == 195313
              walsh_wak_dif2(f,ldn);    dt=    2.26683    MB/s= 6731   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    1.47338    MB/s=10356   rel=   0.649977 *
              walsh_wak_dif8(f,ldn);    dt=    1.65262    MB/s= 9233   rel=   0.729044
            walsh_wak_matrix(f,ldn);    dt=    1.91859    MB/s= 7953   rel=   0.846378
      walsh_wak_matrix_1(f,ldn,+1);     dt=    1.69215    MB/s= 9017   rel=   0.746485
                 walsh_gray(f,ldn);     dt=    2.18454    MB/s= 6985   rel=     0.9637

12 == ldn;   MemSize ==    32 kB  ==  4096 doubles;   rep == 20345
              walsh_wak_dif2(f,ldn);    dt=     1.0884    MB/s= 7010   rel=          1
              walsh_wak_dif4(f,ldn);    dt=   0.723136    MB/s=10550   rel=   0.664403 *
              walsh_wak_dif8(f,ldn);    dt=   0.790313    MB/s= 9654   rel=   0.726124
            walsh_wak_matrix(f,ldn);    dt=    1.01233    MB/s= 7536   rel=   0.930112
      walsh_wak_matrix_1(f,ldn,+1);     dt=   0.926387    MB/s= 8236   rel=   0.851146
                 walsh_gray(f,ldn);     dt=    1.05364    MB/s= 7241   rel=   0.968062

14 == ldn;   MemSize ==   128 kB  ==  16384 doubles;   rep == 2180
              walsh_wak_dif2(f,ldn);    dt=    1.17042    MB/s= 3260   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    1.14861    MB/s= 3321   rel=   0.981368
              walsh_wak_dif8(f,ldn);    dt=    1.08501    MB/s= 3516   rel=   0.927026
            walsh_wak_matrix(f,ldn);    dt=   0.669182    MB/s= 5701   rel=   0.571747
      walsh_wak_matrix_1(f,ldn,+1);     dt=   0.552063    MB/s= 6910   rel=   0.471681 *
                 walsh_gray(f,ldn);     dt=    1.00794    MB/s= 3785   rel=    0.86118
16 == ldn;   MemSize ==   512 kB  ==  65536 doubles;   rep == 477
              walsh_wak_dif2(f,ldn);    dt=    1.40004    MB/s= 2726   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    1.70347    MB/s= 2240   rel=    1.21673
              walsh_wak_dif8(f,ldn);    dt=    1.12997    MB/s= 3377   rel=   0.807095
            walsh_wak_matrix(f,ldn);    dt=   0.801902    MB/s= 4759   rel=   0.572769
      walsh_wak_matrix_1(f,ldn,+1);     dt=   0.628073    MB/s= 6076   rel=   0.448609 *
                 walsh_gray(f,ldn);     dt=    1.18693    MB/s= 3215   rel=   0.847779
18 == ldn;   MemSize ==     2 MB  ==  256 K doubles;   rep == 106
              walsh_wak_dif2(f,ldn);    dt=    2.61599    MB/s= 1459   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    2.55153    MB/s= 1496   rel=   0.975359
              walsh_wak_dif8(f,ldn);    dt=     1.9791    MB/s= 1928   rel=   0.756538
            walsh_wak_matrix(f,ldn);    dt=    1.77306    MB/s= 2152   rel=   0.677776
      walsh_wak_matrix_1(f,ldn,+1);     dt=    1.14735    MB/s= 3326   rel=   0.438591 *
                 walsh_gray(f,ldn);     dt=    2.51539    MB/s= 1517   rel=   0.961545
20 == ldn;   MemSize ==     8 MB  ==  1024 K doubles;   rep == 24
              walsh_wak_dif2(f,ldn);    dt=    2.64158    MB/s= 1454   rel=          1
              walsh_wak_dif4(f,ldn);    dt=     2.8532    MB/s= 1346   rel=    1.08011
              walsh_wak_dif8(f,ldn);    dt=    2.34867    MB/s= 1635   rel=   0.889113
            walsh_wak_matrix(f,ldn);    dt=    1.88431    MB/s= 2038   rel=   0.713327
      walsh_wak_matrix_1(f,ldn,+1);     dt=    1.21084    MB/s= 3171   rel=   0.458376 *
                 walsh_gray(f,ldn);     dt=    2.58747    MB/s= 1484   rel=   0.979514
22 == ldn;   MemSize ==    32 MB  ==  4096 K doubles;   rep == 5
              walsh_wak_dif2(f,ldn);    dt=    2.43537    MB/s= 1445   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    2.82337    MB/s= 1247   rel=    1.15932
              walsh_wak_dif8(f,ldn);    dt=    2.07422    MB/s= 1697   rel=   0.851708
            walsh_wak_matrix(f,ldn);    dt=    1.99251    MB/s= 1767   rel=   0.818155
      walsh_wak_matrix_1(f,ldn,+1);     dt=    1.22719    MB/s= 2868   rel=   0.503901 *
                 walsh_gray(f,ldn);     dt=    2.38611    MB/s= 1475   rel=   0.979776
24 == ldn;   MemSize ==   128 MB  ==  16384 K doubles;   rep == 1
              walsh_wak_dif2(f,ldn);    dt=    2.10939    MB/s= 1456   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    2.61517    MB/s= 1175   rel=    1.23977
              walsh_wak_dif8(f,ldn);    dt=    2.11508    MB/s= 1452   rel=     1.0027
            walsh_wak_matrix(f,ldn);    dt=    2.16597    MB/s= 1418   rel=    1.02683
      walsh_wak_matrix_1(f,ldn,+1);     dt=    1.28349    MB/s= 2393   rel=   0.608466 *
                 walsh_gray(f,ldn);     dt=    2.04744    MB/s= 1500   rel=   0.970635
```

Figure 10.2: Relative speed of different implementations of the Walsh (wak) transform. The transforms were run '`rep`' times for each measurement, the quantity '`dt`' gives the elapsed time for `rep` transforms of the given type. The quantity '`MB/s`' gives the memory transfer rate as if a radix-2 algorithm was used, it equals '`Memsize`' times '`ldn`' divided by the time elapsed for a single transform. The '`rel`' gives the performance relative to the radix-2 version, smaller values mean better performance.

The letters 'h' denote places processed and the blocks of twos, threes and fours denote postponed blocks (see also section 11.6 on page 353).

The algorithm described is most easily implemented via recursion:

```
template <typename Type>
void walsh_wak_loc_dit2(Type *f, ulong ldn)
{
    if ( ldn<1 )  return;

    // Recursion:
    for (ulong ldm=1; ldm<ldn; ++ldm)  walsh_wak_loc_dit2(f+(1UL<<ldm), ldm);

    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
    }
}
```

Optimizations are obtained by avoiding recursions for small sizes. We use a radix-4 algorithm as soon as the transform size is smaller or equal to the cache size and we avoid recursion for tiny transforms:

```
template <typename Type>
void walsh_wak_loc_dit2(Type *f, ulong ldn)
{
    if ( ldn<=13 )
    {
        walsh_wak_dif4(f,ldn);  // note: DIF version, result is the same
        return;
    }

    // Recursion:
    short_walsh_wak_dit_2(f+2);  // ldm==1
    short_walsh_wak_dit_4(f+4);  // ldm==2
    short_walsh_wak_dit_8(f+8);  // ldm==3
    short_walsh_wak_dit_16(f+16);  // ldm==4
    for (ulong ldm=5; ldm<ldn; ++ldm)  walsh_wak_loc_dit2(f+(1UL<<ldm), ldm);

    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
    }
}
```

The routine is given in [FXT: file walsh/walshwakloc2.h]. A decimation in frequency (DIF) version is obtained by reversion of the steps:

```
template <typename Type>
void walsh_wak_loc_dif2(Type *f, ulong ldn)
{
    if ( ldn<=13 )
    {
        walsh_wak_dif4(f,ldn);
        return;
    }

    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)
        {
            Type u = f[t1];
            Type v = f[t2];
            f[t1] = u + v;
            f[t2] = u - v;
        }
    }

    // Recursion:
    short_walsh_wak_dif_2(f+2);  // ldm==1
    short_walsh_wak_dif_4(f+4);  // ldm==2
    short_walsh_wak_dif_8(f+8);  // ldm==3
    short_walsh_wak_dif_16(f+16);  // ldm==4
```

```
    for (ulong ldm=5; ldm<ldn; ++ldm)  walsh_wak_loc_dif2(f+(1UL<<ldm), ldm);
}
```

The double loop in the algorithm is a reversed Haar transform, see chapter 11 on page 343. The double loop in the DIF algorithm is a transposed reversed Haar transform.

```
    14 == ldn;   MemSize ==   128 kB  ==  16384 doubles;   rep == 2180
                walsh_wak_matrix(f,ldn);    dt=   0.672327    MB/s= 5674   rel=          1
           walsh_wak_matrix_1(f,ldn,+1);    dt=   0.555851    MB/s= 6863   rel=   0.826756
              walsh_wak_loc_dit2(f,ldn);    dt=   0.498558    MB/s= 7652   rel=   0.741541 *
              walsh_wak_loc_dif2(f,ldn);    dt=   0.533746    MB/s= 7148   rel=   0.793878
    16 == ldn;   MemSize ==   512 kB  ==  65536 doubles;   rep == 477
                walsh_wak_matrix(f,ldn);    dt=   0.919579    MB/s= 4150   rel=          1
           walsh_wak_matrix_1(f,ldn,+1);    dt=   0.692488    MB/s= 5511   rel=   0.753049
              walsh_wak_loc_dit2(f,ldn);    dt=   0.653256    MB/s= 5842   rel=   0.710386 *
              walsh_wak_loc_dif2(f,ldn);    dt=   0.670104    MB/s= 5695   rel=   0.728707
    18 == ldn;   MemSize ==     2 MB  ==  256 K doubles;   rep == 106
                walsh_wak_matrix(f,ldn);    dt=     2.2111    MB/s= 1726   rel=          1
           walsh_wak_matrix_1(f,ldn,+1);    dt=    1.36827    MB/s= 2789   rel=   0.618819
              walsh_wak_loc_dit2(f,ldn);    dt=   0.938006    MB/s= 4068   rel=   0.424225
              walsh_wak_loc_dif2(f,ldn);    dt=   0.927804    MB/s= 4113   rel=   0.419611 *
    20 == ldn;   MemSize ==     8 MB  ==  1024 K doubles;   rep == 24
                walsh_wak_matrix(f,ldn);    dt=    2.31178    MB/s= 1661   rel=          1
           walsh_wak_matrix_1(f,ldn,+1);    dt=    1.42614    MB/s= 2693   rel=   0.616901
              walsh_wak_loc_dit2(f,ldn);    dt=    1.11847    MB/s= 3433   rel=   0.483811
              walsh_wak_loc_dif2(f,ldn);    dt=    1.11142    MB/s= 3455   rel=   0.480765 *
    22 == ldn;   MemSize ==    32 MB  ==  4096 K doubles;   rep == 5
                walsh_wak_matrix(f,ldn);    dt=    2.00573    MB/s= 1755   rel=          1
           walsh_wak_matrix_1(f,ldn,+1);    dt=    1.23695    MB/s= 2846   rel=   0.616707
              walsh_wak_loc_dit2(f,ldn);    dt=    1.16461    MB/s= 3022   rel=   0.580644
              walsh_wak_loc_dif2(f,ldn);    dt=    1.16164    MB/s= 3030   rel=   0.579162 *
    24 == ldn;   MemSize ==   128 MB  ==  16384 K doubles;   rep == 1
                walsh_wak_matrix(f,ldn);    dt=    2.16536    MB/s= 1419   rel=          1
           walsh_wak_matrix_1(f,ldn,+1);    dt=    1.28455    MB/s= 2392   rel=   0.593226
              walsh_wak_loc_dit2(f,ldn);    dt=    1.10769    MB/s= 2773   rel=   0.511552
              walsh_wak_loc_dif2(f,ldn);    dt=    1.10601    MB/s= 2778   rel=   0.510775 *
```

Figure 10.3: Speed comparison between localized and matrix algorithms for the Walsh transform.

The used strategy leads to a very favorable memory access pattern that results in excellent performance for large transforms. Figure 10.3 shows a comparison between the localized transforms and the matrix algorithm. Small sizes are omitted because the localized algorithm has the very same speed as the radix-4 algorithm it falls back to. The localized algorithms are the clear winners, even against the matrix algorithm with only one transposition. For very large transforms the decimation in frequency version is very slightly faster. This is due to the fact that it starts with smaller chunks of data so more of the data is in cache when the larger sub-arrays are accessed.

The localized algorithm can easily be implemented for transforms where a radix-2 step is known. Section 12.9 on page 371 gives the fast Hartley transform variant of the localized algorithm.

One can develop similar routines with higher radix. However, a radix-4 version was found to be slower than the given routines. A certain speedup can be achieved by unrolling and prefetching. We use the C-type `double` whose size is 8 bytes. Substitute the double loop in the DIF version (that is, the Haar transform) by

```
// machine specific prefetch instruction:
#define PREF(p,o)  asm volatile ("prefetchw " #o "(%0) " : : "r" (p) )
    ulong ldm;
    for (ldm=ldn; ldm>=6; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        PREF(f, 0);  PREF(f+mh, 0);
        PREF(f, 64);  PREF(f+mh, 64);
        PREF(f, 128);  PREF(f+mh, 128);
        PREF(f, 192);  PREF(f+mh, 192);
```

```
        for (ulong t1=0, t2=mh;  t1<mh;  t1+=8, t2+=8)
        {
            double *p1 = f + t1,  *p2 = f + t2;
            PREF(p1, 256);  PREF(p2, 256);

            double u0 = f[t1+0], v0 = f[t2+0];
            double u1 = f[t1+1], v1 = f[t2+1];
            double u2 = f[t1+2], v2 = f[t2+2];
            double u3 = f[t1+3], v3 = f[t2+3];
            sumdiff(u0, v0); f[t1+0] = u0; f[t2+0] = v0;
            sumdiff(u1, v1); f[t1+1] = u1; f[t2+1] = v1;
            sumdiff(u2, v2); f[t1+2] = u2; f[t2+2] = v2;
            sumdiff(u3, v3); f[t1+3] = u3; f[t2+3] = v3;

            double u4 = f[t1+4], v4 = f[t2+4];
            double u5 = f[t1+5], v5 = f[t2+5];
            double u6 = f[t1+6], v6 = f[t2+6];
            double u7 = f[t1+7], v7 = f[t2+7];
            sumdiff(u4, v4); f[t1+4] = u4; f[t2+4] = v4;
            sumdiff(u5, v5); f[t1+5] = u5; f[t2+5] = v5;
            sumdiff(u6, v6); f[t1+6] = u6; f[t2+6] = v6;
            sumdiff(u7, v7); f[t1+7] = u7; f[t2+7] = v7;
        }
    }

    for (  ; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
    }
```

The following list gives the speed ratio between the optimized and the unoptimized DIF routine:

```
    14 == ldn;   MemSize ==   128 kB;     ratio = 1.24252
    16 == ldn;   MemSize ==   512 kB;     ratio = 1.43568
    18 == ldn;   MemSize ==     2 MB;     ratio = 1.23875
    20 == ldn;   MemSize ==     8 MB;     ratio = 1.21012
    22 == ldn;   MemSize ==    32 MB;     ratio = 1.19939
    24 == ldn;   MemSize ==   128 MB;     ratio = 1.18245
```

For sizes that are out of (level-2) cache most of the speedup is due to the memory prefetch.

Iterative versions of the algorithms

In the DIF algorithm the Haar transforms are executed at positions $f+2$, $f+4$, $f+6$, ... and the length of the transform at position $f+s$ is determined by the lowest set bit in s. Additionally, a full-length Haar transform has to be done at the beginning. As C++ code:

```
template <typename Type>
inline void haar_dif2(Type *f, ulong n)
{
    for (ulong m=n; m>=2; m>>=1)
    {
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
    }
}

template <typename Type>
void loc_dif2(Type *f, ulong n)
{
    haar_dif2(f, n);
    for (ulong z=2; z<n; z+=2)  haar_dif2(f+z,  (z&-z));
}
```

Note that the routines now take the length of the transform as second argument, not its base-2 logarithm.

With the DIT algorithm matters are slightly more complicated. A pattern can be observed by printing the binary expansions of the starting position and length of the transforms:

```
    start     length
    ...1.     ....1.
    ...11.    ....1.
    ..1..     ...1..
    ..1.1.    ....1.
```

```
    ..111.    ...1.
    ..11..    ..1..
    .1...     .1...
    .1..1.    ....1.
    .1.11.    ...1.
    .1.1..    ..1..
    .11.1.    ....1.
    .1111.    ...1.
    .111..    ..1..
    .11...    .1...
    .1....    1....
```

The lengths are again determined by the lowest bit of the start position. And we have also seen the pattern in the left column: The reversed binary words in reversed lexicographic order, see section 1.29 on page 59. The code:

```
template <typename Type>
inline void haar_dit2(Type *f, ulong n)
{
    for (ulong m=1; m<=n; m<<=1)
    {
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
    }
}
template <typename Type>
void loc_dit2(Type f, ulong n)
{
    for (ulong z=2, u=1; z<n; z+=2)
    {
        ulong s = u<<1;
        haar_dit2(f+s,  (s&-s));
        u = prev_lexrev(u);
    }
    haar_dit2(f, n);
}
```

A somewhat surprising application of the seemingly esoteric routine [FXT: `prev_lexrev()` in bits/bitlex.h]. Plus we have found a recursive algorithm [FXT: demo bits/bitlex-rec-demo.cc]:

```
void bitlex_b(ulong f, ulong n)
{
    for (ulong m=1; m<n; m<<=1)  bitlex_b(f+m, m);
    print_bin("  ", f, ldn);
}
```

10.6 Dyadic convolution

Dyadic convolution has XOR where the usual one has plus

Using [FXT: `dyadic_convolution()` in walsh/dyadiccnvl.h]

```
template <typename Type>
void dyadic_convolution(Type * restrict f, Type * restrict g, ulong ldn)
{
    walsh_wak(f, ldn);
    walsh_wak(g, ldn);
    for (ulong k=0; k<n; ++k)  g[k] *= f[k];
    walsh_wak(g, ldn);
}
```

one obtains the so-called *dyadic* convolution defined by

$$h = a \diamond b \tag{10.22a}$$

$$h_\tau := \sum_{x \wedge y = \tau} a_x \, b_y \tag{10.22b}$$

where the symbol '$\wedge$' stands for the XOR operator. Equivalently,

$$h_\tau := \sum_{x} a_x \, b_{x \wedge \tau} \tag{10.22c}$$

```
+--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
|
0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
1:   1  0  3  2   5  4  7  6   9  8 11 10  13 12 15 14
2:   2  3  0  1   6  7  4  5  10 11  8  9  14 15 12 13
3:   3  2  1  0   7  6  5  4  11 10  9  8  15 14 13 12

4:   4  5  6  7   0  1  2  3  12 13 14 15   8  9 10 11
5:   5  4  7  6   1  0  3  2  13 12 15 14   9  8 11 10
6:   6  7  4  5   2  3  0  1  14 15 12 13  10 11  8  9
7:   7  6  5  4   3  2  1  0  15 14 13 12  11 10  9  8

8:   8  9 10 11  12 13 14 15   0  1  2  3   4  5  6  7
9:   9  8 11 10  13 12 15 14   1  0  3  2   5  4  7  6
10: 10 11  8  9  14 15 12 13   2  3  0  1   6  7  4  5
11: 11 10  9  8  15 14 13 12   3  2  1  0   7  6  5  4

12: 12 13 14 15   8  9 10 11   4  5  6  7   0  1  2  3
13: 13 12 15 14   9  8 11 10   5  4  7  6   1  0  3  2
14: 14 15 12 13  10 11  8  9   6  7  4  5   2  3  0  1
15: 15 14 13 12  11 10  9  8   7  6  5  4   3  2  1  0
```

Figure 10.4: Semi-symbolic scheme for the dyadic convolution of two length-16 sequences.

The semi-symbolic scheme equivalent to the one given in figure 9.1 on page 286 is shown in figure 10.4. The dyadic convolution can be used for the multiplication of hypercomplex numbers as shown in section 20.6 on page 514.

```
+--   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
|
0:    0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
1:    1  0  3  2   5  4  7  6   9  8 11 10  13 12 15 14
2:    2  3  0  1   6  7  4  5  10 11  8  9  14 15 12 13
3:    3  2  1  0   7  6  5  4  11 10  9  8  15 14 13 12

4:    4  5  6  7   0  1  2  3  12 13 14 15   8  9 10 11
5:    5  4  7  6   1  0  3  2  13 12 15 14   9  8 11 10
6:    6  7  4  5   2  3  0  1  14 15 12 13  10 11  8  9
7:    7  6  5  4   3  2  1  0  15 14 13 12  11 10  9  8

8:   16 17 18 19  20 21 22 23  24 25 26 27  28 29 30 31
9:   17 16 19 18  21 20 23 22  25 24 27 26  29 28 31 30
10:  18 19 16 17  22 23 20 21  26 27 24 25  30 31 28 29
11:  19 18 17 16  23 22 21 20  27 26 25 24  31 30 29 28

12:  20 21 22 23  16 17 18 19  28 29 30 31  24 25 26 27
13:  21 20 23 22  17 16 19 18  29 28 31 30  25 24 27 26
14:  22 23 20 21  18 19 16 17  30 31 28 29  26 27 24 25
15:  23 22 21 20  19 18 17 16  31 30 29 28  27 26 25 24
```

Figure 10.5: Semi-symbolic scheme for the dyadic convolution of two zero-padded sequences.

The nearest equivalent to the acyclic convolution can be computed using a sequence that has both prepended and appended runs of $n/2$ zeros, the corresponding table is shown in figure 10.5.

An scheme similar to that of the negacyclic convolution can be obtained via

```
walsh_wal_dif2_core(f, ldn);
walsh_wal_dif2_core(g, ldn);
ulong n = (1UL<<ldn);
for (ulong i=0,j=n-1; i<j; --j,++i)  fht_mul(f[i], f[j], g[i], g[j], 0.5);
walsh_wal_dit2_core(g, ldn);
```

where `fht_mul()` is the operation used for the convolution with fast Hartley transforms [FXT: file convolution/fhtmulsqr.h]:

```
template <typename Type>
static inline  void
fht_mul(Type xi, Type xj, Type &yi, Type &yj, double v)
// yi <-- v*( 2*xi*xj + xi*xi - xj*xj )
// yj <-- v*( 2*xi*xj - xi*xi + xj*xj )
{
    Type h1p = xi,  h1m = xj;
```

```
          +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
          |
          0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
          1:   1  0  3  2   5  4  7  6   9  8 11 10  13 12 15 14
          2:   2  3  0  1   6  7  4  5  10 11  8  9  14 15 12 13
          3:   3  2  1  0   7  6  5  4  11 10  9  8  15 14 13 12

          4:   4  5  6  7   0  1  2  3  12 13 14 15   8  9 10 11
          5:   5  4  7  6   1  0  3  2  13 12 15 14   9  8 11 10
          6:   6  7  4  5   2  3  0  1  14 15 12 13  10 11  8  9
          7:   7  6  5  4   3  2  1  0  15 14 13 12  11 10  9  8

          8:   8  9 10 11  12 13 14 15   0- 1- 2- 3-  4- 5- 6- 7-
          9:   9  8 11 10  13 12 15 14   1- 0- 3- 2-  5- 4- 7- 6-
         10:  10 11  8  9  14 15 12 13   2- 3- 0- 1-  6- 7- 4- 5-
         11:  11 10  9  8  15 14 13 12   3- 2- 1- 0-  7- 6- 5- 4-

         12:  12 13 14 15   8  9 10 11   4- 5- 6- 7-  0- 1- 2- 3-
         13:  13 12 15 14   9  8 11 10   5- 4- 7- 6-  1- 0- 3- 2-
         14:  14 15 12 13  10 11  8  9   6- 7- 4- 5-  2- 3- 0- 1-
         15:  15 14 13 12  11 10  9  8   7- 6- 5- 4-  3- 2- 1- 0-
```

Figure 10.6: Semi-symbolic scheme for the dyadic equivalent of the negacyclic convolution. Negative contributions to a bucket have a minus appended.

```
    Type s1 = h1p + h1m,  d1 = h1p - h1m;
    Type h2p = yi,  h2m = yj;
    yi = (h2p * s1 + h2m * d1) * v;
    yj = (h2m * s1 - h2p * d1) * v;
}
```

One obtains a convolution corresponding to the semi-symbolic table shown in figure 10.6.

10.7 The Walsh transform: Walsh-Paley basis

A Walsh transform with a different (ordering of the) basis can be obtained by [FXT: file walsh/walshpal.h]:

```
template <typename Type>
void walsh_pal(Type *f, ulong ldn)
{
    const ulong n = 1UL<<ldn;
    revbin_permute(f, n);
    walsh_wak(f, ldn);
// =^=
//    walsh_wak(f, ldn);
//    revbin_permute(f, n);
}
```

The basis functions are shown in figure 10.7 on the following page. Actually one can also apply the revbin permutation before the transform. That is,

$$W_p \;=\; W_k\, R \;=\; R\, W_k \tag{10.23}$$

One has for W_p

$$W_p \;=\; G\, W_p\, G \;=\; G^{-1}\, W_p\, G^{-1} \tag{10.24}$$

$$\;=\; Z\, W_p\, Z \;=\; Z^{-1}\, W_p\, Z^{-1} \tag{10.25}$$

where Z denotes the zip permutation (see section 2.5 on page 87) and G denotes the Gray permutation (see section 2.8 on page 91).

A function that computes the k-th base function of the transform is [FXT: `walsh_pal_basefunc()` in walsh/walshbasefunc.h]:

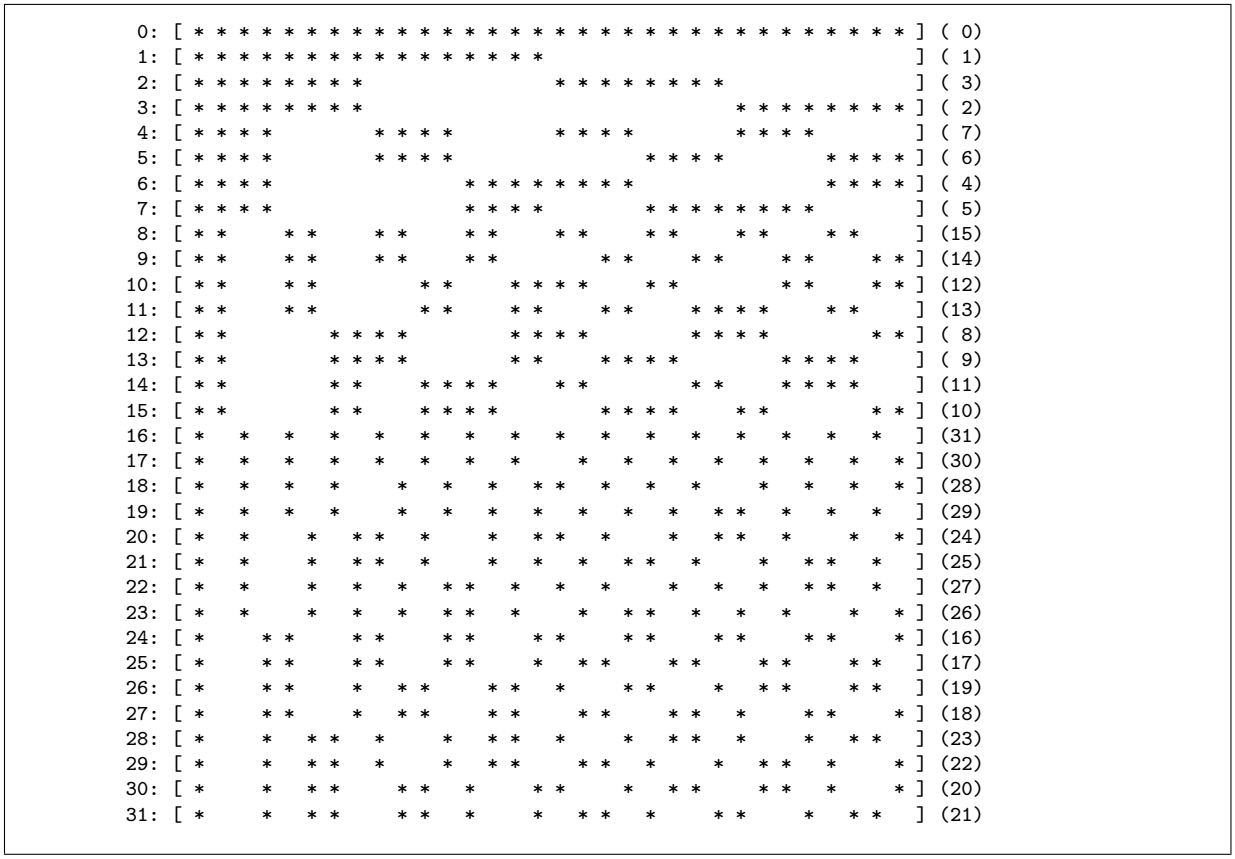


Figure 10.7: Basis functions for the Walsh transform (Walsh-Paley basis). Asterisks denote the value +1, blank entries denote −1.

```
template <typename Type>
void walsh_pal_basefunc(Type *f, ulong n, ulong k)
{
    k = revbin(k, ld(n));
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

10.8 Sequency ordered Walsh transforms

The term corresponding to the frequency of the Fourier basis functions is the so-called *sequency* of the Walsh functions, the number of the changes of sign of the individual functions. Note that the sequency of a signal with frequency f usually is $2\,f$.

If the basis functions shall be ordered by their sequency one can use

```
const ulong n = (1UL<<ldn);
walsh_wak(f, ldn);
revbin_permute(f, n);
inverse_gray_permute(f, n);
```

That is

$$W_w \;=\; G^{-1}\,R\,W_k \;=\; W_k\,R\,G \tag{10.26}$$

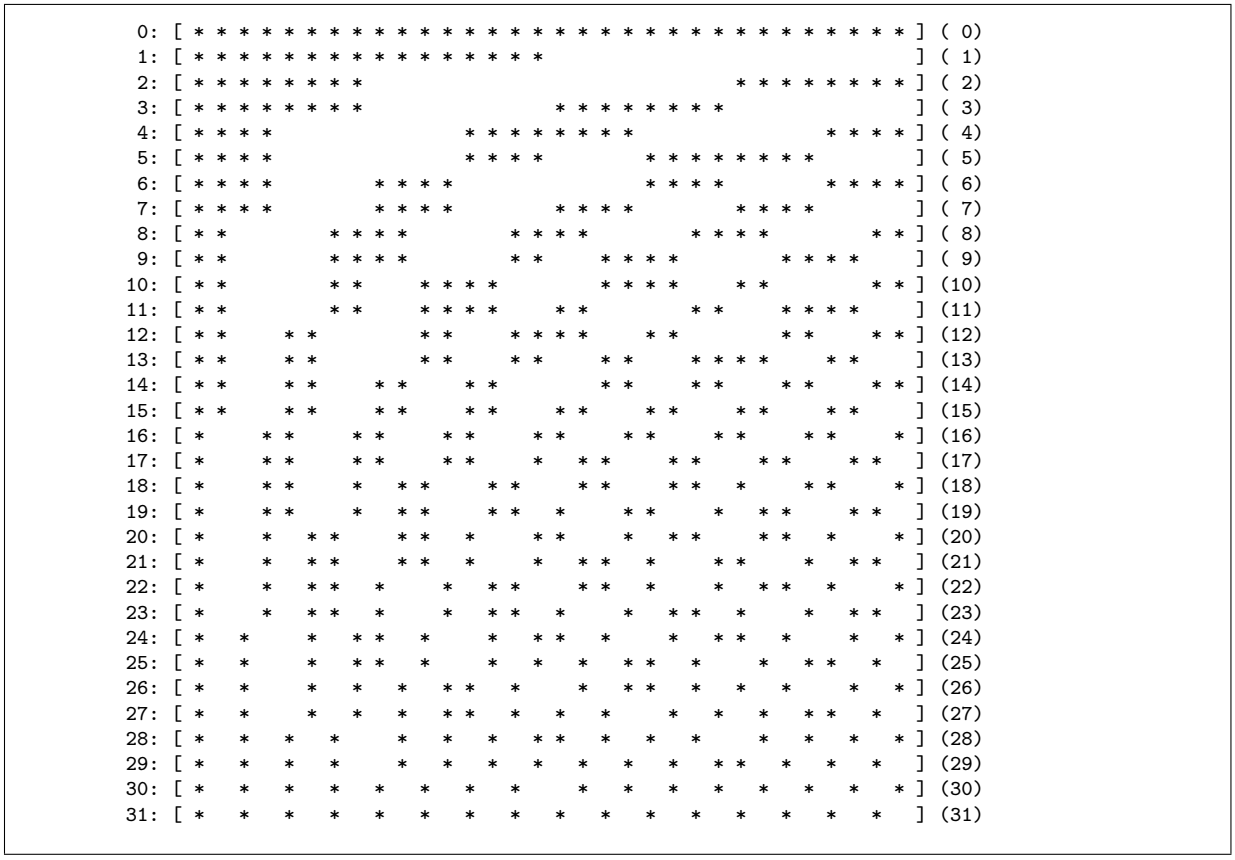


Figure 10.8: Basis functions for the sequency-ordered Walsh transform (Walsh-Kacmarz basis). Asterisks denote the value $+1$, blank entries denote -1.

A function that computes the k-th base function of the transform is [FXT: `walsh_wal_basefunc()` in walsh/walshbasefunc.h]:

```
template <typename Type>
void walsh_wal_basefunc(Type *f, ulong n, ulong k)
{
    k = revbin(k, ld(n)+1);
    k = gray_code(k);
//    // =^=
//    k = revbin(k, ld(n));
//    k = green_code(k);
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

A version of the transform that avoids the Gray permutation is based on [FXT: file walsh/walshwal.h]

```
template <typename Type>
void walsh_wal_dif2_core(Type *f, ulong ldn)
// Core routine for sequency ordered Walsh transform.
// Radix-2 decimation in frequency (DIF) algorithm.
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=2; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
```

```
        for (ulong r=0; r<n; r+=m)
        {
            ulong j;
            for (j=0; j<m4; ++j)
            {
                ulong t1 = r+j;
                ulong t2 = t1+mh;
                double u = f[t1];
                double v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
            for (  ; j<mh; ++j)
            {
                ulong t1 = r+j;
                ulong t2 = t1+mh;
                double u = f[t1];
                double v = f[t2];
                f[t1] = u + v;
                f[t2] = v - u;  // reversed
            }
        }
    }
    if ( ldn )
    {
        // ulong ldm=1;
        const ulong m = 2; //(1UL<<ldm);
        const ulong mh = 1; //(m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong j = 0;
//            for (ulong j=0; j<mh; ++j)
            {
                ulong t1 = r+j;
                ulong t2 = t1+mh;
                double u = f[t1];
                double v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

The transform still needs the revbin permutation:

```
template <typename Type>
inline void walsh_wal(Type *f, ulong ldn)
{
    revbin_permute(f, (1UL<<ldn));
    walsh_wal_dif2_core(f, ldn);
//  =^=
//    walsh_wal_dit2_core(f, ldn);
//    revbin_permute(f, (1UL<<ldn));
}
```

A decimation in time (DIT) version of the core-routine is also given in [FXT: file walsh/walshwal.h]. The procedure `gray_permute()` is introduced in section 2.8 on page 91.

The sequence of statements

```
walsh_gray(f, ldn);
grs_negate(f, n);
revbin_permute(f, n);
```

is equivalent to `walsh_wal(f, ldn)` and might be faster for large arrays. We have

$$W_w \;=\; R\,Q\,W_g \;=\; W_g^{-1}\,R\,Q \tag{10.27}$$

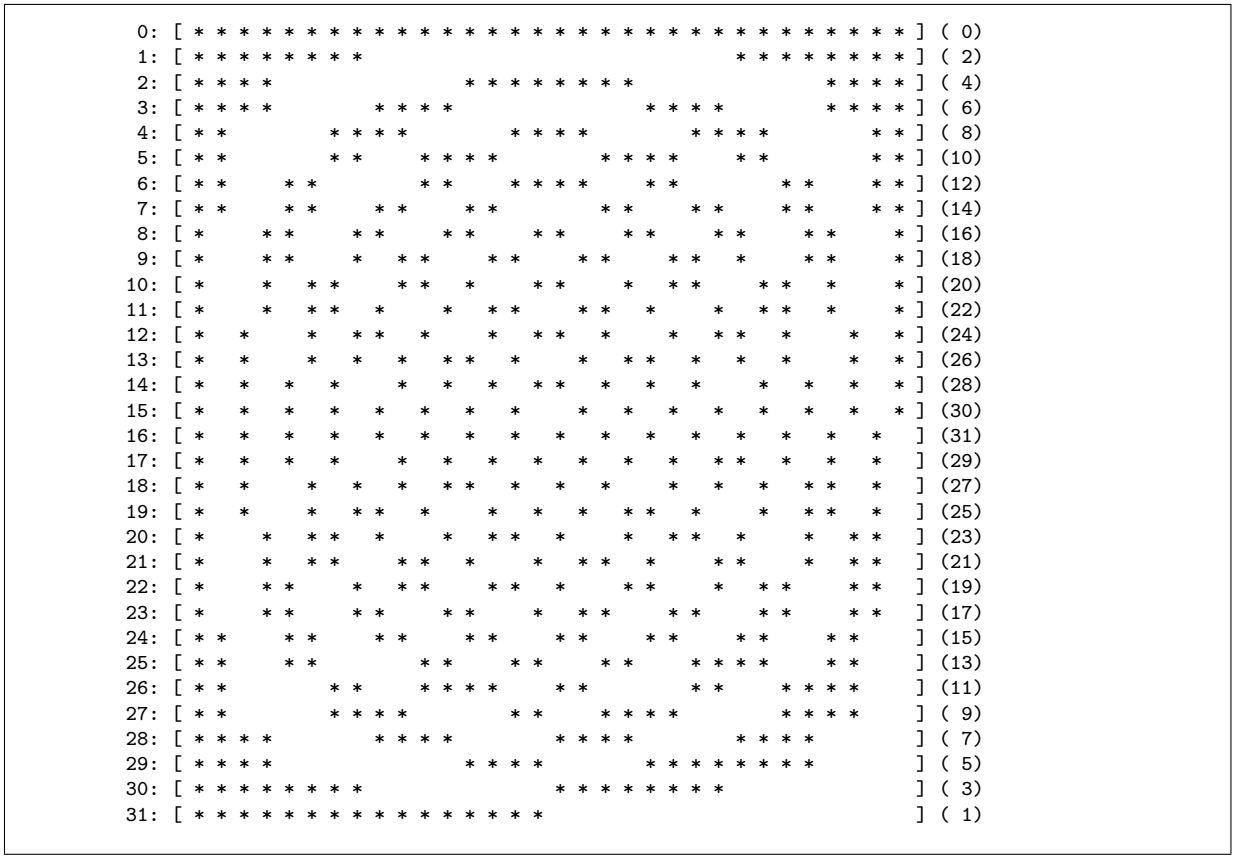


Figure 10.9: Basis functions for the reversed sequency ordered Walsh transform. Asterisks denote the value +1, blank entries denote −1.

Even/odd ordering of sequencies

An alternative ordering of the base functions (first even sequencies ascending then odd sequencies descending [FXT: `walsh_wal_rev()` in walsh/walshwalrev.h]) can be obtained by either of (with `n=1UL<<ldn`)

```
revbin_permute(f, n);
gray_permute(f, n);
walsh_wak(f,ldn);
```

```
walsh_wak(f,ldn);
inverse_gray_permute(f, n);
revbin_permute(f, n);
```

```
zip_rev(f, n);
walsh_wal(f, ldn);
```

```
walsh_wal(f, ldn);
unzip_rev(f, n);
```

```
walsh_wak(f, ldn);
inverse_gray_permute(f, n);
revbin_permute(f, n);
```

```
revbin_permute(f, n);
walsh_gray(f, ldn);
grs_negate(f, n);
```

That is,

$$\overline{W}_w = W_k\, G\, R = R\, G^{-1}\, W_k \tag{10.28}$$
$$= W_w\, \overline{Z} = \overline{Z}^{-1}\, W_w \tag{10.29}$$
$$= Q\, W_g\, R \tag{10.30}$$

However, an implementation that is more efficient uses the core-routines that have the Gray permutation 'absorbed' [FXT: `walsh_wal_rev()` in walsh/walshwalrev.h]:

```
template <typename Type>
inline void walsh_wal_rev(Type *f, ulong ldn)
{
    revbin_permute(f, (1UL<<ldn));
    walsh_wal_dit2_core(f, ldn);
    // =^=
//    walsh_wal_dif2_core(f, ldn);
//    revbin_permute(f, n);
}
```

This implementation uses the fact that

$$\overline{W}_w \quad = \quad R\, W_w\, R \tag{10.31}$$

We can do still better, the following sequence of statements gives the same transform:

```
    revbin_permute(f, n);
    walsh_gray(f, ldn);
    grs_negate(f, n);
```

Similar relations as for the transform with Walsh-Paley basis (10.24 and 10.25 on page 319) hold for W_w:

$$W_w \quad = \quad G\, W_w\, G \;=\; G^{-1}\, W_w\, G^{-1} \tag{10.32}$$

$$\quad = \quad \overline{Z}\, W_w\, \overline{Z} \;=\; \overline{Z}^{-1}\, W_w\, \overline{Z}^{-1} \tag{10.33}$$

The k-th base function of the transform can be computed as [FXT: `walsh_wal_rev_basefunc()` in walsh/walshbasefunc.h]

```
template <typename Type>
void walsh_wal_rev_basefunc(Type *f, ulong n, ulong k)
{
    k = revbin(k, ld(n));
    k = gray_code(k);
    // =^=
//    k = green_code(k);
//    k = revbin(k, ld(n));
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

Transforms with all sequencies $n/2$ or $n/2-1$

The next variant of the Walsh transform has the interesting feature that the basis functions for a length-n transform have only sequencies $n/2$ and $n/2-1$ at the even and odd indices, respectively. The transform is self-inverse (the basis is shown in figure 10.10 on the next page) and can be obtained via [FXT: `walsh_q1()` in walsh/walshq.h]

```
template <typename Type>
void walsh_q1(Type *f, ulong ldn)
{
    ulong n = 1UL << ldn;
    grs_negate(f, n);
    walsh_gray(f, ldn);
    revbin_permute(f, n);
}
```

A different transform with sequency $n/2$ for the first half of the basis, sequency $n/2-1$ for the second half ([FXT: `walsh_q2()` in walsh/walshq.h], basis shown in figure 10.11 on page 326) is computed by

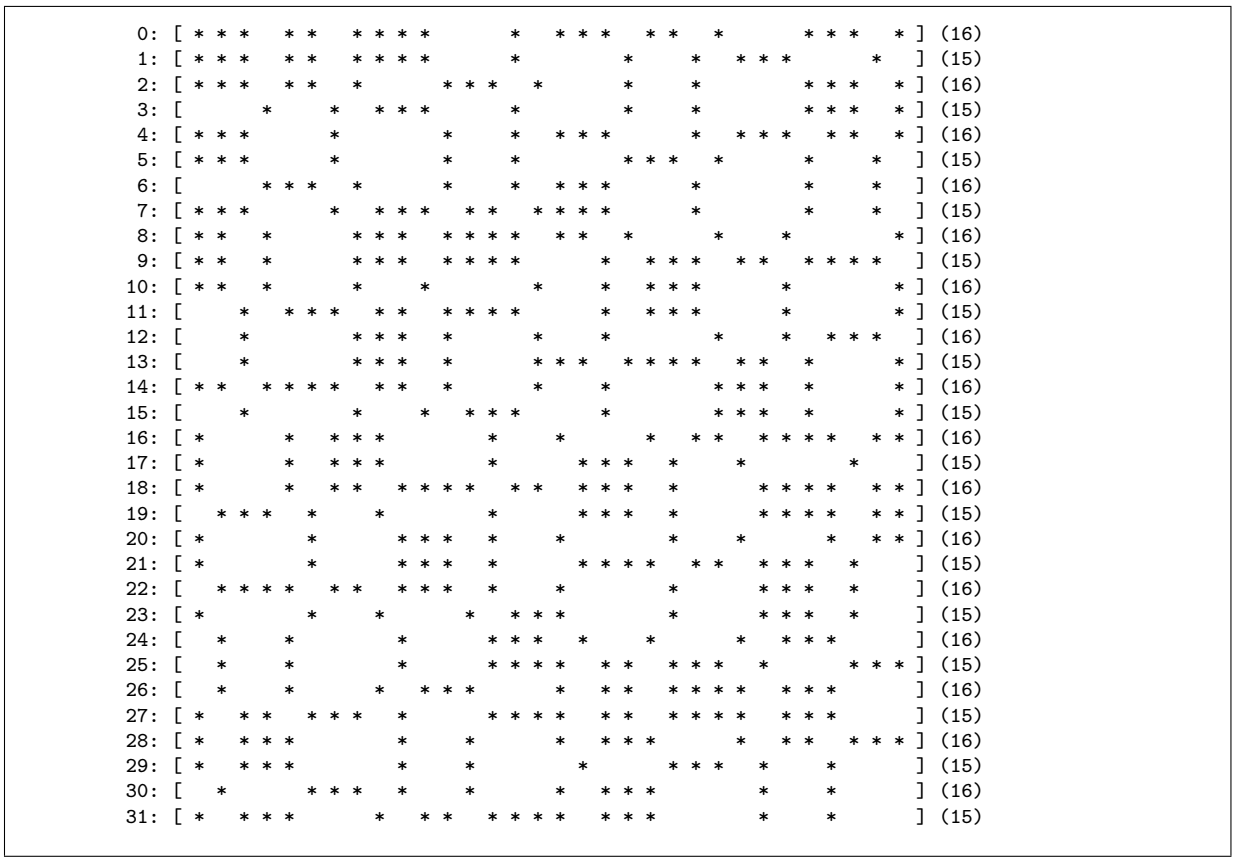


Figure 10.10: Basis functions for a self-inverse Walsh transform that has sequencies $n/2$ and $n/2-1$ only. Asterisks denote the value $+1$, blank entries denote -1.

```
template <typename Type>
void walsh_q2(Type *f, ulong ldn)
{
    ulong n = 1UL << ldn;
    revbin_permute(f, n);
    grs_negate(f, n);
    walsh_gray(f, ldn);
// =^=
//    grs_negate(f, n);
//    revbin_permute(f, n);
//    walsh_gray(f, ldn);
}
```

One has:

$$W_{q2} \quad = \quad R\,W_{q1}\,R \tag{10.34}$$

The base functions of the transforms can be computed as [FXT: file walsh/walshbasefunc.h]

```
template <typename Type>
void walsh_q1_basefunc(Type *f, ulong n, ulong k)
{
    ulong qk = (grs_negative_q(k) ? 1 : 0);
    k = gray_code(k);
    k = revbin(k, ld(n));
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        ulong qi = (grs_negative_q(i) ? 1 : 0);
```

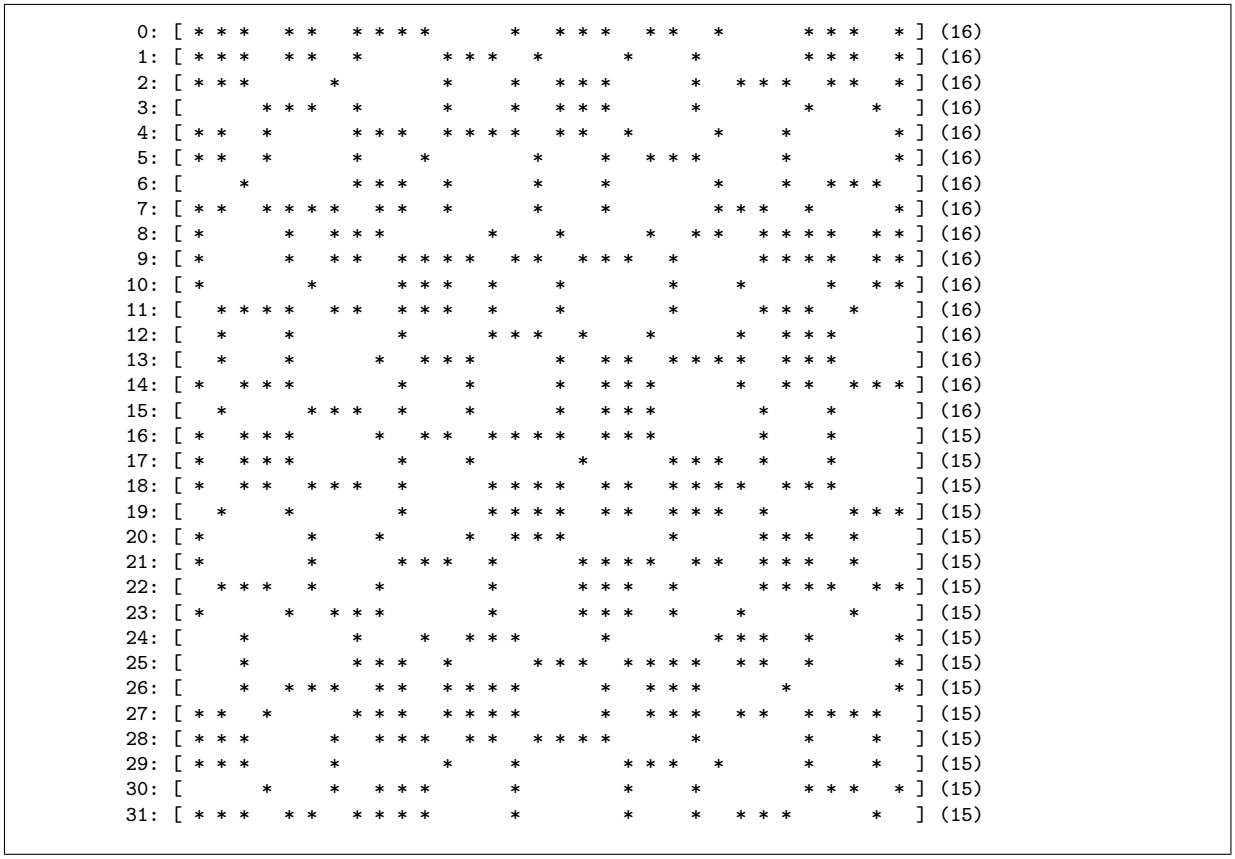


Figure 10.11: Basis functions for a self-inverse Walsh transform (second form) that has sequencies $n/2$ and $n/2-1$ only. Asterisks denote the value $+1$, blank entries denote -1.

```
        x ^= (qk ^ qi);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

and

```
template <typename Type>
void walsh_q2_basefunc(Type *f, ulong n, ulong k)
{
    ulong qk = (grs_negative_q(k) ? 1 : 0);
    k = revbin(k, ld(n));
    k = gray_code(k);
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        ulong qi = (grs_negative_q(i) ? 1 : 0);
        x ^= (qk ^ qi);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

The function `grs_negative_q()` is described in section 1.16.4 on page 37.

10.9 The slant transform

The *slant transform* can be implemented using a Walsh Transform and just a little pre/post-processing [FXT: file walsh/slant.cc]:

```
void slant(double *f, ulong ldn)
{
    walsh_wak(f, ldn);

    ulong n = 1UL<<ldn;
    for (ulong ldm=0; ldm<ldn-1; ++ldm)
    {
        ulong m = 1UL<<ldm;  // m = 1, 2, 4, 8, ..., n/4
        double N = m*2, N2 = N*N;
        double a = sqrt(3.0*N2/(4.0*N2-1.0));
        double b = sqrt(1.0-a*a); // == sqrt((N2-1)/(4*N2-1));
        for (ulong j=m; j<n-1; j+=4*m)
        {
            ulong t1 = j;
            ulong t2 = j + m;
            double f1 = f[t1],  f2 = f[t2];
            f[t1] = a * f1 - b * f2;
            f[t2] = b * f1 + a * f2;
        }
    }
}
```

Apart from the Walsh transform only an amount of work linear with the array size has to be done: The inner loop accesses the elements in strides of 4, 8, 16, ..., 2^{n-1}.

The inverse transform is:

```
void inverse_slant(double *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    ulong ldm=ldn-2;
    do
    {
        ulong m = 1UL<<ldm;  // m = n/4, n/2, ..., 4, 2, 1
        double N = m*2, N2 = N*N;
        double a = sqrt(3.0*N2/(4.0*N2-1.0));
        double b = sqrt(1.0-a*a); // == sqrt((N2-1)/(4*N2-1));
        for (ulong j=m; j<n-1; j+=4*m)
        {
            ulong t1 = j;
            ulong t2 = j + m;
            double f1 = f[t1],  f2 = f[t2];
            f[t1] = b * f2 + a * f1;
            f[t2] = a * f2 - b * f1;
        }
    }
    while ( ldm-- );

    walsh_wak(f, ldn);
}
```

A sequency ordered version of the transform can be implemented as follows:

```
void slant_seq(double *f, ulong ldn)
{
    slant(f, ldn);

    ulong n = 1UL<<ldn;
    inverse_gray_permute(f, n);
    unzip_rev(f, n);
    revbin_permute(f, n);
}
```

This implementation can be optimized by combining the involved permutations, see [105].

The inverse is obtained by calling the inverse operations in reversed order:

```
void inverse_slant_seq(double *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    revbin_permute(f, n);
    zip_rev(f, n);
    gray_permute(f, n);

    inverse_slant(f, ldn);
}
```

10.10 The Reed-Muller transform

How to make a Reed-Muller transform out of a Walsh transform:
'*Replace* `(u+v)` *by* `(u)` *and* `(u-v)` *by* `(u XOR v)`*, done.*'

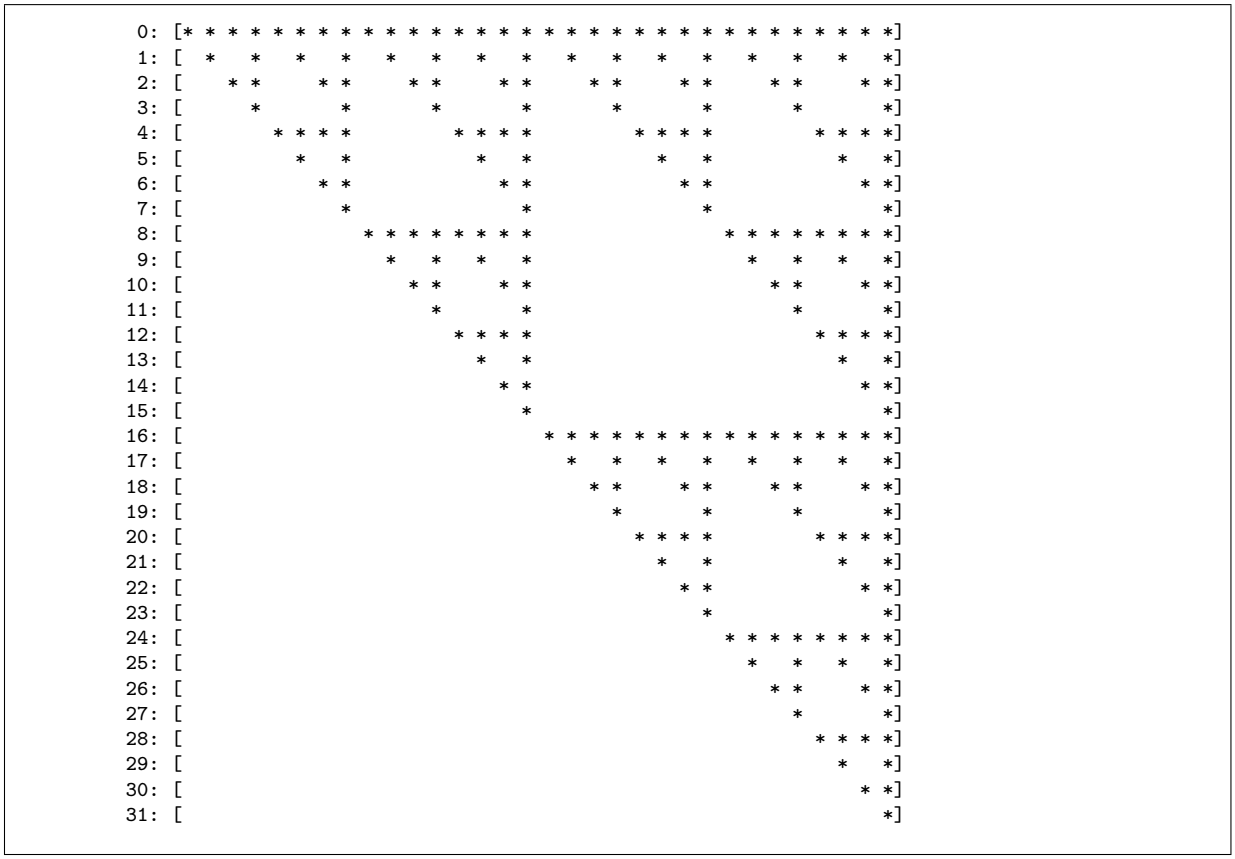


Figure 10.12: Basis functions for the Reed-Muller transform. Asterisks denote positions where the functions are equal to one. Blank entries correspond to zero.

There we go [FXT: file walsh/reedmuller.h]:

```
template <typename Type>
void word_reed_muller_dif2(Type *f, ulong ldn)
// Reed-Muller Transform.
// Radix-2 decimation in frequency (DIF) algorithm
// Self-inverse.
// Type must have the XOR operator.
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u;
                f[t2] = u ^ v;
            }
        }
```

```
    }
}
```

This routine is almost identical to [FXT: `walsh_wak_dif2()` in walsh/walshwak2.h]. The only changes are

```
    walsh          f[t1] = u + v;
    walsh          f[t2] = u - v;

    reedmuller     f[t1] = u;
    reedmuller     f[t2] = u ^ v;
```

The decimation in time algorithm can be obtained from [FXT: `walsh_wak_dit2()` in walsh/walshwak2.h] by the very same changes.

As given, the transforms work word-wise, if the bit-wise transform is wanted use

```
template <typename Type>
inline void bit_reed_muller(Type *f, ulong ldn)
{
    word_reed_muller_dif2(f, ldn);
    ulong n = 1UL << ldn;
    for (ulong k=0; k<n; ++k)  f[k] = yellow_code(f[k]);
}
```

The `yellow_code()` (see section 1.19 on page 41) can also be applied before the main loop. In fact, the yellow code *is* the Reed-Muller transform on a binary word.

The other 'color-transforms' of section 1.19 on page 41 lead to variants of the Reed-Muller transform, the blue code gives another self-inverse transform, the red code and the cyan code give transforms R and C so that

$$R\,R\,R = \mathrm{id} \qquad R^{-1} = R\,R = C \tag{10.35}$$

$$C\,C\,C = \mathrm{id} \qquad C^{-1} = C\,C = R \tag{10.36}$$

$$R\,C = C\,R = \mathrm{id} \tag{10.37}$$

In fact, all relations given in the referenced section hold.

As can be seen from the 'atomic' matrices (relations 1.12c ... 1.12f on page 46) the four transforms corresponding to the 'color codes' are obtained by

```
 Walsh:   f[t1] = u + v;    f[t2] = u - v;

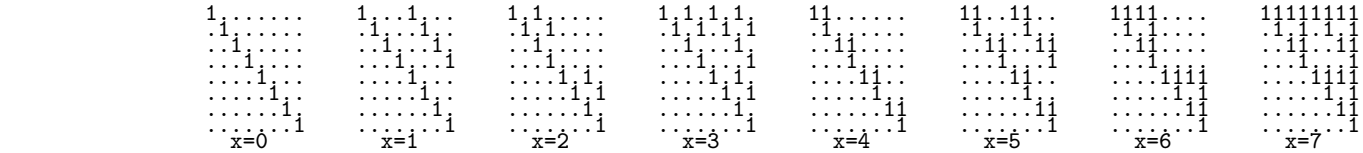
    B:    f[t1] = u ^ v;    f[t2] = v;
    Y:    f[t1] = u;        f[t2] = u ^ v;  (Reed-Muller transform)
    R:    f[t1] = v;        f[t2] = u ^ v;
    C:    f[t1] = u ^ v;    f[t2] = u;
```

The basis functions of the transforms are shown in figure 10.13.

```
          blue              yellow              red               cyan
    1...............  1111111111111111  1111111111111111  ...............1
    11..............  .1.1.1.1.1.1.1.1  1.1.1.1.1.1.1.1.  ..............11
    1.1.............  ..11..11..11..11  11..11..11..11..  .............1.1
    1111............  ...1...1...1...1  1...1...1...1...  ............1111
    1...1...........  ....1111....1111  1111....1111....  ...........1...1
    11..11..........  .....1.1.....1.1  1.1.....1.1.....  ..........11..11
    1.1.1.1.........  ......11......11  11......11......  .........1.1.1.1
    11111111........  .......1.......1  1.......1.......  ........11111111
    1.......1.......  ........11111111  11111111........  .......1.......1
    11......11......  ........1.1.1.1.  1.1.1.1.........  ......11......11
    1.1.....1.1.....  ........11..11..  11..11..........  .....1.1.....1.1
    1111....1111....  ........1...1...  1...1...........  ....1111....1111
    1...1...1...1...  ........1111....  1111............  ...1...1...1...1
    11..11..11..11..  ........1.1.....  1.1.............  ..11..11..11..11
    1.1.1.1.1.1.1.1.  ........11......  11..............  .1.1.1.1.1.1.1.1
    1111111111111111  ........1.......  1...............  1111111111111111
```

Figure 10.13: Basis functions of the length-16 blue, red, red and cyan transforms.

The symbolic powering idea from section 1.19 on page 41 leads to transforms with bases (using eight element arrays and the yellow code):



The program [FXT: demo bits/bitxtransforms-demo.cc] gives the matrices for 64-bit words.

Functions that are the word-wise equivalents of the Gray code are given in [FXT: file aux1/wordgray.h]:

```
template <typename Type>
void word_gray(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong k=0;  k<n-1;  ++k)  f[k] ^= f[k+1];
}
```

and

```
template <typename Type>
void inverse_word_gray(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong s=1; s<n; s*=2)
    {
        // word_gray ** s:
        for (ulong k=0, j=k+s;  j<n;  ++k,++j)  f[k] ^= f[j];
    }
}
```

The routine can be improved by using the equivalent of the 'bitwise integration':

```
    ulong x = 0,  k = n;
    while ( k-- )  { x ^= f[k];  f[k] = x; }
```

As one might suspect, these are related to the Reed-Muller transform. Writing Y ('yellow') for the Reed-Muller transform, g for the word-wise Gray code and S_k for the cyclic shift by k words (word zero is moved to position k) one has

$$Y\,S_{+1}\,Y = g \tag{10.38}$$

$$Y\,S_{-1}\,Y = g^{-1} \tag{10.39}$$

$$Y\,S_{k}\,Y = g^{k} \tag{10.40}$$

These are exactly the relations 1.10a ... 1.10c on page 45 for the bit-wise transforms. For $k >= 0$ the operator S_k corresponds to the shift toward element zero (use [FXT: `rotate_sgn()` in perm/rotate.h]). The power of the word-wise Gray code is perfectly equivalent to the bit-wise version:

```
template <typename Type>
void word_gray_pow(Type *f, ulong n, ulong x)
{
    for (ulong s=1; s<n; s*=2)
    {
        if ( x & 1)
        {
            // word_gray ** s:
            for (ulong k=0, j=k+s;  j<n;  ++k,++j)  f[k] ^= f[j];
        }
        x >>= 1;
    }
}
```

Let e be the green code operator, then for the 'blue-variant':

$$B\,S_{+1}\,B = e^{-1} \tag{10.41}$$

$$B\,S_{-1}\,B = e \tag{10.42}$$

$$B\,S_{k}\,B = e^{-k} \tag{10.43}$$

Further,

$$C\,S_k\,R = e^k \tag{10.44}$$
$$C\,e^k\,R = S_k \tag{10.45}$$

The transforms as Kronecker products (all operations are modulo two):

$$B_n = \bigotimes_{k=1}^{\log_2(n)} B_2 \quad \text{where} \quad B_2 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \tag{10.46a}$$
$$Y_n = \bigotimes_{k=1}^{\log_2(n)} Y_2 \quad \text{where} \quad Y_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \tag{10.46b}$$
$$R_n = \bigotimes_{k=1}^{\log_2(n)} R_2 \quad \text{where} \quad R_2 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \tag{10.46c}$$
$$C_n = \bigotimes_{k=1}^{\log_2(n)} C_2 \quad \text{where} \quad C_2 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \tag{10.46d}$$

A function that computes the k-th base function of the word-wise transform is [FXT: file walsh/reedmuller.h]:

```
template <typename Type>
inline void word_reed_muller_basefunc(Type *f, ulong n, ulong k)
{
    for (ulong i=0; i<n; ++i)
    {
        ulong x = (n-1-i) & (k);
        f[i] = ( 0==x ? +1 : 0 );
    }
}
```

10.11 The arithmetic transform

How to make a arithmetic transform out of a Walsh transform:
'*Forward: replace* `(u+v)` *by* `(u)` *and* `(u-v)` *by* `(v-u)`.
Backward: replace `(u+v)` *by* `(u)` *and* `(u-v)` *by* `(u+v)`. '

On to the code [FXT: file walsh/arithtransform.h]:

```
template <typename Type>
void arith_transform_plus(Type *f, ulong ldn)
// Arithmetic Transform (positive sign).
// Radix-2 decimation In Frequency (DIF) algorithm.
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u;
                f[t2] = u + v;
```

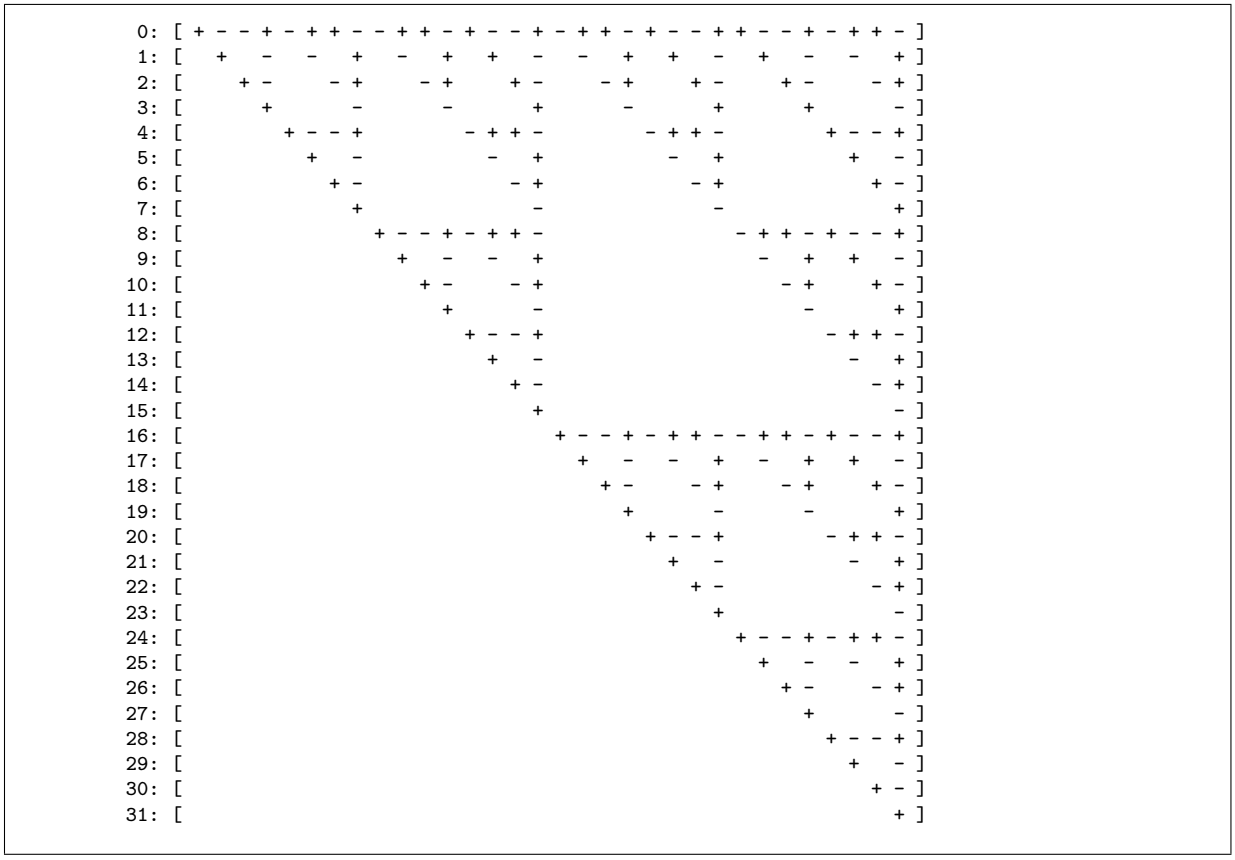


Figure 10.14: Basis functions for the Arithmetic transform (A^-, the one with the minus sign). The values are ± 1, blank entries denote 0.

```
          }
      }
   }
}
```

and

```
template <typename Type>
void arith_transform_minus(Type *f, ulong ldn)
// Arithmetic Transform (negative sign).
// Radix-2 decimation In Frequency (DIF) algorithm.
// Inverse of arith_transform_plus().
{
  [--snip--]
                f[t1] = u;
                f[t2] = v - u;
  [--snip--]
}
```

The length-2 transforms can be written as

$$A_2^+ v = \begin{bmatrix} +1 & 0 \\ +1 & +1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ a+b \end{bmatrix} \tag{10.47}$$

$$A_2^- v = \begin{bmatrix} +1 & 0 \\ -1 & +1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ b-a \end{bmatrix} \tag{10.48}$$

That the one with the minus is called the forward transform is tradition. Similar to the Fourier transform we avoid the forward- backward- naming scheme and put:

```
template <typename Type>
```

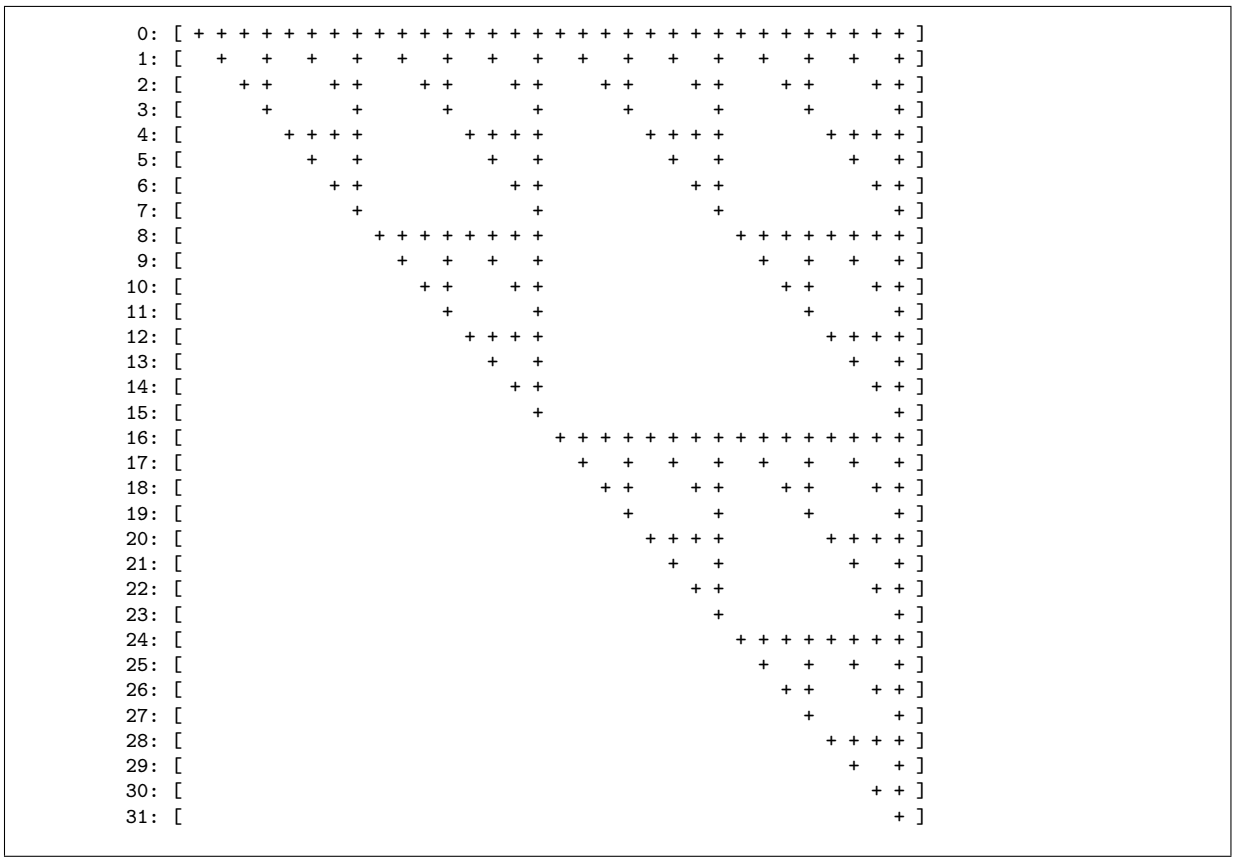


Figure 10.15: Basis functions for the inverse Arithmetic transform (A^+, the one without minus sign). The values are +1, blank entries denote 0.

```
inline void arith_transform(Type *f, ulong ldn, int is)
{
    if ( is>0 )  arith_transform_plus(f, ldn);
    else         arith_transform_minus(f, ldn);
}
```

In Kronecker product notation (see section 10.2 on page 305) the arithmetic transform and its inverse can be written as

$$A_2^+ = \begin{bmatrix} +1 & 0 \\ +1 & +1 \end{bmatrix} \qquad A_n^+ = \bigotimes_{k=1}^{\log_2(n)} A_2^+ \tag{10.49a}$$

$$A_2^- = \begin{bmatrix} +1 & 0 \\ -1 & +1 \end{bmatrix} \qquad A_n^- = \bigotimes_{k=1}^{\log_2(n)} A_2^- \tag{10.49b}$$

To establish the relation to the Walsh transform recall that its decomposition as a Kronecker product is

$$W_2 = \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \qquad W_n = \bigotimes_{k=1}^{\log_2(n)} W_2 \tag{10.50}$$

Now as $(W_2\, A_2^+)\, A_2^- = W_2$ the expression in parenthesis is the matrix that transforms the 2-point arithmetic transform (with the negative sign) to the Walsh transform. Similarly, as $(\frac{1}{2}\, A_2^+\, W_2)\, W_2 = A_2^+$, the matrix leading to the conversion from Walsh to arithmetic transform can be determined. One finds:

$$\left(W_2\, A_2^+\right)\, A_2^- \;=\; W_2 = \begin{bmatrix} +2 & +1 \\ 0 & -1 \end{bmatrix} A_2^- \tag{10.51a}$$

$$\left(W_2\, A_2^-\right)\, A_2^+ \;=\; W_2 = \begin{bmatrix} 0 & +1 \\ +2 & -1 \end{bmatrix} A_2^+ \tag{10.51b}$$

$$\left(\frac{1}{2}\, A_2^-\, W_2\right)\, W_2 \;=\; A_2^- = \frac{1}{2}\begin{bmatrix} +1 & +1 \\ 0 & -2 \end{bmatrix} W_2 \tag{10.51c}$$

$$\left(\frac{1}{2}\, A_2^+\, W_2\right)\, W_2 \;=\; A_2^+ = \frac{1}{2}\begin{bmatrix} +1 & +1 \\ +2 & 0 \end{bmatrix} W_2 \tag{10.51d}$$

The Kronecker product of the given matrices gives the converting transform. For example, using relation 10.51a, define

$$T_n \;:=\; \bigotimes_{k=1}^{\log_2(n)} \begin{bmatrix} +2 & +1 \\ 0 & -1 \end{bmatrix} \tag{10.52}$$

Then T_n converts a arithmetic (minus sign) transform to a Walsh transform: $W_n = T_n\, A_n^-$. For the relations between the arithmetic transform, the Reed-Muller transform and the Walsh transform, see [102].

10.12 Hadamard and conference matrices *

If we denote the transform matrix for an N-point Walsh transform by $\mathbf{H}$, then

$$\mathbf{H}\,\mathbf{H}^T \;=\; N\ \mathrm{id} \tag{10.53}$$

where id is the unit matrix. We see that H is orthogonal (up to normalization) and its determinant equals

$$\det(\mathbf{H}) \;=\; \det\left(\mathbf{H}\,\mathbf{H}^T\right)^{1/2} \;=\; N^{N/2} \tag{10.54}$$

Further, all entries are either $+1$ or -1. An orthogonal matrix with these properties is called a *Hadamard matrix*. We know that for $N = 2^n$ we always can find such a matrix. For $N = 2$ we have

$$\mathbf{H}_2 \;=\; \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix}$$

and we can construct (see section 10.2 on page 305) $\mathbf{H}_{2N}$ from $\mathbf{H}_N$ via

$$\mathbf{H}_n \;=\; \begin{bmatrix} +\mathbf{H}_{N/2} & +\mathbf{H}_{N/2} \\ +\mathbf{H}_{N/2} & -\mathbf{H}_{N/2} \end{bmatrix} \;=\; \mathbf{H}_2 \otimes \mathbf{H}_{N/2}$$

The problem of determining Hadamard matrices (especially for N not a power of two) comes from combinatorics. Hadamard matrices of size $N \times N$ can only exist if $N = 1$, $N = 2$ or $N = 4\,k$.

10.12.1 Hadamard matrices via LFSR

We start with a construction for certain Hadamard matrices for N a power of two that uses m-sequences that are created by shift registers (see sections 6.1 on page 217 and 6.3 on page 221). Figure 10.16 on the facing page shows three Hadamard matrices that were constructed as follows:

- Choose $N = 2^n$ and create a maximum length binary shift register sequence S of length $N - 1$.

```
Signed SRS:
   - + + + - + + - - + - + - - -
Hadamard matrix H:
 + + + + + + + + + + + + + + + +
 + - + + + - + + - - + - + - - -
 + - - + + + - + + - - + - + - -
 + - - - + + + - + + - - + - + -
 + - - - - + + + - + + - - + - +
 + + - - - - + + + - + + - - + -
 + - + - - - - + + + - + + - - +
 + + - + - - - - + + + - + + - -
 + - + - + - - - - + + + - + + -
 + - - + - + - - - - + + + - + +
 + + - - + - + - - - - + + + - +
 + + + - - + - + - - - - + + + -
 + - + + - - + - + - - - - + + +
 + + - + + - - + - + - - - - + +
 + + + - + + - - + - + - - - - +
 + + + + - + + - - + - + - - - -

Signed SRS:
   - + + - + - -
Hadamard matrix H:
 + + + + + + + +
 + - + + + - + -
 + - - + + - + -
 + - - - + + - +
 + + - - - + + -
 + - + - - - + +
 + + - + - - - +
 + + + - + - - -

Signed SRS:
   - + -
Hadamard matrix H:
 + + + +
 + - + -
 + - - +
 + + - -
```

Figure 10.16: Hadamard matrices created with binary shift register sequences (SRS) of maximum length. Only the sign of the entries is given, all entries are ± 1.

- Make S signed, that is, replace all ones by -1 and all zeros by $+1$.
- The $N \times N$ matrix $\mathbf{H}$ is obtained by filling the first row and the first column with ones and filling the remaining entries with cyclical copies of s: For $r = 1, 2, \ldots N-1$ and $c = 1, 2, \ldots N-1$ set $\mathbf{H}_{r,c} := S_{c-r+1 \bmod N-1}$.

For given n we can obtain as many different Hadamard matrices as there are m-sequences.

The matrices in figure 10.16 where produced with the demo [FXT: demo comb/hadamard-srs-demo.cc].

```
#include "bpol/lfsr.h" // class lfsr
#include "aux1/copy.h"  // copy_cyclic()

#include "matrix/matrix.h"  // class matrix
typedef matrix<int>  Smat;  // matrix with integer entries

  [-snip-]
    ulong n = 5;
    ulong N = 1UL << n;
  [-snip-]

    // --- create signed SRS:
    int vec[N-1];
    lfsr sr(n, c);
    for (ulong k=0; k<N-1; ++k)
    {
        ulong x = 1UL & sr.get_a();
        vec[k] = ( x ? -1 : +1 );
        sr.next();
    }

    // --- create Hadamard matrix:
    Smat H(N,N);
    for (ulong c=0; c<N; ++c)  H.set(0, c, +1);  // first row = [1,1,1,...,1]
    for (ulong r=1; r<N; ++r)
    {
        H.set(r, 0, +1);  // first column = [1,1,1,...,1]^T
        copy_cyclic(vec, H.rowp_[r]+1, N-1, N-r);
    }
  [-snip-]
```

The function `copy_cyclic()` is defined in [FXT: file aux1/copy.h]:

```
template <typename Type>
inline void copy_cyclic(const Type *src, Type *dst, ulong n, ulong s)
// Copy array src[] to dst[]
// starting from position s in src[]
// wrap around end of src[]  (src[n-1])
//
// src[] is assumed to be of length n
// dst[] must be length n at least
//
```

```
// Equivalent to:  { copy(src, dst, n); rotate_right(dst, n, s)}
{
    ulong k = 0;
    while ( s<n )  dst[k++] = src[s++];
    s = 0;
    while ( k<n )  dst[k++] = src[s++];
}
```

If we define the matrix $\mathbf{X}$ to be the $(N-1) \times (N-1)$ block of $\mathbf{H}$ obtained by deleting the first row and column then

$$\mathbf{X}\,\mathbf{X}^T \quad = \quad \begin{bmatrix} N-1 & -1 & -1 & \cdots & -1 \\ -1 & N-1 & -1 & \cdots & -1 \\ -1 & -1 & N-1 & \cdots & -1 \\ \vdots & \vdots & \cdots & \ddots & \vdots \\ -1 & -1 & -1 & \cdots & N-1 \end{bmatrix} \tag{10.55}$$

Equivalently, for the (cyclic) auto correlation of S (see section 6.4 on page 222):

$$\sum_{k=0}^{L-1} S_k\, S_{k+\tau \bmod L} \quad = \quad \begin{cases} +L & \text{if} \quad \tau = 0 \\ -1 & \text{else} \end{cases} \tag{10.56}$$

where $L = N - 1$ is the length of the sequence.

A alternative way to obtain Hadamard matrices of dimension 2^n is to use the signs in the multiplication table for hypercomplex numbers described in section 20.6 on page 514.

10.12.2 Hadamard matrices via conference matrices

```
Quadratic characters modulo 13:
   0 + - + + - - - - + + - +
 14x14 conference matrix C:
  0 + + + + + + + + + + + +
  + 0 + - + + - - - - + + - +
  + + 0 + - + + - - - - + + -
  + - + 0 + - + + - - - - + +
  + + - + 0 + - + + - - - - +
  + + + - + 0 + - + + - - - -
  + - + + - + 0 + - + + - - -
  + - - + + - + 0 + - + + - -
  + - - - + + - + 0 + - + + -
  + - - - - + + - + 0 + - + +
  + + - - - - + + - + 0 + - +
  + + + - - - - + + - + 0 + -
  + - + + - - - - + + - + 0 +
  + + - + + - - - - + + - + 0
```

```
Quadratic characters modulo 11:
   0 + - + + + - - - + -
12x12 conference matrix C:
  0 + + + + + + + + + + +
  - 0 + - + + + - - - + -
  - - 0 + - + + + - - - +
  - + - 0 + - + + + - - -
  - - + - 0 + - + + + - -
  - - - + - 0 + - + + + -
  - - - - + - 0 + - + + +
  - + - - - + - 0 + - + +
  - + + - - - + - 0 + - +
  - + + + - - - + - 0 + -
  - + - + + + - - - + - 0
```

Figure 10.17: Two Conference matrices, the entries not on the diagonal are ± 1 and only the sign is given. The left is a symmetric 14×14 matrix ($13 \equiv 1 \bmod 4$), the right is a antisymmetric 12×12 matrix ($11 \equiv 3 \bmod 4$). Replacing all diagonal elements of the right matrix with $+1$ gives a 12×12 Hadamard matrix.

A *conference matrix* $\mathbf{C}_Q$ is a $Q \times Q$ matrix with zero diagonal and all other entries ± 1 so that

$$\mathbf{C}\,\mathbf{C}^T \quad = \quad (Q-1)\ \mathrm{id} \tag{10.57}$$

An algorithm for the construction of $\mathbf{C}_q$ for $Q = q + 1$ where q is an odd prime:

- Create a length-q array S with entries $S_k \in \{-1,\, 0,\, +1\}$ as follows: Set $S_0 = 0$ and, for $1 \le k < q$ set $S_k = +1$ if k is a square modulo q, $S_k = -1$ else.
- Set $y = 1$ if $q \equiv 1 \bmod 4$, else $y = -1$ (then $q \equiv 3 \bmod 4$).

- Create a $Q \times Q$ matrix $\mathbf{C}$ as follows: Set $\mathbf{C}_{0,0} = 0$ and $\mathbf{C}_{0,k} = +1$ for $1 \le k < Q$ (first row). Set $\mathbf{C}_{k,0} = y$ for $1 \le k < Q$ (first column). Fill the remaining entries with cyclical copies of S: For $1 \le r < q$ and $1 \le c < q$ set $\mathbf{C}_{r,c} = S_{c-r+1 \bmod q}$.

The quantity y tells us whether $\mathbf{C}$ is symmetric ($y = +1$) or antisymmetric ($y = -1$).

If $\mathbf{C}_Q$ is antisymmetric then $H_Q = \mathbf{C}_Q + \mathrm{id}$ is a Hadamard matrix. For example, replacing all zeros in the 12×12 matrix in figure 10.17 on the facing page by $+1$ gives a 12×12 Hadamard matrix..

```
12x12 Hadamard matrix H:
 + + + + + + - + + + + +
 + + + - - + + - + - - +
 + + + + - - + + - + - -
 + - + + + - + - + - + -
 + - - + + + + - - + - +
 + + - - + + + + - - + -
 - + + + + + - - - - - -
 + - + - - + - - - + + -
 + + - + - - - - - - + +
 + - + - + - - + - - - +
 + - - + - + - + + - - -
 + + - - + - - - + + - -

Quadratic characters modulo 5:
   0 + - - +
6x6 conference matrix C:
 0 + + + + +
 + 0 + - - +
 + + 0 + - -
 + - + 0 + -
 + - - + 0 +
 + + - - + 0
```

Figure 10.18: A 12×12 Hadamard matrix (left) created from the symmetric 6×6 conference matrix on the right.

If $\mathbf{C}_Q$ is symmetric then a $2Q \times 2Q$ Hadamard matrix is given by

$$\mathbf{H}_{2Q} \quad := \quad \begin{bmatrix} +\mathrm{id} + \mathbf{C} & -\mathrm{id} + \mathbf{C} \\ -\mathrm{id} + \mathbf{C} & -\mathrm{id} - \mathbf{C} \end{bmatrix} \tag{10.58}$$

Figure 10.18 shows a 12×12 Hadamard matrix that was created using this formula.

The demo [FXT: demo comb/conference-quadres-demo.cc] outputs for a given q the $Q \times Q$ conference matrix and the corresponding Hadamard matrix:

```
#include "mod/numtheory.h" // kronecker()
#include "matrix/matrix.h"  // class matrix
#include "aux1/copy.h"  // copy_cyclic()

  [-snip-]
    int y = ( 1==q%4 ? +1 : -1 );
    ulong Q = q+1;
  [-snip-]
    // --- create table of quadratic characters modulo q:
    int vec[q];  fill<int>(vec, q, -1);  vec[0] = 0;
    for (ulong k=1; k<(q+1)/2; ++k)  vec[(k*k)%q] = +1;
  [-snip-]
    // --- create Q x Q conference matrix:
    Smat C(Q,Q);
    C.set(0,0, 0);
    for (ulong c=1; c<Q; ++c)  C.set(0, c, +1);  // first row = [1,1,1,...,1]
    for (ulong r=1; r<Q; ++r)
    {
        C.set(r, 0, y);  // first column = +-[1,1,1,...,1]^T
        copy_cyclic(vec, C.rowp_[r]+1, q, Q-r);
    }
  [-snip-]
    // --- create a N x N conference matrix:
    ulong N = ( y<0 ? Q : 2*Q );
    Smat H(N,N);
    if ( N==Q )
    {
        copy(C, H);
        H.diag_add_val(1);
    }
    else
    {
        Smat K2(2,2);  K2.fill(+1); K2.set(1,1, -1);  // K2 = [+1,+1; +1,-1]
        H.kronecker(K2, C);  // Kronecker product of matrices
        for (ulong k=0; k<Q; ++k)  // adjust diagonal of submatrices
```

```
        {
            ulong r, c;
            r=k;   c=k;   H.set(r,c, H.get(r,c)+1);
            r=k;   c=k+Q; H.set(r,c, H.get(r,c)-1);
            r=k+Q; c=k;   H.set(r,c, H.get(r,c)-1);
            r=k+Q; c=k+Q; H.set(r,c, H.get(r,c)-1);
        }
    }
  [-snip-]
```

If both $\mathbf{H}_a$ and $\mathbf{H}_b$ are Hadamard matrices then their Kronecker product $\mathbf{H}_{ab} = \mathbf{H}_a \otimes \mathbf{H}_b$ is again a Hadamard matrix.

10.12.3 Conference matrices via finite fields

The algorithm for odd primes q can be modified to work also for powers of odd primes. Then one has to work with the finite fields $GF(q^n)$. The entries $\mathbf{C}_{r+1,c+1}$ for $r = 0, 1, \ldots, q^n - 1$ and $c = 0, 1, \ldots, q^n - 1$ have to be the quadratic character of $z_r - z_c$ where $z_0, z_1, \ldots, z_{q^n-1}$ are the elements in $GF(q^n)$ in some (fixed) order.

We implement the algorithm in pari/gp. Firstly, we give two simple routines that map the elements $z_i \in GF(q^n)$ (represented as polynomials modulo q) to the numbers $0, 1, \ldots, q^n - 1$. The polynomial $p(x) = c_0 + c_1\, x + \ldots + c_{n-1}\, x^{n-1}$ is mapped to $N = c_0 + c_1\, q + \ldots + c_{n-1}\, q^{n-1}$.

```
pol2num(p,q)=
\\ Return number for polynomial p.
{
    local(t, n);  n=0;
    for (k=0, poldegree(p),
        t = polcoeff(p, k);
        t *= Mod(1,q);
        t = component(t, 2);
        t *= q^k;
        n += t;
    );
    return( n );
}
```

The inverse routine is

```
num2pol(n,q)=
\\ Return polynomial for number n.
{
    local(p, mq, k);
    p = Pol(0,'x);
    k = 0;
    while ( 0!=n,
        mq = n % q;
        p += mq * ('x)^k;
        n -= mq;
        n \= q;
        k++;
    );
    return( p );
}
```

The quadratic character of an element z can be determined by computing $z^{(q^n-1)/2}$ modulo the field polynomial. The result will be zero for $z = 0$, else ± 1. The following routine determines the character of the difference of two elements as required for the computation of conference matrices:

```
getquadchar_n(n1, n2, q, fp, n)=
\\ Return the quadratic character of (n2-n1) in GF(q^n)
\\ Powering method
{
    local(p1, p2, d, nd, sc);
    if ( n1==n2, return(0) );
    p1 = num2pol(n1, q);
    p2 = num2pol(n2, q);
    d = Mod(1,q)* (p2-p1);
```

```
    d = Mod(d,fp)^((q^n-1)/2);
    d = component(d, 2);
    if ( Mod(1,q)==d,  sc=+1, sc=-1 );
    return( sc );
}
```

The input are two numbers that are mapped to the corresponding field elements.

In order to reduce the computational work we create a table of the quadratic characters for later lookup:

```
quadcharvec(fp, q)=
\\ Return a table of quadratic characters in in GF(q^n)
\\ fp is the field polynomial.
{
    local(n, qn, sv, pl);
    n=poldegree(fp);
    qn=q^n-1;
    sv=vector(qn+1, j, -1);
    sv[1] = 0;
    for (k=1, qn,
        pl = num2pol(k,q);
        pl = Mod(Mod(1,q)*pl, fp);
        sq = pl * pl;
        sq = component(sq, 2);
        i = pol2num( sq, q );
        sv[i+1] = +1;
    );
    return( sv );
}
```

Using this table we can compute the quadratic characters of the difference of two elements in a more efficient manner:

```
getquadchar_v(n1, n2, q, fp, sv)=
\\ Return the quadratic character of (n2-n1) in GF(q^n)
\\ Table lookup method
{
    local(p1, p2, d, nd, sc);
    if ( n1==n2, return(0) );
    p1 = num2pol(n1, q);
    p2 = num2pol(n2, q);
    d = (p2-p1) % fp;
    nd = pol2num(d, q);
    sc = sv[nd+1];
    return( sc );
}
```

Now we can compute conference matrices:

```
matconference(q, fp, sv)=
\\ Return a QxQ conference matrix.
\\ q  an odd prime.
\\ fp an irreducible polynomial modulo q.
\\ sv table of quadratic characters in GF(q^n)
\\   where n is the degree of fp.
{
    local(y, Q, C, n);
    n = poldegree(fp);
    Q=q^n+1;
    C = matrix(Q,Q);
    for (k=2, Q, C[1,k]=+1);  \\ first row
    for (k=2, Q, C[k,1]=y);  \\ first column
    for (r=2, Q,
        for (c=2, Q,
            sc = getquadchar_n(r-2, c-2, q, fp, n);
\\            sc = getquadchar_v(r-2, c-2, q, fp, sv);  \\ same result
            C[r,c] = sc;
        );
    );
    return( C );
}
```

To compute a $Q \times Q$ conference matrix where $Q = q^n$ we need to find an polynomial of degree n that is

irreducible modulo q. With $q = 3$ and the field polynomial $f = x^2 + 1$ we obtain the 10×10 conference matrix shown in figure 10.19 A conference matrix for $q = 3$ and $f = x^3 - x + 1$ is given in figure 10.20.

```
q = 3   fp = x^2 + 1   GF(3^2)
Table of quadratic characters:
   0 + + + - - + - -
10x10 conference matrix C:
 0 + + + + + + + + +
 - 0 + + + - - + - -
 - + 0 + - + - - + -
 - + + 0 - - + - - +
 - + - - 0 + + + - -
 - - + - + 0 + - + -
 - - - + + + 0 - - +
 - + - - + - - 0 + +
 - - + - - + - + 0 +
 - - - + - - + + + 0
```

Figure 10.19: A 10×10 conference matrix for $q = 3$ and the field polynomial $f = x^2 + 1$.

Hadamard matrices can be created in the same manner as before, the symmetry criterion being whether $q^n \equiv \pm 1 \bmod 4$.

```
q = 3   fp = x^3 - x + 1   GF(3^3)
Table of quadratic characters:
   0 + - - - - + + + + - + + + - + + - - - + - + - - + -
28x28 conference matrix C:
 0 + + + + + + + + + + + + + + + + + + + + + + + + + + +
 - 0 + - - - - + + + + - + + + - + + - - - + - + - - + -
 - - 0 + - - - + + + + + - - + + - + + + - - - - + - - +
 - + - 0 - - - + + + - + + + - + + - + - + - + - - + - -
 - + + + 0 + - - - - + + - + - + + + - - + - - - + - + -
 - + + + - 0 + - - - - + + + + - - + + - - + + - - - - +
 - + + + + - 0 - - - + - + - + + + - + + - - - + - + - -
 - - - - + + + 0 + - + + - + + - + - + - + - - + - - - +
 - - - - + + + - 0 + - + + - + + + + - - - + - - + + - -
 - - - - + + + + - 0 + - + + - + - + + + - - + - - - + -
 - - - + - + - - + - 0 + - - - - + + + + - + + + - + + -
 - + - - - - + - - + - 0 + - - - + + + + + - - + + - + +
 - - + - + - - + - - + - 0 - - - + + + - + + + - + + - +
 - - + - - - + - + - + + + 0 + - - - - + + - + - + + + -
 - - - + + - - - - + + + + - 0 + - - - - + + + + - - + +
 - + - - - + - + - - + + + + - 0 - - - + - + - + + + - +
 - - + - - + - - - + - - - + + + 0 + - + + - + + - + - +
 - - - + - - + + - - - - - + + + - 0 + - + + - + + + + -
 - + - - + - - - + - - - - + + + + - 0 + - + + - + - + +
 - + - + + + - + + - - - + - + - - + - 0 + - - - - + + +
 - + + - - + + - + + + - - - - + - - + - 0 + - - - + + +
 - - + + + - + + - + - + - + - - + - - + - 0 - - - + + +
 - + + - + - + + + - - + - - - + - + - + + + 0 + - - - -
 - - + + + + - - + + - - + + - - - - + + + + - 0 + - - -
 - + - + - + + + - + + - - - + - + - - + + + + - 0 - - -
 - + + - + + - + - + - + - - + - - - + - - - + + + 0 + -
 - - + + - + + + + - - - + - - + + - - - - - + + + - 0 +
 - + - + + - + - + + + - - + - - - + - - - - + + + + - 0
```

Figure 10.20: A 28×28 conference matrix for $q = 3$ and the field polynomial $f = x^3 - x + 1$.

The construction of Hadamard matrices via conference matrices is due to R.E.A.C.Paley. The conference matrices obtained are of size $c = q^n + 1$ where q is an odd prime. The values $c \leq 100$ are:

```
4, 6, 8, 10, 12, 14, 18, 20, 24, 26, 28, 30, 32, 38, 42, 44, 48,
50, 54, 60, 62, 68, 72, 74, 80, 82, 84, 90, 98
```

We do not obtain conference matrices for any odd c and these even values $c \leq 100$:

```
2, 16, 22, 34, 36, 40, 46, 52, 56, 58, 64, 66, 70, 76, 78, 86, 88, 92, 94, 96, 100
```

For example, $c = 16 = 15 + 1 = 3 \cdot 5 + 1$ has not the required form.

If a conference matrix of size c exists then we can create Hadamard matrices of sizes $N = c$ whenever $q^n \equiv 3 \bmod 4$ and $2\,c$ whenever $q^n \equiv 1 \bmod 4$. Further, if Hadamard matrices of sizes N and M exist then a $N \cdot M \times N \cdot M$ Hadamard matrix can be obtained via the Kronecker product.

The values of $N = 4\,k \leq 2000$ such that this construction does *not* give a $N \times N$ Hadamard matrix are:

```
    92, 116, 156, 172, 184, 188, 232, 236, 260, 268, 292, 324, 356, 372,
    376, 404, 412, 428, 436, 452, 472, 476, 508, 520, 532, 536, 584,
    596, 604, 612, 652, 668, 712, 716, 732, 756, 764, 772, 808, 836,
    852, 856, 872, 876, 892, 904, 932, 940, 944, 952, 956, 964, 980,
    988, 996, 1004, 1012, 1016, 1028, 1036, 1068, 1072, 1076, 1100,
    1108, 1132, 1148, 1168, 1180, 1192, 1196, 1208, 1212, 1220, 1244,
    1268, 1276, 1300, 1316, 1336, 1340, 1364, 1372, 1380, 1388, 1396,
    1412, 1432, 1436, 1444, 1464, 1476, 1492, 1508, 1528, 1556, 1564,
    1588, 1604, 1612, 1616, 1636, 1652, 1672, 1676, 1692, 1704, 1712,
    1732, 1740, 1744, 1752, 1772, 1780, 1796, 1804, 1808, 1820, 1828,
    1836, 1844, 1852, 1864, 1888, 1892, 1900, 1912, 1916, 1928, 1940,
    1948, 1960, 1964, 1972, 1976, 1992
```

This is sequence A046116 of [OEIS]. It can be obtained by starting with all numbers of the form $4\,k$ and deleting all values $k = 2^a\,(q+1)$ where q is a power of an odd prime. In pari/gp:

```
{ forprime(p=3,n,
    f=1;
    while ( 1,
        q=1;
        while ( 1,
            t = (q+1)*f;
            if ( t>n, break());
            v[t] = 1;
            q *= p;
        );
        f *= 2;
        if ( f>n, break() );
    );
 ); }
forstep (j=4, n, 4, if ( v[j]==0, print1(j, ", ") ) );
```

Constructions for Hadamard matrices for numbers of certain forms are known, see [CComb]. Whether Hadamard matrices exist for all values $N = 4\,k$ is an open problem.

A readable source about constructions for Hadamard matrices is [98].

10.13 Eigenvectors of the Walsh transform *

```
 0:  [ +5 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 ]
 1:  [ +1 +3 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 ]
 2:  [ +1 +1 +3 -1 +1 +1 -1 -1 +1 +1 -1 -1 +1 +1 -1 -1 ]
 3:  [ +1 -1 -1 +5 +1 -1 -1 +1 +1 -1 -1 +1 +1 -1 -1 +1 ]
 4:  [ +1 +1 +1 +1 +3 -1 -1 -1 +1 +1 +1 +1 -1 -1 -1 -1 ]
 5:  [ +1 -1 +1 -1 -1 +5 -1 +1 +1 -1 +1 -1 -1 +1 -1 +1 ]
 6:  [ +1 +1 -1 -1 -1 -1 +5 +1 +1 +1 -1 -1 -1 -1 +1 +1 ]
 7:  [ +1 -1 -1 +1 -1 +1 +1 +3 +1 -1 -1 +1 -1 +1 +1 -1 ]
 8:  [ +1 +1 +1 +1 +1 +1 +1 +1 -5 -1 -1 -1 -1 -1 -1 -1 ]
 9:  [ +1 -1 +1 -1 +1 -1 +1 -1 -1 -3 -1 +1 -1 +1 -1 +1 ]
10:  [ +1 +1 -1 -1 +1 +1 -1 -1 -1 -1 -3 +1 -1 -1 +1 +1 ]
11:  [ +1 -1 -1 +1 +1 -1 -1 +1 -1 +1 +1 -5 -1 +1 +1 -1 ]
12:  [ +1 +1 +1 +1 -1 -1 -1 -1 -1 -1 -1 -1 -3 +1 +1 +1 ]
13:  [ +1 -1 +1 -1 -1 +1 -1 +1 -1 +1 -1 +1 +1 -5 +1 -1 ]
14:  [ +1 +1 -1 -1 -1 -1 +1 +1 -1 -1 +1 +1 +1 +1 -5 -1 ]
15:  [ +1 -1 -1 +1 -1 +1 +1 -1 -1 +1 +1 -1 +1 -1 -1 -3 ]
```

Figure 10.21: Eigenvectors of the length-16 Walsh transform (Walsh-Kronecker basis) as row vectors. The eigenvalues are $+1$ for the vectors $0 \ldots 7$ and -1 for the vectors $8 \ldots 16$. Linear combinations of vectors with the same eigenvalue e are again eigenvectors with eigenvalue e.

The Walsh transforms are self-inverse, so their eigenvalues can only be plus or minus one. Let a be a sequence and let $W(.)$ denote the Walsh transform. Set

$$u_+ \quad := \quad W(a) + a \tag{10.59}$$

Then

$$W(u_+) \quad = \quad W(W(a)) + W(a) \;=\; a + W(a) \;=\; +1 \cdot u_+ \tag{10.60}$$

That is, u_+ is an eigenvector of W with eigenvalue $+1$. Equivalently, $u_- := W(a) - a$ is an eigenvector with eigenvalue -1. Thereby, two eigenvectors can be obtained from an arbitrary nonzero sequence.

We are interested in a simple routine that for a Walsh transform of length n gives a set of n eigenvectors that span the n-dimensional space. With a routine that computes the k-th basis function of the transform we can obtain an eigenvector efficiently by simply adding a delta peak at position k to the basis function. The delta peak has to be scaled according to whether a positive or negative eigenvalue is desired and according to the normalization of the transform.

A suitable routine for the Walsh-Kronecker basis (whose basis functions are given in figure 10.1 on page 304) is

```
void
walsh_wak_eigen(double *v, ulong ldn, ulong k)
// Eigenvectors of the Walsh transform (walsh_wak).
// Eigenvalues are +1 if k<n/2, else -1
{
    ulong n = 1UL << ldn;
    walsh_wak_basefunc(v, n, k);
    double d = sqrt(n);
    v[k] += (k<n/2 ? +d : -d);
}
```

This routine is given in [FXT: file walsh/walsheigen.cc]. Figure 10.21 on the previous page was created with the program [FXT: demo fft/walsh-eigenvec-demo.cc].

Note that with the unnormalized transforms the eigenvalues are $\pm\sqrt{n}$.

Chapter 11

The Haar transform

Haar transforms are invertible transforms that do not involve trigonometric factors. We present several variants of the transform whose computation involves just proportional n operations.

Finally, Haar transforms can be used as building blocks of the Walsh transform.

11.1 The 'standard' Haar transform

The basis functions for the 'standard' Haar transform are shown in figure 11.1 on the following page. The corresponding matrix is sparse: only $\sim n \log_2(n)$ entries are nonzero. A C++ implementation of the Haar transform is [FXT: `haar()` in haar/haar.h]:

```
template <typename Type>
void haar(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type s2 = sqrt(0.5);
    Type v = 1.0;
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    for (ulong m=n; m>1; m>>=1)
    {
        v *= s2;
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[j];
            Type y = f[j+1];
            g[k]    =  x + y;
            g[mh+k] = (x - y) * v;
        }
        copy(g, f, m);
    }
    f[0] *= v; // v == 1.0/sqrt(n);
    if ( !ws )  delete [] g;
}
```

The above routine uses a temporary workspace that can be supplied by the caller. The computational cost is only $\sim n$. Code for the inverse Haar transform [FXT: `inverse_haar()` in haar/haar.h]:

```
template <typename Type>
void inverse_haar(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type s2 = sqrt(2.0);
    Type v = 1.0/sqrt(n);
    Type *g = ws;
    if ( !ws )  g = new Type[n];
```

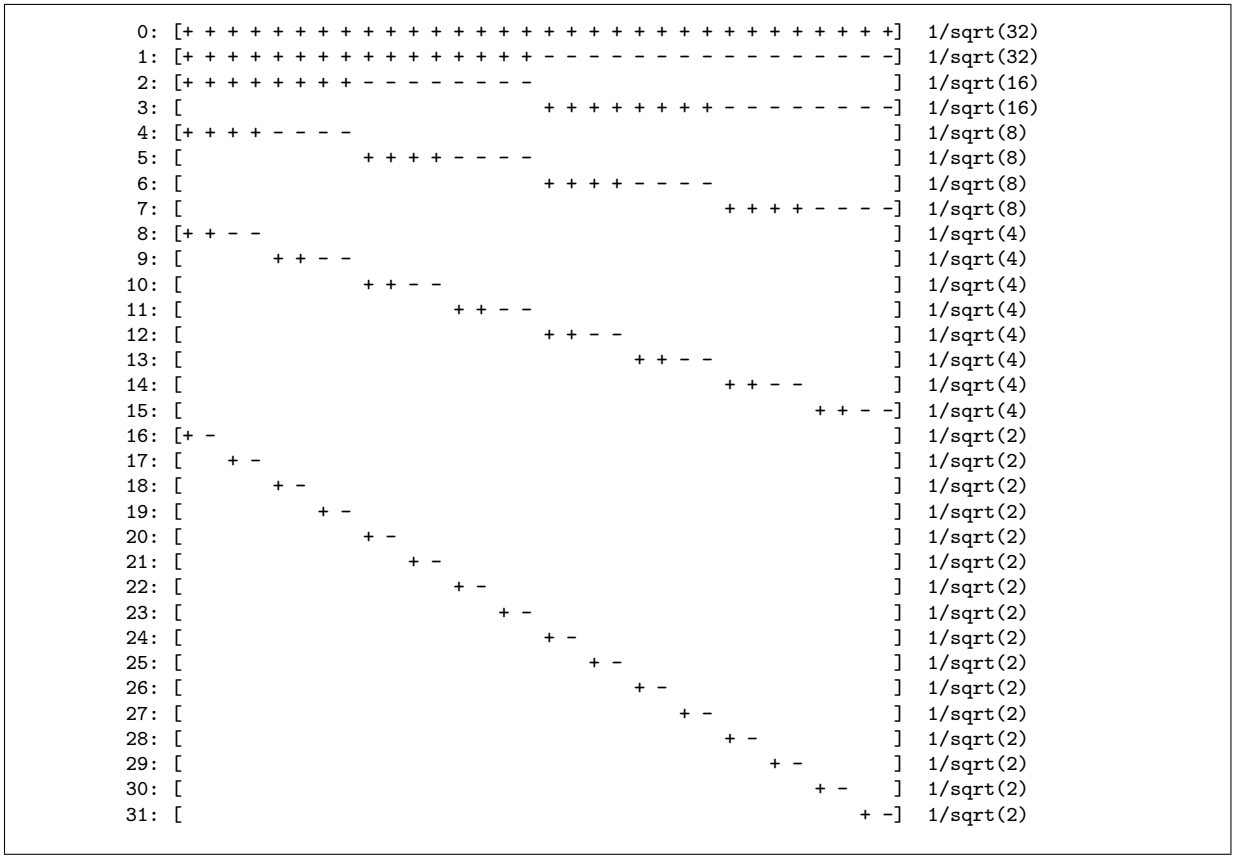


Figure 11.1: Basis functions for the Haar transform. Only the sign of the basis functions is shown. The absolute value of the nonzero entries in each row is given at the right. The norm of each row is one. At the blank entries the functions are zero.

```
    f[0] *= v;
    for (ulong m=2; m<=n; m<<=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[k];
            Type y = f[mh+k] * v;
            g[j]    =  x + y;
            g[j+1]  =  x - y;
        }
        copy(g, f, m);
        v *= s2;
    }
    if ( !ws )  delete [] g;
}
```

That the given routines use a temporary storage may be seen as a disadvantage. A rather simple reordering of the basis functions, however, allows for to an in-place algorithm. This leads to the in-place Haar transform.

11.2 In-place Haar transform

Code for a Haar transform that can be computed in-place [FXT: `haar_inplace()` in haar/haar.h]:

```
template <typename Type>
```

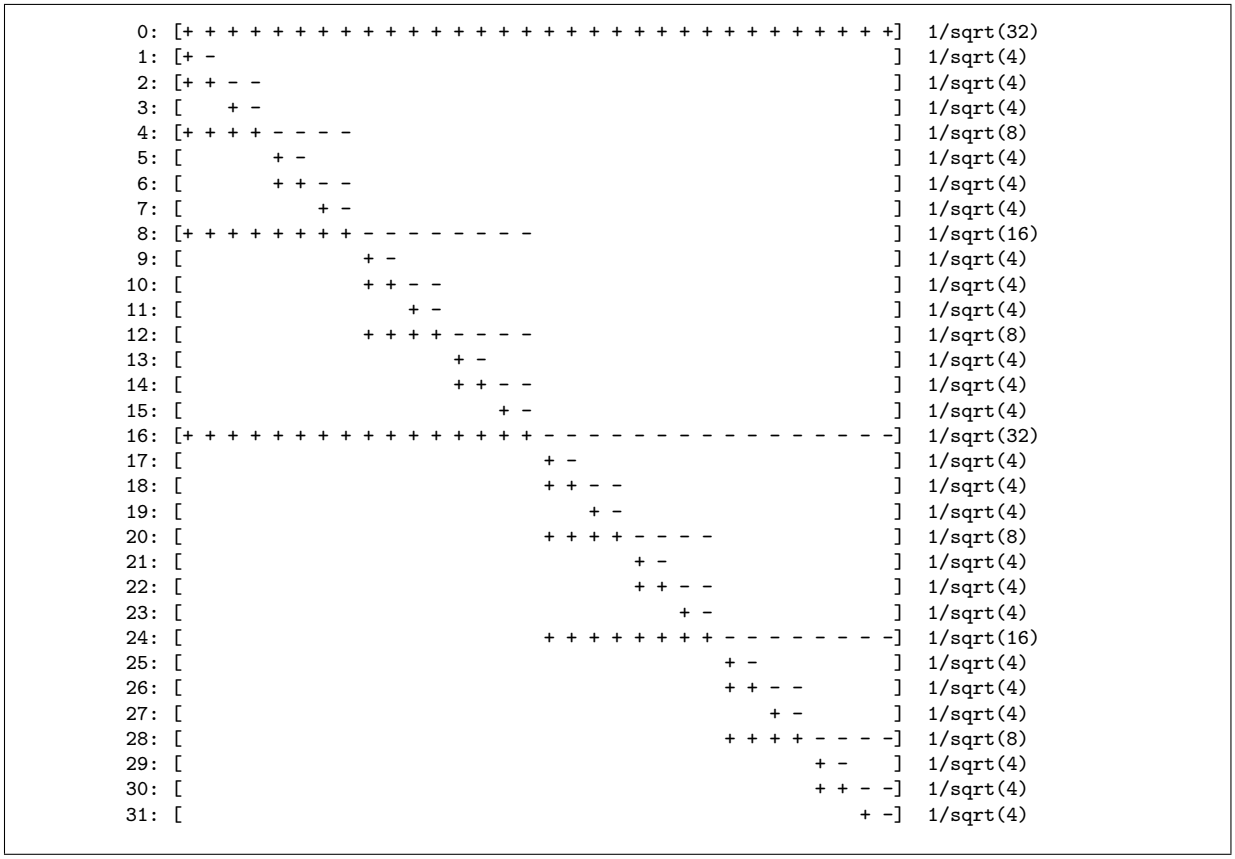


Figure 11.2: Haar basis functions, in-place order. Only the sign of the basis functions is shown. The absolute value of the nonzero entries in each row is given at the right. The norm of each row is one. At the blank entries the functions are zero.

```
void haar_inplace(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    Type s2 = sqrt(0.5);
    Type v = 1.0;
    for (ulong js=2; js<=n; js<<=1)
    {
        v *= s2;
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t];
            f[j]  =  x + y;
            f[t]  = (x - y) * v;
        }
    }
    f[0] *= v; // v==1.0/sqrt(n);
}
```

The inverse is [FXT: `inverse_haar_inplace()` in haar/haar.h]:

```
template <typename Type>
void inverse_haar_inplace(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    Type s2 = sqrt(2.0);
    Type v = 1.0/sqrt(n);
    f[0] *= v;
    for (ulong js=n; js>=2; js>>=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
```

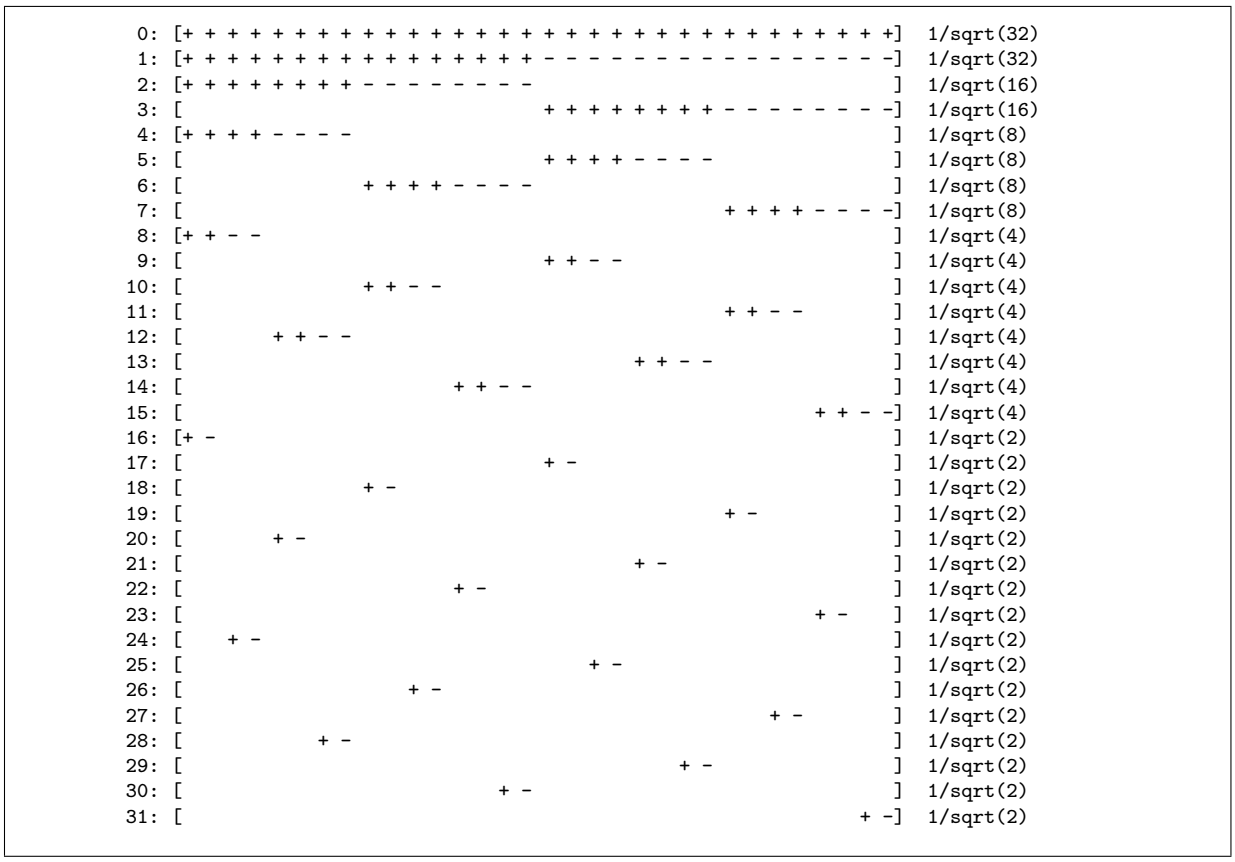


Figure 11.3: Basis functions of the in-place order Haar transform followed by a revbin permutation. The ordering is such that basis functions that are identical up to a shift appear consecutively.

```
        {
            Type x = f[j];
            Type y = f[t] * v;
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
        v *= s2;
    }
}
```

The in-place Haar transform H_i is related to the 'usual' Haar transform H by a permutation P_H via the relations

$$H = P_H \cdot H_i \tag{11.1}$$

$$H^{-1} = H_i^{-1} \cdot P_H^{-1} \tag{11.2}$$

The permutation P_H can be programmed as

```
template <typename Type>
void haar_permute(Type *f, ulong n)
{
    revbin_permute(f, n);
    for (ulong m=4; m<=n/2; m*=2)  revbin_permute(f+m, m);
}
```

The revbin permutations in the loop do not overlap, so the inverse Haar permutation is obtained by simply swapping the loop with the full-length revbin permutation [FXT: file perm/haarpermute.h]:

```
template <typename Type>
void inverse_haar_permute(Type *f, ulong n)
{
```

```
    for (ulong m=4; m<=n/2; m*=2)  revbin_permute(f+m, m);
    revbin_permute(f, n);
}
```

Then, as given above, `haar` is equivalent to

```
    haar_inplace();  haar_permute();
```

and `inverse_haar` is equivalent to

```
    inverse_haar_permute();  inverse_haar_inplace();
```

11.3 Non-normalized Haar transforms

Versions of the Haar transform without normalization are given in [FXT: file haar/haarnn.h]. The basis functions are the same as for the normalized versions, only the absolute value of the nonzero entries are different.

```
template <typename Type>
void haar_nn(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    for (ulong m=n; m>1; m>>=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[j];
            Type y = f[j+1];
            g[k]    =  x + y;
            g[mh+k] =  x - y;
        }
        copy(g, f, m);
    }
    if ( !ws )  delete [] g;
}
```

The inverse is

```
template <typename Type>
void inverse_haar_nn(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type s2 = 2.0;
    Type v = 1.0/n;
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    f[0] *= v;
    for (ulong m=2; m<=n; m<<=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[k];
            Type y = f[mh+k] * v;
            g[j]    =  x + y;
            g[j+1]  =  x - y;
        }
        copy(g, f, m);
        v *= s2;
    }
    if ( !ws )  delete [] g;
}
```

An unnormalized transform that works in-place is

```
template <typename Type>
void haar_inplace_nn(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
```

```
    for (ulong js=2; js<=n; js<<=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t];
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
    }
}
```

The inverse routine is

```
template <typename Type>
void inverse_haar_inplace_nn(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    Type s2 = 2.0;
    Type v = 1.0/n;
    f[0] *= v;
    for (ulong js=n; js>=2; js>>=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t] * v;
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
        v *= s2;
    }
}
```

The sequence of statements `{ haar_inplace_nn(); haar_permute(); }` is equivalent to `{ haar_nn(); }`. The sequence `{ inverse_haar_permute(); inverse_haar_inplace(); }` is equivalent to `{ inverse_haar(); }`.

11.4 Transposed Haar transforms

Figure 11.4 on the next page shows the bases functions of the transposed Haar transform. The shown routine are given in [FXT: file haar/transposedhaarnn.h]. The following routine does an unnormalized Haar transform. The result is, up to normalization, the same as with `inverse_haar()`. The implementation uses a scratch array:

```
template <typename Type>
void transposed_haar_nn(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    for (ulong m=2; m<=n; m<<=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[k];
            Type y = f[mh+k];
            g[j]    =  x + y;
            g[j+1]  =  x - y;
        }
        copy(g, f, m);
    }
    if ( !ws )  delete [] g;
}
```

The inverse transform is

```
template <typename Type>
void inverse_transposed_haar_nn(Type *f, ulong ldn, Type *ws=0)
```

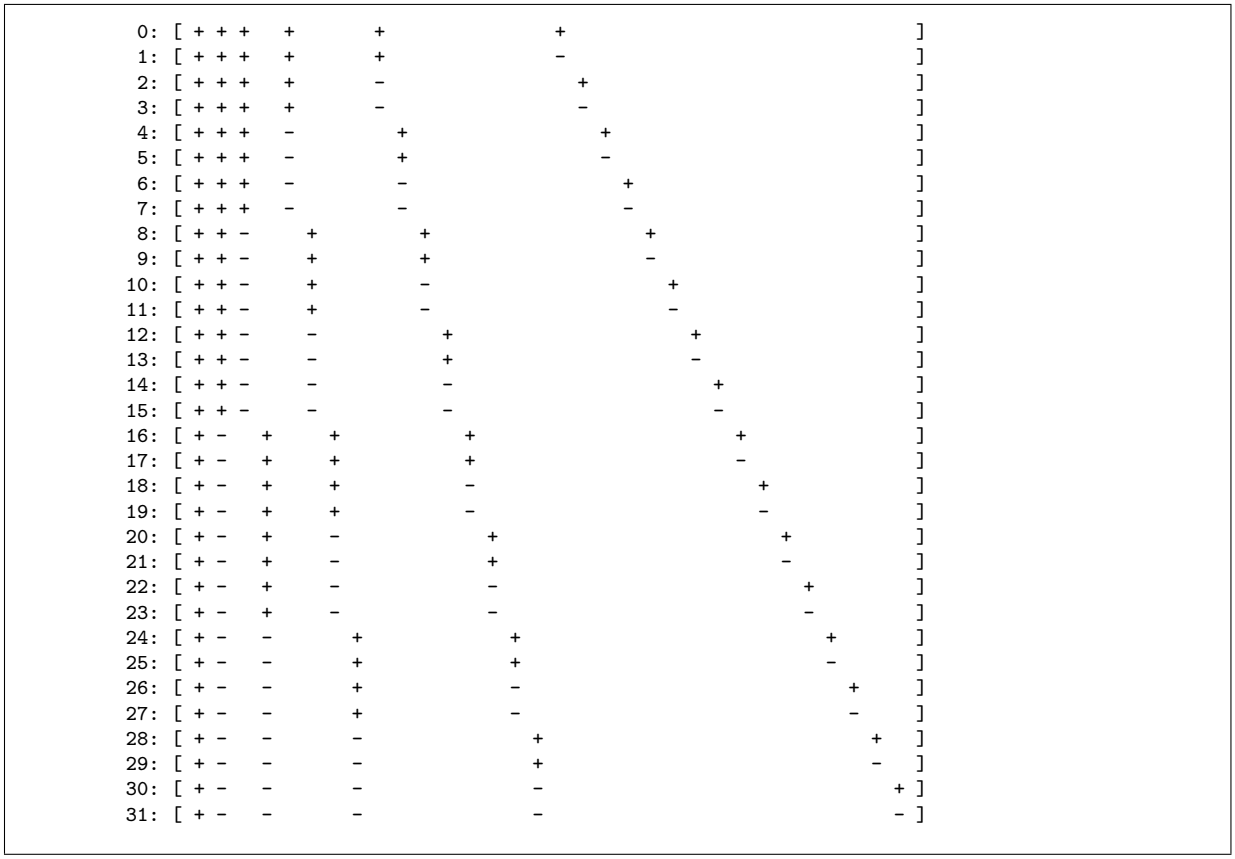


Figure 11.4: Basis functions for the transposed Haar transform. Only the sign of the basis functions is shown. At the blank entries the functions are zero.

```
{
    ulong n = (1UL<<ldn);
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    for (ulong m=n; m>1; m>>=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[j]   * 0.5;
            Type y = f[j+1] * 0.5;
            g[k]    =  x + y;
            g[mh+k] =  x - y;
        }
        copy(g, f, m);
    }
    if ( !ws )  delete [] g;
}
```

The next routine is equivalent to the sequence of statements `{ inverse_haar_permute(); transposed_haar_inplace_nn(); }`. Its advantage is that no scratch array is needed:

```
template <typename Type>
void transposed_haar_inplace_nn(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong js=n; js>=2; js>>=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t];
```

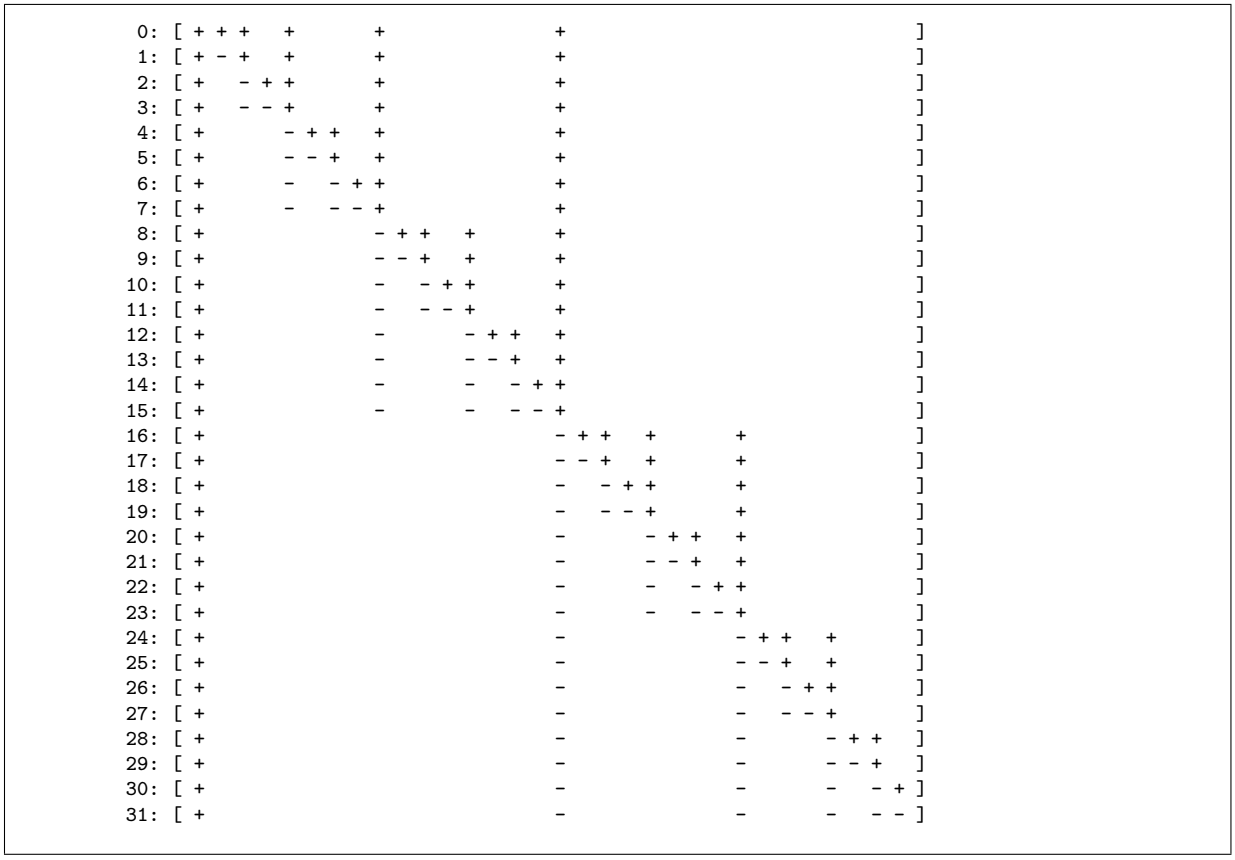


Figure 11.5: Basis functions for the transposed in-place Haar transform. Only the sign of the basis functions is shown. At the blank entries the functions are zero.

```
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
    }
}
```

The inverse transform is

```
template <typename Type>
void inverse_transposed_haar_inplace_nn(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong js=2; js<=n; js<<=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j] * 0.5;
            Type y = f[t] * 0.5;
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
    }
}
```

11.5 The reversed Haar transform

Let H_{ni} denote the non-normalized in-place Haar transform (`haar_inplace_nn`), Let H_{tni} denote the transposed non-normalized in-place Haar transform (`transposed_haar_inplace_nn`), R the revbin per-

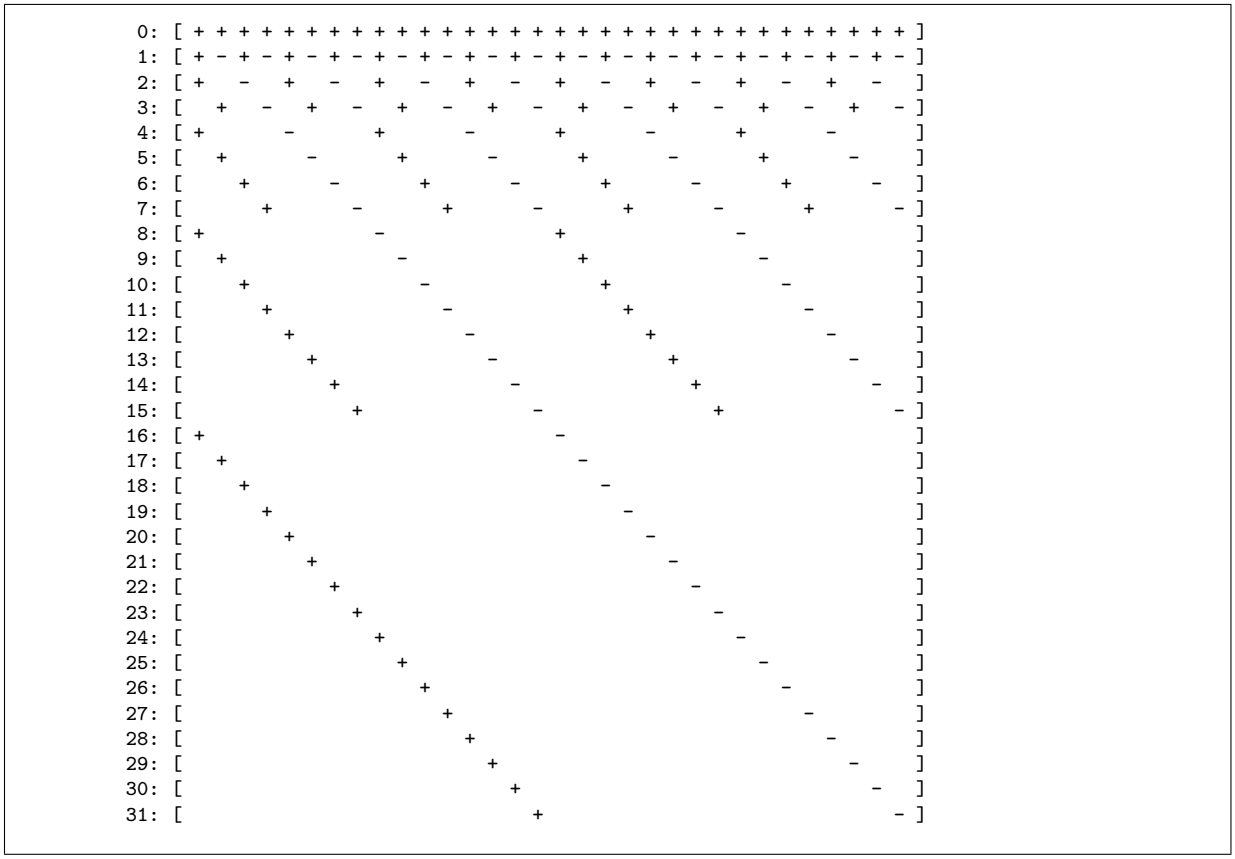


Figure 11.6: Basis functions for the reversed Haar transform. Only the sign of the nonzero entries is shown.

mutation, $\overline{H}$ the reversed Haar transform and $\overline{H}_t$ the transposed reversed Haar transform. Then

$$\overline{H} = R\,H_{ni}\,R \tag{11.3a}$$
$$\overline{H}_t = R\,H_{tni}\,R \tag{11.3b}$$
$$\overline{H}^{-1} = R\,H_{ni}^{-1}\,R \tag{11.3c}$$
$$\overline{H}_t^{-1} = R\,H_{tni}^{-1}\,R \tag{11.3d}$$

Code for the reversed Haar transform [FXT: `haar_rev_nn()` in haar/haarrevnn.h]:

```
template <typename Type>
void haar_rev_nn(Type *f, ulong ldn)
{
//    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
//        for (ulong r=0; r<n; r+=m)  // almost walsh_wak_dif2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
```

```
    }
}
```

Note that this is almost the radix-2 DIF implementation for the Walsh transform The only change is that the line `for (ulong r=0; r<n; r+=m)` was replaced by `ulong r = 0`. The transform can also be computed via the following sequence of statements: `{ revbin_permute(); haar_inplace_nn(); revbin_permute(); }`.

The inverse transform is is obtained by the equivalent modification with the DIT implementation for the Walsh transform and normalization:

```
template <typename Type>
void inverse_haar_rev_nn(Type *f, ulong ldn)
{
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
//        for (ulong r=0; r<n; r+=m)  // almost walsh_wak_dit2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1] * 0.5;
                Type v = f[t2] * 0.5;
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

The reversed transposed Haar transform is, up to normalization, the inverse of `haar_rev_nn()`. It is given in [FXT: file haar/transposedhaarrevnn.h]:

```
template <typename Type>
void transposed_haar_rev_nn(Type *f, ulong ldn)
{
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
//        for (ulong r=0; r<n; r+=m)  // almost walsh_wak_dit2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

The same result would be obtained by the following sequence of statements: `{ revbin_permute(); transposed_haar_inplace_nn(); revbin_permute(); }`. The inverse transform is

```
template <typename Type>
void inverse_transposed_haar_rev_nn(Type *f, ulong ldn)
{
//    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
//        for (ulong r=0; r<n; r+=m)  // almost walsh_wak_dif2()
        {
```

```
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1] * 0.5;
                Type v = f[t2] * 0.5;
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

11.6 Relations between Walsh- and Haar- transforms

Walsh transforms from Haar transforms

```
Haar transforms:
    H(16)                H(8)          H(4)      H(2)
  AAAAAAAAaaaaaaaa      BBBBbbbb       CCcc      Dd
  AAAAaaaa              BBbb           Cc
  AAaa                  Bb
  Aa

Walsh(16) =^= 1*H(16) + 1*H(8) + 2*H(4) + 4*H(2)
  AAAAAAAAaaaaaaaa
  AAAAaaaaBBBBbbbb
  AAaaCCccBBbbCCcc
  AaDdCcDdBbDdCcDd
```

Figure 11.7: Symbolic description of how to build a Walsh transform from Haar transforms.

```
Transposed Haar transforms:
    H(16)                H(8)          H(4)      H(2)
  Aa
  AAaa                  Bb
  AAAAaaaa              BBbb           Cc
  AAAAAAAAaaaaaaaa      BBBBbbbb       CCcc      Dd

Walsh(16) =^= 1*H(16) + 1*H(8) + 2*H(4) + 4*H(2)
  AaDdCcDdBbDdCcDd
  AAaaCCccBBbbCCcc
  AAAAaaaaBBBBbbbb
  AAAAAAAAaaaaaaaa
```

Figure 11.8: Symbolic description of how to build a Walsh transform from Haar transforms, transposed version.

A length-n Walsh transform can be obtained from one length-n Haar transform, one transform of length-$\frac{n}{2}$, two transforms of length-$\frac{n}{4}$, four transforms of length-$\frac{n}{8}$, ... and $\frac{n}{4}$ transforms of length-2. Using the reversed Haar transform the implementation is most straightforward: A Walsh transform (W_k, the one with the Walsh Kronecker base) can be implemented as

```
// algorithm WH1:
ulong n = 1UL<<ldn;
```

```
haar_rev_nn(f, ldn);
for (ulong ldk=ldn-1; ldk>0; --ldk)
{
    ulong k = 1UL << ldk;
    for (ulong j=k; j<n; j+=2*k)  haar_rev_nn(f+j, ldk);
}
```

The idea, as a symbolic scheme, is shown in figure 11.7 on the preceding page. The scheme obtained by reversing the order of the lines is shown in figure 11.8 on the previous page. It corresponds to the computation of W_k using the transposed version of the Haar transform:

```
// algorithm WH1T:
ulong n = 1UL<<ldn;
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    for (ulong j=k; j<n; j+=2*k)  transposed_haar_rev_nn(f+j, ldk);
}
transposed_haar_rev_nn(f, ldn);
```



Figure 11.9: Symbolic scheme of the four versions of the computation of the Walsh transform via Haar transforms.

Two more methods are obtained by reversing the individual lines of the schemes seen so far, see figure 11.9. These correspond to the computation of the inverse Walsh transform ($W_k^{-1} = \frac{1}{n} W_k$) either as

```
// algorithm WH2T:
ulong n = 1UL<<ldn;
inverse_transposed_haar_rev_nn(f, ldn);
for (ulong ldk=ldn-1; ldk>0; --ldk)
{
    ulong k = 1UL << ldk;
    for (ulong j=k; j<n; j+=2*k)  inverse_transposed_haar_rev_nn(f+j, ldk);
}
```

or as

```
// algorithm WH2:
ulong n = 1UL<<ldn;
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    for (ulong j=k; j<n; j+=2*k)  inverse_haar_rev_nn(f+j, ldk);
}
inverse_haar_rev_nn(f, ldn);
```

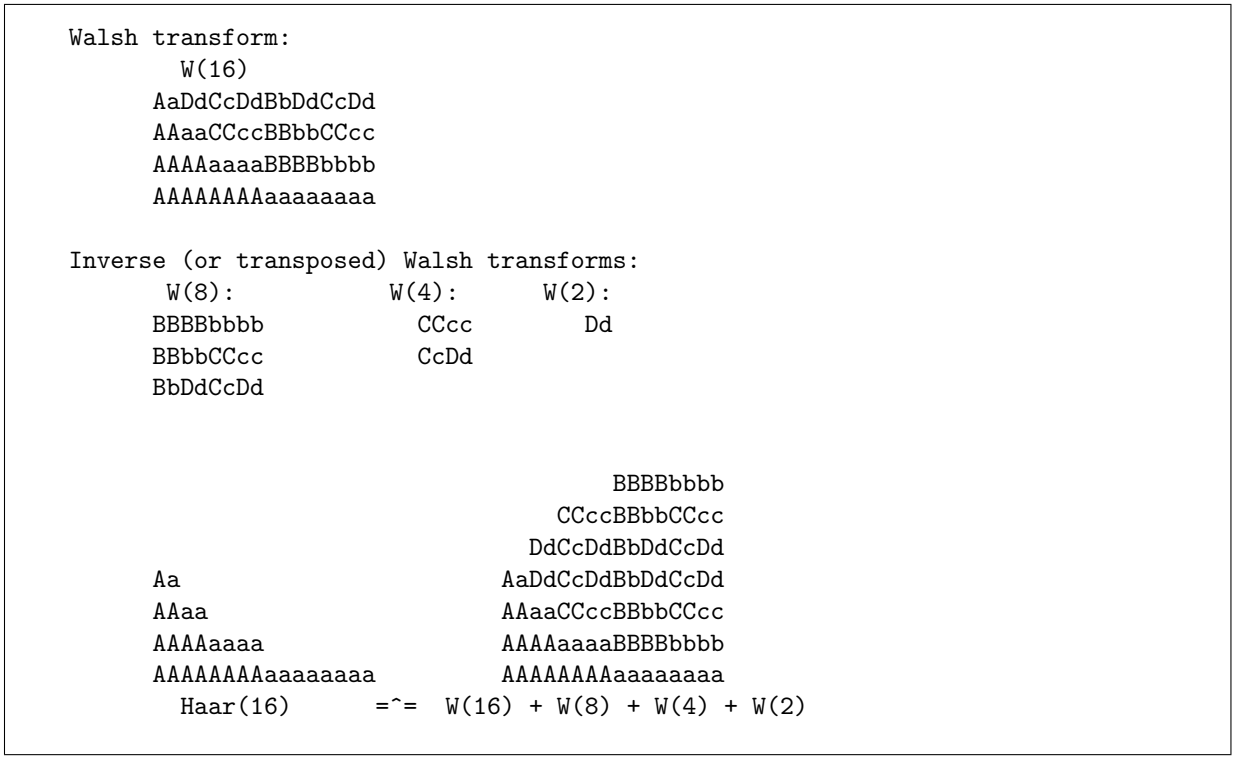


Figure 11.10: Symbolic description of how to build a Haar transform from Walsh transforms.

Haar transforms from Walsh transforms

The ($\sim n \log(n)$) schemes given here are not a efficient method to compute the Haar transform (which is $\sim n$). Instead, they can be used to identify the type of Haar transform that is the building block of a given Walsh transform.

The non-normalized transposed reversed Haar transform can (up to normalization) be obtained via

```
// algorithm HW1:  transposed_haar_rev_nn(f, ldn); =^=
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    walsh_wak(f+k, ldk);
}
walsh_wak(f, ldn);
```

and its inverse as

```
// algorithm HW1I:  inverse_transposed_haar_rev_nn(f, ldn); =^=
walsh_wak(f, ldn);
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    walsh_wak(f+k, ldk);
}
```

The non-normalized transposed Haar transform can (again, up to normalization) be obtained via

```
// algorithm HW2:  transposed_haar_nn(f, ldn); =^=
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    walsh_pal(f+k, ldk);
}
walsh_pal(f, ldn);
```

and its inverse as

```
// algorithm HW2I:  inverse_transposed_haar_nn(f, ldn); =^=
walsh_pal(f, ldn);  // =^= revbin_permute(f, n); walsh_wak(f, ldn);
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    walsh_pal(f+k, ldk);
}
```

The symbolic scheme is given in figure 11.10 on the previous page.

Chapter 12

The Hartley transform

The Hartley transform is a trigonometric transform whose practical importance comes from the fact that it maps real data to real data. While the fast algorithms for radix-2 can be found without great difficulty the higher radix algorithms are not obvious. Therefore it is appropriate to describe the Hartley transform in terms of the Fourier transform. A method for the conversion of FFT algorithms to fast Hartley transform (FHT) algorithms is given.

Routines for the conversion of Hartley transforms to and from Fourier transforms are described. Convolution routines based on the FHT are given for complex and real valued data. An efficient procedure for the computation of the negacyclic convolution is described.

12.1 Definition and symmetries

The *discrete Hartley transform* (HT) of a length-n sequence a is defined as

$$c = \mathcal{H}[a] \tag{12.1a}$$

$$c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \left(\cos \frac{2\,\pi\,k\,x}{n} + \sin \frac{2\,\pi\,k\,x}{n} \right) \tag{12.1b}$$

That is, almost like the Fourier transform but with '$\cos + \sin$' instead of '$\cos + i \cdot \sin$'.

The Hartley transform of a purely real sequence is purely real:

$$\mathcal{H}[a] \in \mathbb{R} \qquad \text{for} \quad a \in \mathbb{R} \tag{12.2}$$

It also is its own inverse:

$$\mathcal{H}[\mathcal{H}[a]] = a \tag{12.3}$$

Symmetry is conserved, like for the Fourier transform: The Hartley transform of a symmetric, antisymmetric sequence is symmetric, antisymmetric, respectively. Therefore one has

$$\mathcal{H}[a_S] = +\overline{\mathcal{H}[a_S]} = +\mathcal{H}[\overline{a_S}] \tag{12.4a}$$

$$\mathcal{H}[a_A] = -\overline{\mathcal{H}[a_A]} = -\mathcal{H}[\overline{a_A}] \tag{12.4b}$$

An algorithm for the fast ($n \log(n)$-) computation of the Hartley transform is called a *fast Hartley transform* (FHT).

12.2 Radix-2 FHT algorithms

12.2.1 Decimation in time (DIT) FHT

Notation: For a length-n sequence a of let $\mathcal{X}^{1/2}a$ denote the sequence with elements $a_x \cos \pi\, x/n + \overline{a}_x \sin \pi\, x/n$. The operator $\mathcal{X}^{1/2}$ is the equivalent to the operator $\mathcal{S}^{1/2}$ of the Fourier transform algorithms. The radix-2 decimation in time (DIT) step for the FHT:

$$\mathcal{H}\,[a]^{(left)} \quad \stackrel{n/2}{=} \quad \mathcal{H}\left[a^{(even)}\right] + \mathcal{X}^{1/2}\mathcal{H}\left[a^{(odd)}\right] \tag{12.5a}$$

$$\mathcal{H}\,[a]^{(right)} \quad \stackrel{n/2}{=} \quad \mathcal{H}\left[a^{(even)}\right] - \mathcal{X}^{1/2}\mathcal{H}\left[a^{(odd)}\right] \tag{12.5b}$$

This is the equivalent to relations 8.21a and 8.21b on page 257.

Pseudo code for a recursive radix-2 DIT FHT (C++ version in [FXT: file fht/recfht2.cc]):

```
procedure rec_fht_dit2(a[], n, x[])
// real a[0..n-1] input
// real x[0..n-1] result
{
    real b[0..n/2-1], c[0..n/2-1]   // workspace
    real s[0..n/2-1], t[0..n/2-1]   // workspace

    if n == 1 then
    {
        x[0] := a[0]
        return
    }

    nh := n/2;

    for k:=0 to nh-1
    {
        s[k] := a[2*k]    // even indexed elements
        t[k] := a[2*k+1]  // odd  indexed elements
    }

    rec_fht_dit2(s[], nh, b[])
    rec_fht_dit2(t[], nh, c[])

    hartley_shift(c[], nh, 1/2)

    for k:=0 to nh-1
    {
        x[k]    := b[k] + c[k];
        x[k+nh] := b[k] - c[k];
    }
}
```

The result is returned in the array in `x[]`. The procedure `hartley_shift()` implements the operator $\mathcal{X}^{1/2}$, it replaces element c_k of the input sequence c by $c_k\, cos(\pi\, k/n) + c_{n-k}\, sin(\pi\, k/n)$. As pseudo code:

```
procedure hartley_shift_05(c[], n)
// real c[0..n-1] input, result
{
    nh := n/2
    j := n-1
    for k:=1 to nh-1
    {
        c := cos( PI*k/n )
        s := sin( PI*k/n )

        {c[k], c[j]} := {c[k]*c+c[j]*s, c[k]*s-c[j]*c}

        j := j-1
    }
}
```

C++ implementations are given in [FXT: file fht/hartleyshift.h]. A version that exploits the symmetry of the trigonometric factors is

```
template <typename Type>
inline void hartley_shift_05_v2(Type *f, ulong n)
{
    const double phi0 = PI/n;
```

```
    double phi = 0.0;
    const ulong nh = n/2;
    if ( n>=4 )
    {
        ulong i=nh/2, j=3*i;
        Type fi = f[i],  fj = f[j];
        double cs = SQRT1_2;
        f[i] = (fi + fj) * cs;
        f[j] = (fi - fj) * cs;

        if ( n>=8 )
        {
            for (ulong i=1, j=n-1, k=nh-1, l=nh+1;  i<k;  ++i, --j, --k, ++l)
            {
                phi += phi0;
                double s, c;
                SinCos(phi, &s, &c);

                Type fi, fj;
                fi = f[i];
                fj = f[j];
                f[i] = fi * c + fj * s;
                f[j] = fi * s - fj * c;

                // swap(c, s);
                fi = f[k];
                fj = f[l];
                f[k] = fi * s + fj * c;
                f[l] = fi * c - fj * s;
            }
        }
    }
}
```

Pseudo code for a non-recursive radix-2 DIT FHT:

```
procedure fht_depth_first_dit2(a[], ldn)
// real a[0..n-1] input,result
{
    n := 2**ldn  // length of a[] is a power of 2

    revbin_permute(a[], n)

    for ldm:=1 to ldn
    {
        m  := 2**ldm
        mh := m/2
        m4 := m/4

        for r:=0 to n-m step m
        {
            for j:=1 to m4-1  // hartley_shift(a+r+mh,mh,1/2)
            {
                k := mh - j

                u := a[r+mh+j]
                v := a[r+mh+k]

                c := cos(j*PI/mh)
                s := sin(j*PI/mh)

                {u, v} := {u*c+v*s, u*s-v*c}

                a[r+mh+j] := u
                a[r+mh+k] := v
            }

            for j:=0 to mh-1
            {
                u := a[r+j]
                v := a[r+j+mh]

                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
        }
    }
}
```

The derivation of the 'usual' DIT2 FHT algorithm starts by combining the Hartley-shift with the sum/diff-operations [FXT: file fht/fhtdit2.cc]:

```
void fht_depth_first_dit2(double *f, ulong ldn)
{
    const ulong n = 1UL<<ldn;

    revbin_permute(f, n);

    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        const double phi0 = M_PI/mh;

        for (ulong r=0; r<n; r+=m)
        {
            {  // j == 0:
                ulong t1 = r;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }

            if ( m4 )
            {
                ulong t1 = r + m4;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }

            for (ulong j=1, k=mh-1; j<k; ++j,--k)
            {
                double s, c;
                SinCos(phi0*j, &s, &c);

                ulong tj = r + mh + j;
                ulong tk = r + mh + k;
                double fj = f[tj];
                double fk = f[tk];
                f[tj] = fj * c + fk * s;
                f[tk] = fj * s - fk * c;

                ulong t1 = r + j;
                ulong t2 = tj; // == t1 + mh;
                sumdiff(f[t1], f[t2]);

                t1 = r + k;
                t2 = tk; // == t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
        }
    }
}
```

Finally, as with the FFT equivalent (see page 258), the number of trigonometric computations can be reduced by swapping the innermost loops [FXT: file fht/fhtdit2.cc]:

```
void fht_dit2(double *f, ulong ldn)
// Radix-2 decimation in time (DIT) FHT.
{
    const ulong n = 1UL<<ldn;

    revbin_permute(f, n);

    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        const double phi0 = M_PI/mh;

        for (ulong r=0; r<n; r+=m)
        {
            {  // j == 0:
                ulong t1 = r;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }

            if ( m4 )
            {
                ulong t1 = r + m4;
                ulong t2 = t1 + mh;
```

```
                sumdiff(f[t1], f[t2]);
            }
        }

        for (ulong j=1, k=mh-1; j<k; ++j,--k)
        {
            double s, c;
            SinCos(phi0*j, &s, &c);

            for (ulong r=0; r<n; r+=m)
            {
                ulong tj = r + mh + j;
                ulong tk = r + mh + k;
                double fj = f[tj];
                double fk = f[tk];
                f[tj] = fj * c + fk * s;
                f[tk] = fj * s - fk * c;

                ulong t1 = r + j;
                ulong t2 = tj; // == t1 + mh;
                sumdiff(f[t1], f[t2]);

                t1 = r + k;
                t2 = tk; // == t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
        }
    }
}
```

12.2.2 Decimation in frequency (DIF) FHT

The radix-2 decimation in frequency step for the FHT is (compare to relations 8.24a and 8.24b on page 260):

$$\mathcal{H}\,[a]^{(even)} \quad \overset{n/2}{=} \quad \mathcal{H}\left[a^{(left)} + a^{(right)}\right] \tag{12.6a}$$

$$\mathcal{H}\,[a]^{(odd)} \quad \overset{n/2}{=} \quad \mathcal{H}\left[\mathcal{X}^{1/2}\left(a^{(left)} - a^{(right)}\right)\right] \tag{12.6b}$$

Pseudo code for a recursive radix-2 DIF FHT (the C++ equivalent is given in [FXT: file fht/recfht2.cc]):

```
procedure rec_fht_dif2(a[], n, x[])
// real a[0..n-1] input
// real x[0..n-1] result
{
    real b[0..n/2-1], c[0..n/2-1]   // workspace
    real s[0..n/2-1], t[0..n/2-1]   // workspace

    if n == 1 then
    {
        x[0] := a[0]
        return
    }

    nh := n/2;

    for k:=0 to nh-1
    {
        s[k] := a[k]     // 'left'  elements
        t[k] := a[k+nh]  // 'right' elements
    }

    for k:=0 to nh-1
    {
        {s[k], t[k]} := {s[k]+t[k], s[k]-t[k]}
    }

    hartley_shift(t[], nh, 1/2)

    rec_fht_dif2(s[], nh, b[])
    rec_fht_dif2(t[], nh, c[])

    j := 0
    for k:=0 to nh-1
    {
        x[j]   := b[k]
```

```
        x[j+1] := c[k]
        j := j+2
    }
}
```

Pseudo code for a non-recursive radix-2 DIF FHT (C++ version in [FXT: file fht/fhtdif2.cc]):

```
procedure fht_depth_first_dif2(a[], ldn)
// real a[0..n-1] input,result
{
    n := 2**ldn  // length of a[] is a power of 2
    for ldm:=ldn to 1 step -1
    {
        m  := 2**ldm
        mh := m/2
        m4 := m/4
        for r:=0 to n-m step m
        {
            for j:=0 to mh-1
            {
                u := a[r+j]
                v := a[r+j+mh]
                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
            for j:=1 to m4-1
            {
                k := mh - j
                u := a[r+mh+j]
                v := a[r+mh+k]
                c := cos(j*PI/mh)
                s := sin(j*PI/mh)
                {u, v} := {u*c+v*s, u*s-v*c}
                a[r+mh+j] := u
                a[r+mh+k] := v
            }
        }
    }
    revbin_permute(a[], n)
}
```

The 'usual' DIF2 FHT algorithm then is again obtained by swapping the inner loops, a C++ implementation is [FXT: `fht_dif2()` in fht/fhtdif2.cc]:

```
void fht_dif2(double *f, ulong ldn)
// Radix-2 decimation in frequency (DIF) FHT
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        const double phi0 = M_PI/mh;
        for (ulong r=0; r<n; r+=m)
        {
            {  // j == 0:
                ulong t1 = r;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
            if ( m4 )
            {
                ulong t1 = r + m4;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
        }
        for (ulong j=1, k=mh-1; j<k; ++j,--k)
```

```
        {
            double s, c;
            SinCos(phi0*j, &s, &c);

            for (ulong r=0; r<n; r+=m)
            {
                ulong tj = r + mh + j;
                ulong tk = r + mh + k;

                ulong t1 = r + j;
                ulong t2 = tj; // == t1 + mh;
                sumdiff(f[t1], f[t2]);

                t1 = r + k;
                t2 = tk; // == t1 + mh;
                sumdiff(f[t1], f[t2]);

                double fj = f[tj];
                double fk = f[tk];
                f[tj] = fj * c + fk * s;
                f[tk] = fj * s - fk * c;
            }
        }
    }

    revbin_permute(f, n);
}
```

12.3 Complex FT by HT

The relations between the HT and the FT can be read off directly from their definitions and their symmetry relations. Let σ be the sign of the exponent in the FT, then the HT of a complex sequence $d \in \mathbb{C}$ is:

$$\mathcal{F}\big[\,d\,\big] \;=\; \frac{1}{2}\left(\mathcal{H}\,[d] + \overline{\mathcal{H}\,[d]} + \sigma\, i\,\left(\mathcal{H}\,[d] - \overline{\mathcal{H}\,[d]}\right)\right) \tag{12.7}$$

Written out for the real and imaginary part $d = a + i\,b$ $(a, b \in \mathbb{R})$:

$$\mathrm{Re}\,\mathcal{F}\big[\,a + i\,b\,\big] \;=\; \frac{1}{2}\left(\mathcal{H}\,[a] + \overline{\mathcal{H}\,[a]} - \sigma\,\left(\mathcal{H}\,[b] - \overline{\mathcal{H}\,[b]}\right)\right) \tag{12.8a}$$

$$\mathrm{Im}\,\mathcal{F}\big[\,a + i\,b\,\big] \;=\; \frac{1}{2}\left(\mathcal{H}\,[b] + \overline{\mathcal{H}\,[b]} + \sigma\,\left(\mathcal{H}\,[a] - \overline{\mathcal{H}\,[a]}\right)\right) \tag{12.8b}$$

Using the symmetry relations 12.4a and 12.4b on page 357 one can recast the relations as

$$\mathrm{Re}\,\mathcal{F}\big[\,a + i\,b\,\big] \;=\; \frac{1}{2}\,\mathcal{H}\,[a_S - \sigma\,b_A] \tag{12.9a}$$

$$\mathrm{Im}\,\mathcal{F}\big[\,a + i\,b\,\big] \;=\; \frac{1}{2}\,\mathcal{H}\,[b_S + \sigma\,a_A] \tag{12.9b}$$

Both formulations lead to the very same conversion procedure.

Pseudo code for a complex FT by HT conversion:

```
fht_fft_conversion(a[], b[], n, is)
// preprocessing to use two length-n FHTs
// to compute a length-n complex FFT
// or
// postprocessing to use two length-n FHTs
// to compute a length-n complex FFT
//
// self-inverse
{
    for k:=1 to n/2-1
    {
        t := n-k

        as := a[k] + a[t]
        aa := a[k] - a[t]
```

```
        bs := b[k] + b[t]
        ba := b[k] - b[t]

        aa := is * aa
        ba := is * ba

        a[k] := 1/2 * (as - ba)
        a[t] := 1/2 * (as + ba)

        b[k] := 1/2 * (bs + aa)
        b[t] := 1/2 * (bs - aa)
    }
}
```

The C++ implementations are given in [FXT: file fft/fhtfft.cc] for type `double` and [FXT: file fft/fhtcfft.cc] for type `complex`.

Now we have two options to compute a complex FT by two HTs: Version 1 does the FHTs first:

```
fft_by_fht1(a[], b[], n, is)
// real a[0..n-1] input,result (real part)
// real b[0..n-1] input,result (imaginary part)
{
    fht(a[], n)
    fht(b[], n)
    fht_fft_conversion(a[], b[], n, is)
}
```

Version 2 does the FHTs at the end of the routine:

```
fft_by_fht2(a[], b[], n, is)
// real a[0..n-1] input,result (real part)
// real b[0..n-1] input,result (imaginary part)
{
    fht_fft_conversion(a[], b[], n, is)
    fht(a[], n)
    fht(b[], n)
}
```

Note that the real and imaginary parts of the FT are computed independently by this procedure. This can be very advantageous when the real and imaginary part of complex data lies in separate arrays. The C++ version is given in [FXT: file fft/fhtfft.cc].

12.4 Complex FT by complex HT and vice versa

A complex valued HT is simply two HTs (one of the real, one of the imaginary part). So we can use either version from section 12.3 and there is nothing new. Really? If one has a type `complex` version of both the conversion and the FHT routine then the complex FFT can be computed as either

```
fft_by_fht1(c[], n, is)
// complex c[0..n-1] input,result
{
    fht(c[], n)
    fht_fft_conversion(c[], n, is)
}
```

or the same with swapped statements.

This may not make you scream but here is the message: it makes sense to do so. One saves half of the trigonometric computations and book keeping. It is pretty easy to derive a complex FHT from the real version (in fact this is done automatically in FXT) and with a well optimized FHT you get an even better optimized FFT.

C++ implementations of complex FHTs are given in [FXT: file fht/cfhtdif.cc] (DIF algorithm), [FXT: file fht/cfhtdit.cc] (DIT algorithm), and [FXT: file fht/cfht0.cc] (version for zero padded data). All three files are script-converted versions of the corresponding real-valued FHTs.

The other way round: computation of a complex FHT using FFTs. Let T be the operator corresponding to the `fht_fft_conversion`. The operator is its own inverse: $T = T^{-1}$. We have seen that

$$\mathcal{F} = \mathcal{H} \cdot T \qquad \text{and} \qquad \mathcal{F} = T \cdot \mathcal{H} \tag{12.10}$$

Thereby (multiply the relations with T and use $T \cdot T = 1$):

$$\mathcal{H} = T \cdot \mathcal{F} \qquad \text{and} \qquad \mathcal{H} = \mathcal{F} \cdot T \tag{12.11}$$

Hence we have either

```
fht_by_fft(c[], n, is)
// complex c[0..n-1] input,result
{
    fft(c[], n)
    fht_fft_conversion(c[], n, is)
}
```

or the same thing with swapped lines [FXT: file fft/fhtcfft.cc]. The same ideas also work for separate real and imaginary parts but in that case one should rather use separate FHTs for the two arrays.

12.5 Real FT by HT and vice versa

To express the real and imaginary part of a Fourier transform of a purely real sequence $a \in \mathbb{R}$ by its Hartley transform use relations 12.8a and 12.8b on page 363 and set $b = 0$:

$$\operatorname{Re} \mathcal{F}\big[\, a \,\big] = \frac{1}{2} \left(\mathcal{H}\,[a] + \overline{\mathcal{H}\,[a]} \right) \tag{12.12a}$$

$$\operatorname{Im} \mathcal{F}\big[\, a \,\big] = \sigma \frac{1}{2} \left(\mathcal{H}\,[a] - \overline{\mathcal{H}\,[a]} \right) \tag{12.12b}$$

A C++ implementation is [FXT: `fht_real_complex_fft()` in realfft/realfftbyfht.cc]:

```
template <typename Type>
static inline void sumdiff05(Type &a, Type &b)
// {a, b}  <--| {0.5*(a+b),  0.5*(a-b)}
{ Type t=(a-b)*0.5; a+=b; a*=0.5; b=t; }

template <typename Type>
static inline void sumdiff05_r(Type &a, Type &b)
// {a, b}  <--| {0.5*(a+b), 0.5*(b-a)}
{ Type t=(b-a)*0.5; a+=b; a*=0.5; b=t; }

void
fht_real_complex_fft(double *f, ulong ldn, int is/*=+1*/)
{
    fht(f, ldn);

    const ulong n = (1UL<<ldn);

    if ( is>0 )  for (ulong i=1,j=n-1; i<j; i++,j--)  sumdiff05(f[i], f[j]);
    else         for (ulong i=1,j=n-1; i<j; i++,j--)  sumdiff05_r(f[i], f[j]);
}
```

At the end of the procedure the ordering of the output data $c = \mathcal{F}\big[\, a \,\big] \in \mathbb{C}$ is

$$\begin{aligned}
\texttt{a[0]} &= \mathrm{Re}\, c_0 \\
\texttt{a[1]} &= \mathrm{Re}\, c_1 \\
\texttt{a[2]} &= \mathrm{Re}\, c_2 \\
&\dots \\
\texttt{a[n/2]} &= \mathrm{Re}\, c_{n/2} \\
\texttt{a[n/2+1]} &= \mathrm{Im}\, c_{n/2-1} \\
\texttt{a[n/2+2]} &= \mathrm{Im}\, c_{n/2-2} \\
\texttt{a[n/2+3]} &= \mathrm{Im}\, c_{n/2-3} \\
&\dots \\
\texttt{a[n-1]} &= \mathrm{Im}\, c_1
\end{aligned} \tag{12.13}$$

The inverse procedure is given in [FXT: file realfft/realfftbyfht.cc]:

```
void
fht_complex_real_fft(double *f, ulong ldn, int is/*=+1*/)
{
    const ulong n = (1UL<<ldn);

    if ( is>0 )  for (ulong i=1,j=n-1; i<j; i++,j--)  sumdiff(f[i], f[j]);
    else         for (ulong i=1,j=n-1; i<j; i++,j--)  diffsum(f[i], f[j]);

    fht(f,ldn);
}
```

The function `sumdiff()` is defined in [FXT: file aux0/sumdiff.h]:

```
template <typename Type>
static inline void sumdiff(Type &a, Type &b)
// {a, b}  <--| {a+b, a-b}
{ Type t=a-b; a+=b; b=t; }

template <typename Type>
static inline void diffsum(Type &a, Type &b)
// {a, b}  <--| {a-b, a+b}
{ Type t=a-b; b+=a; a=t; }
```

The input has to be ordered as given above (relations 12.13). The sign of the transform (`is`) has to be the same as with the forward version.

Computation of a (real-valued) FHT using a real-valued FFT: Same line of thought as for complex versions. Let T_{r2c} be the operator corresponding to the post-processing in `real_complex_fft_by_fht()`, and T_{c2r} correspond to the preprocessing in `complex_real_fft_by_fht()`. That is

$$\mathcal{F}_{c2r} = \mathcal{H} \cdot T_{c2r} \qquad \text{and} \qquad \mathcal{F}_{r2c} = T_{r2c} \cdot \mathcal{H} \tag{12.14}$$

The operators are mutually inverse: $T_{r2c} = T_{c2r}^{-1}$ and $T_{c2r} = T_{r2c}^{-1}$. Multiplying the relations and using $T_{r2c} \cdot T_{c2r} = T_{c2r} \cdot T_{r2c} = 1$ gives

$$\mathcal{H} = T_{c2r} \cdot \mathcal{F}_{r2c} \qquad \text{and} \qquad \mathcal{H} = \mathcal{F}_{c2r} \cdot T_{r2c} \tag{12.15}$$

The corresponding code should be obvious. Watch out for real-to-complex FFTs that use a different ordering than given in 12.13.

12.6 Higher radix FHT algorithms

Higher radix FHT algorithms seem to get complicated due to the structure of the Hartley shift operator. In fact there is a straightforward way to turn any FFT decomposition into an FHT algorithm.

For the moment assume that we want to compute a complex HT, further assume we want to transform an radix-r algorithm. At each step we have r short HTs and want to combine them to a longer HT but we do not know how this might be done.

In section 12.3 on page 363 we learned how to turn a HT into an FT using the T-operator. And we have seen radix-r algorithms for the FFT.

The crucial idea is to use the conversion operator T as a wrapper around the FFT-step that combines several short FTs into a longer one. Here is how to turn a radix-r FFT-step into an FHT-step, simply do the following:

1. first convert the r short HTs into FTs (use T on the subsequences)
2. then perform the radix-r the FFT step
3. finally convert the FT into a HT (use T on the sequence)

For efficient implementations one obviously wants to combine the computations.

To obtain real-valued FHTs note that the real and imaginary parts do not 'mix': one can use the identical algorithm with real input (and the corresponding data types).

With a radix-r step the scheme always accesses $2r$ elements simultaneously. The symmetry of the trigonometric factors is thereby automatically exploited.

12.7 Convolution via FHT

The convolution property of the Hartley transform can be stated as

$$\mathcal{H}[a \circledast b] = \frac{1}{2}\Big(\mathcal{H}[a]\,\mathcal{H}[b] - \overline{\mathcal{H}[a]}\,\overline{\mathcal{H}[b]} + \mathcal{H}[a]\,\overline{\mathcal{H}[b]} + \overline{\mathcal{H}[a]}\,\mathcal{H}[b]\Big) \tag{12.16}$$

or, with $c := \mathcal{H}[a]$ and $d := \mathcal{H}[b]$, written element-wise:

$$\mathcal{H}[a \circledast b]_k = \frac{1}{2}\Big(c_k\,d_k - \overline{c_k}\,\overline{d_k} + c_k\,\overline{d_k} + \overline{c_k}\,d_k\Big) \tag{12.17a}$$

$$= \frac{1}{2}\Big(c_k\,(d_k + \overline{d_k}) + \overline{c_k}\,(d_k - \overline{d_k})\Big) \tag{12.17b}$$

The latter form reduces the number of multiplications. When turning the relation into an algorithm one has to keep in mind that both elements $y_k = \mathcal{H}[a \circledast b]$ and y_{-k} must be computed simultaneously.

Pseudo code

Pseudo code for the cyclic convolution of two real valued sequences `x[]` and `y[]` via the FHT. `n` must be even, the result is returned in `y[]`:

```
procedure fht_cyclic_convolution(x[], y[], n)
// real x[0..n-1] input, modified
// real y[0..n-1] result
{
    // transform data:
    fht(x[], n)
    fht(y[], n)

    // convolution in transformed domain:
    j := n-1
    for i:=1 to n/2-1
    {
        xi := x[i]
        xj := x[j]

        yp := y[i] + y[j]   // ==   y[j] + y[i]
```

```
        ym := y[i] - y[j]   // == -(y[j] - y[i])
        y[i] := (xi*yp + xj*ym)/2
        y[j] := (xj*yp - xi*ym)/2
        j := j-1
    }
    y[0] := x[0]*y[0]
    if n>1 then  y[n/2] := x[n/2]*y[n/2]
    // transform back:
    fht(y[], n)
    // normalize:
    for i:=0 to n-1
    {
        y[i] := y[i] / n
    }
}
```

It is assumed that the procedure `fht()` does no normalization. The C++ equivalent is given in [FXT: file convolution/fhtcnvl.cc].

Equation 12.17a on the preceding page (slightly optimized) for the auto convolution is

$$\mathcal{H}\left[a \circledast a\right]_k = \frac{1}{2}\Big(c_k\left(c_k + \overline{c_k}\right) + \overline{c_k}\left(c_k - \overline{c_k}\right)\Big) \tag{12.18a}$$

$$= c_k\,\overline{c_k} + \frac{1}{2}\left(c_k^2 - \overline{c_k}^2\right) \tag{12.18b}$$

where $c = \mathcal{H}\left[a\right]$.

We give pseudo code for the cyclic auto convolution that uses a fast Hartley transform, `n` must be even:

```
procedure cyclic_self_convolution(x[], n)
// real x[0..n-1] input, result
{
    // transform data:
    fht(x[], n)
    // convolution in transformed domain:
    j := n-1
    for i:=1 to n/2-1
    {
        ci := x[i]
        cj := x[j]
        t1 := ci*cj               // == cj*ci
        t2 := 1/2*(ci*ci-cj*cj)   // == -1/2*(cj*cj-ci*ci)
        x[i] := t1 + t2
        x[j] := t1 - t2
        j := j-1
    }
    x[0]   := x[0]*x[0]
    if n>1 then  x[n/2] := x[n/2]*x[n/2]
    // transform back:
    fht(x[], n)
    // normalize:
    for i:=0 to n-1
    {
        x[i] := x[i] / n
    }
}
```

For odd `n` replace the line

```
    for i:=1 to n/2-1
```

by

```
    for i:=1 to (n-1)/2
```

and omit the line

```
    if n>1 then  x[n/2] := x[n/2]*x[n/2]
```

in both procedures above.

C++ implementations

The FHT based routine for the cyclic convolution of two real sequences is:

```
void fht_convolution(double * restrict f, double * restrict g, ulong ldn)
{
    fht(f, ldn);
    fht(g, ldn);
    fht_convolution_core(f, g, ldn);
    fht(g, ldn);
}
```

The equivalent to the element-wise multiplication is given in [FXT: file convolution/fhtcnvlcore.cc]:

```
void
fht_convolution_core(const double * restrict f, double * restrict g, ulong ldn,
                     double v/*=0.0*/)
// Auxiliary routine for the computation of convolutions
//   via Fast Hartley Transforms.
// ldn := base-2 logarithm of the array length.
// v!=0.0 chooses alternative normalization.
{
    const ulong n = (1UL<<ldn);

    if ( v==0.0 )  v = 1.0/n;

    g[0]  *=  (v * f[0]);
    const ulong  nh = n/2;

    if ( nh>0 )
    {
        g[nh] *= (v * f[nh]);
        v *= 0.5;
        for (ulong i=1,j=n-1; i<j; i++,j--)  fht_mul(f[i], f[j], g[i], g[j], v);
    }
}
```

where [FXT: file convolution/fhtmulsqr.h]:

```
template <typename Type>
static inline  void
fht_mul(Type xi, Type xj, Type &yi, Type &yj, double v)
// yi <-- v*( 2*xi*xj + xi*xi - xj*xj )
// yj <-- v*( 2*xi*xj - xi*xi + xj*xj )
{
    Type h1p = xi,  h1m = xj;
    Type s1 = h1p + h1m,  d1 = h1p - h1m;
    Type h2p = yi,  h2m = yj;
    yi = (h2p * s1 + h2m * d1) * v;
    yj = (h2m * s1 - h2p * d1) * v;
}
```

A C++ implementation of the FHT based self-convolution is given in [FXT: file convolution/fhtcnvla.cc]. It uses the routine

```
void
fht_auto_convolution_core(double *f, ulong ldn,
                          double v/*=0.0*/)
// v!=0.0 chooses alternative normalization
{
    const ulong  n  = (1UL<<ldn);
    if ( v==0.0 )  v = 1.0/n;
    f[0] *= (v * f[0]);
    if ( n>=2 )
    {
        const ulong  nh = n/2;
        f[nh] *= (v * f[nh]);
        v *= 0.5;
        for (ulong i=1,j=n-1; i<nh; i++,j--)  fht_sqr(f[i], f[j], v);
    }
}
```

where [FXT: file convolution/fhtmulsqr.h]:

```
template <typename Type>
static inline  void
fht_sqr(Type &xi, Type &xj, double v)
// xi <-- v*( 2*xi*xj + xi*xi - xj*xj )
```

```
// xj <-- v*( 2*xi*xj - xi*xi + xj*xj )
{
    Type a = xi,  b = xj;
    Type s1 = (a + b) * (a - b);
    a *= b;
    a += a;
    xi = (a+s1) * v;
    xj = (a-s1) * v;
}
```

Avoiding the revbin permutations

The observation that the revbin permutations can be omitted with FFT based convolutions (see page 287) applies again [FXT: file convolution/fhtcnvlcore.cc]:

```
void
fht_convolution_revbin_permuted_core(const double * restrict f,
                                     double * restrict g,
                                     ulong ldn,
                                     double v/*=0.0*/)
// Same as fht_convolution_core() but with data access in revbin order.
{
    const ulong n = (1UL<<ldn);

    if ( v==0.0 )  v = 1.0/n;

    g[0] *= (v * f[0]);  // 0 == revbin(0)
    if ( n>=2 )  g[1] *= (v * f[1]); // 1 == revbin(nh)

    if ( n<4 )  return;

    v *= 0.5;
    const ulong nh = (n>>1);

    ulong r=nh, rm=n-1; // nh == revbin(1),  n1-1 == revbin(n-1)
    fht_mul(f[r], f[rm], g[r], g[rm], v);

    ulong k=2, km=n-2;
    while ( k<nh  )
    {
        // k even:
        rm -= nh;
        ulong tr = r;
        r^=nh;  for (ulong m=(nh>>1); !((r^=m)&m); m>>=1)  {;}
        fht_mul(f[r], f[rm], g[r], g[rm], v);
        --km;
        ++k;

        // k odd:
        rm += (tr-r);
        r += nh;
        fht_mul(f[r], f[rm], g[r], g[rm], v);
        --km;
        ++k;
    }
}
```

The optimized version saving three revbin permutations is [FXT: file convolution/fhtcnvl.cc]:

```
void fht_convolution(double * restrict f, double * restrict g, ulong ldn)
{
    fht_dif_core(f, ldn);
    fht_dif_core(g, ldn);
    fht_convolution_revbin_permuted_core(f, g, ldn);
    fht_dit_core(g, ldn);
}
```

12.8 Negacyclic convolution via FHT

Pseudo code for the computation of the negacyclic (auto-) convolution via FHT:

```
procedure negacyclic_self_convolution(x[], n)
// real x[0..n-1]  input, result
```

```
{
    // preprocessing:
    hartley_shift_05(x, n)

    // transform data:
    fht(x, n)

    // convolution in transformed domain:
    j := n-1
    for i:=0 to n/2-1  // here i starts from zero
    {
        a := x[i]
        b := x[j]

        x[i] := a*b+(a*a-b*b)/2
        x[j] := a*b-(a*a-b*b)/2
        j := j-1
    }

    // transform back:
    fht(x, n)

    // postprocessing:
    hartley_shift_05(x, n)
}
```

C++ implementations for the negacyclic convolution and self convolution are given in [FXT: file convolution/fhtnegacnvl.cc] The FHT-based negacyclic convolution is turns out to be extremely useful for the computation of weighted transforms, for example in the MFA-based convolution for real input, see section 9.7 on page 297.

12.9 Localized FHT algorithms

Localized routines for the FHT can be obtained by slight modifications of the corresponding algorithms for the Walsh transform described in section 10.5 on page 312. The decimation in time (DIT) version is [FXT: file fht/fhtloc2.h]:

```
template <typename Type>
void fht_loc_dit2_core(Type *f, ulong ldn)
{
    if ( ldn<=13 )
    {
        fht_dit_core(f, ldn);
        return;
    }

    // Recursion:
    fht_dit_core_2(f+2);  // ldm==1
    fht_dit_core_4(f+4);  // ldm==2
    fht_dit_core_8(f+8);  // ldm==3
    for (ulong ldm=4; ldm<ldn; ++ldm)  fht_loc_dit2_core(f+(1UL<<ldm), ldm);

    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        hartley_shift_05(f+mh, mh);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
    }
}
```

The routine `hartley_shift_05()` is described in 12.2.1 on page 358. One should choose a implementation that uses trigonometric recursion as this improves performance considerably.

The decimation in frequency (DIF) version is:

```
template <typename Type>
void fht_loc_dif2_core(Type *f, ulong ldn)
{
    if ( ldn<=13 )  // sizeof(Type)*(2**thres) <= L1_CACHE_BYTES
    {
        fht_dif_core(f, ldn);
```

```
        return;
    }

    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
        hartley_shift_05(f+mh, mh);
    }

    // Recursion:
    fht_dif_core_2(f+2);  // ldm==1
    fht_dif_core_4(f+4);  // ldm==2
    fht_dif_core_8(f+8);  // ldm==3
    for (ulong ldm=4; ldm<ldn; ++ldm)  fht_loc_dif2_core(f+(1UL<<ldm), ldm);
}
```

An additional revbin permutation is needed if the data is required in order. A FHT can be computed by either

```
    fht_loc_dif2_core(f, ldn);
    revbin_permute(f, 1UL<<ldn);
```

or

```
    revbin_permute(f, 1UL<<ldn);
    fht_loc_dit2_core(f, ldn);
```

Performance for large arrays is excellent: The convolutions based on the transforms [FXT: file convolution/fhtloccnvl.cc]

```
void
loc_fht_convolution(double * restrict f, double * restrict g, ulong ldn)
{
    fht_loc_dif2_core(f, ldn);
    fht_loc_dif2_core(g, ldn);
    fht_convolution_revbin_permuted_core(f, g, ldn);
    fht_loc_dit2_core(g, ldn);
}
```

and [FXT: file convolution/fhtloccnvla.cc]

```
void
loc_fht_auto_convolution(double *f, ulong ldn)
{
    fht_loc_dif2_core(f, ldn);
    fht_auto_convolution_revbin_permuted_core(f, ldn);
    fht_loc_dit2_core(f, ldn);
}
```

gave a significant (more than 50 percent) speedup for the high precision multiplication routines (see section 14.3 on page 391) used in the hfloat package.

12.10 Two-dimensional FHTs

A two-dimensional FHT can be computed almost as easy as a two-dimensional FFT, only a trivial additional step is needed. Start with the row-column algorithm described in section 8.10 on page 282 [FXT: file fht/twodimfht.cc]:

```
void
row_column_fht(double *f, ulong nr, ulong nc)
// FHT over rows and columns.
// nr := number of rows
// nc := number of columns
{
    ulong n = nr * nc;

    // fht over rows:
    ulong ldc = ld(nc);
    for (ulong k=0; k<n; k+=nc)  FHT(f+k, ldc);

    // fht over columns:
    double *w = new double[nr];
    for (ulong k=0; k<nc; k++)  skip_fht(f+k, nr, nc, w);
```

```
    delete [] w;
}
```

Note that no attempt has been made to make the routine cache friendly: The routine `skip_fht()` [FXT: file fht/skipfht.cc] simply copies a column into the scratch array, does the FHT and copies the data back. This is not yet a two-dimensional FHT, the following post-processing must be made:

```
void
y_transform(double *f, ulong nr, ulong nc)
// Transforms row-column-FHT to 2-dimensional FHT.
// Self-inverse.
// nr := number of rows
// nc := number of columns
{
    ulong rh = nr/2;
    if ( nr&1 )  rh++;

    ulong ch = nc/2;
    if ( nc&1 )  ch++;

    ulong n = nr*nc;
    for (ulong tr=1, ctr=nc; tr<rh; tr++,ctr+=nc) // ctr=nc*tr
    {
        double *pa = f + ctr;
        double *pb = pa + nc;
        double *pc = f + n - ctr;
        double *pd = pc + nc;
        for (ulong tc=1; tc<ch; tc++)
        {
            pa++;
            pb--;
            pc++;
            pd--;
            double e = (*pa + *pd - *pb - *pc) * 0.5;
            *pa -= e;
            *pb += e;
            *pc += e;
            *pd -= e;
        }
    }
}
```

The canned routine is therefore

```
void
twodim_fht(double *f, ulong nr, ulong nc)
// Two dimensional fast Hartley transform (FHT)
// nr := number of rows
// nc := number of columns
{
    row_column_fht(f, nr, nc);
    y_transform(f, nr, nc);
}
```

12.11 Discrete cosine transform (DCT) by HT

The discrete cosine transform (DCT) with respect to the basis

$$\begin{aligned} u(k) &= \nu(k) \cdot \cos\left(\frac{\pi\, k\,(i+1/2)}{n}\right) \quad \text{where} \\ \nu(k) &= \begin{cases} 1 & \text{if} \quad k=0 \\ \sqrt{2} & \text{else} \end{cases} \end{aligned} \tag{12.19}$$

can be computed from the FHT using an auxiliary routine which is its own inverse. As pseudo code:

```
procedure cos_rot(x[], y[], n)
// Real x[0..n-1] input
// Real y[0..n-1] result
{
    nh := n/2
    x[0]  := y[0]
```

```
    x[nh] := y[nh]
    phi := PI/2/n
    for k:=1 to nh-1
    {
        c := cos(phi*k)
        s := sin(phi*k)
        cps := (c+s)*sqrt(1/2)
        cms := (c-s)*sqrt(1/2)
        y[k]   := cms*x[k] + cps*x[n-k]
        y[n-k] := cps*x[k] - cms*x[n-k]
    }
}
```

The C++ equivalent is [FXT: `cos_rot()` in dctdst/cosrot.cc].

Pseudo code for the computation of the DCT via FHT:

```
procedure dcth(x[], ldn)
// real x[0..n-1] input,result
{
    n := 2**ldn
    real y[0..n-1]  // workspace
    unzip_rev(x, y, n)
    fht(y[],ldn)
    cos_rot(y[], x[], n)
}
```

where `unzip_rev()` is the reversed unzip permutation (see section 2.6 on page 89):

```
procedure unzip_rev(a[], b[], n)
// real a[0..n-1] input
// real b[0..n-1] result
{
    nh := n/2
    for k:=0 to nh-1
    {
        k2 := 2*k
        b[k]    := a[k2]
        b[nh+k] := a[n-1-k2]
    }
}
```

Pseudo code for the computation of the inverse discrete cosine transform (IDCT) via FHT:

```
procedure idcth(x[], ldn)
// real x[0..n-1] input,result
{
    n := 2**ldn
    real y[0..n-1]  // workspace
    cos_rot(x[], y[], n);
    fht(y[],ldn)
    zip_rev(y[], x[], n)
}
```

where `zip_rev()` is the reversed zip permutation:

```
procedure zip_rev(a[], b[], n)
// real a[0..n-1] input
// real b[0..n-1] result
{
    nh := n/2
    for k:=0 to nh-1
    {
        k2 := 2*k
        b[k]    := a[k2]
        b[nh+k] := a[n-1-k2]
    }
}
```

The C++ implementations of both the forward and the backward transform [FXT: file dctdst/dcth.cc] avoid the temporary array if no scratch space is supplied.

The algorithms are given in [79] and [80]. An alternative variant for the computation of the DCT that also uses the FHT is given in [FXT: file dctdst/dctzapata.cc], the algorithm is described in [6].

12.12 Discrete sine transform (DST) by DCT

The basis of the *discrete sine transform* (DST) is

$$u(k) \quad = \quad \sin\left(\frac{\pi\,(k+1)\,(i+1/2)}{n}\right) \tag{12.20}$$

Pseudo code for the computation of the DST via the discrete cosine transform (DCT):

```
procedure dst(x[],ldn)
// real x[0..n-1] input,result
{
    n := 2**ldn
    nh := n/2
    for k:=1 to n-1 step 2
    {
        x[k] := -x[k]
    }
    dct(x, ldn)
    for k:=0 to nh-1
    {
        swap(x[k], x[n-1-k])
    }
}
```

The corresponding C++ implementation is [FXT: `dsth()` in dctdst/dsth.cc].

Pseudo code for the computation of the inverse sine transform (IDST) using the inverse cosine transform (IDCT):

```
procedure idst(x[],ldn)
// real x[0..n-1] input,result
{
    n := 2**ldn
    nh := n/2
    for k:=0 to nh-1
    {
        swap(x[k], x[n-1-k])
    }
    idct(x, ldn)
    for k:=1 to n-1 step 2
    {
        x[k] := -x[k]
    }
}
```

The C++ version is [FXT: `idsth()` in dctdst/dsth.cc].

12.13 Automatic generation of transform codes

FFT *generators* are programs that output FFT routines, usually for fixed (short) lengths. In fact the thoughts here are not at all restricted to FFT codes. However, fast transforms and unroll-able routines like those for matrix multiplication or convolution are prime candidates for automated generation.

One can write code generators that have a built-in algorithmic knowledge. We restrict our attention to a simpler method known as *partial evaluation.* Writing such a program is easy: Take an existing FFT and change all computations into print statements that emit the necessary code. The process, however, is less than delightful and error-prone.

It would be much better to have another program that takes the existing FFT code as input and emit the code for the generator. Let us call this a *meta-generator*. Implementing such a meta-generator of course is highly nontrivial. It actually is equivalent to writing an interpreter for the language used plus

the necessary data flow analysis. A practical compromise is to write a program that, while theoretically not even close to a meta-generator, creates output that, after a little hand editing, is a usable generator code.

One may further want to print the current values of the loop variables of the original code as comments at the beginning of a block. Thereby it is possible to locate the corresponding part (with respect to both file and temporal location) of a piece of generated code in the original file. In addition one may keep the comments of the original code.

With FFTs it is necessary to identify ('reverse engineer') the trigonometric values that occur in the process in terms of the corresponding argument (rational multiples of π). The actual values should be inlined to some greater precision than actually needed, thereby one avoids the generation of multiple copies of the (logically) same value with differences only due to numeric inaccuracies. Printing the arguments, both as they appear and in lowest terms, inside comments helps to understand (or further optimize) the generated code:

```
double c1=.98078528040323044912618223613 4; // == cos(Pi*1/16) == cos(Pi*1/16)
double s1=.19509032201612826784828486847 6; // == sin(Pi*1/16) == sin(Pi*1/16)
double c2=.92387953251128675612818318939 7; // == cos(Pi*2/16) == cos(Pi*1/8)
double s2=.38268343236508977172845998402 9; // == sin(Pi*2/16) == sin(Pi*1/8)
```

Automatic verification of the generated codes against the original is a mandatory part of the process.

A level of abstraction for the array indices is of great use: When the print statements in the generator emit some function of the index instead of its plain value it is easy to generate modified versions of the code for permuted input. That is, instead of

```
cout<<"sumdiff(f0, f2, g["<<k0<<"], g["<<k2<<"]);" <<endl;
cout<<"sumdiff(f1, f3, g["<<k1<<"], g["<<k3<<"]);" <<endl;
```

use

```
cout<<"sumdiff(f0, f2, "<<idxf(g,k0)<<", "<<idxf(g,k2)<<");" <<endl;
cout<<"sumdiff(f1, f3, "<<idxf(g,k1)<<", "<<idxf(g,k3)<<");" <<endl;
```

where `idxf(g, k)` can be defined to print a modified (for example, revbin-permuted) index `k`.

A generated length-8 DIT FHT core (from [FXT: file fht/shortfhtditcore.h]) shall serve as an example:

```
template <typename Type>
inline void fht_dit_core_8(Type *f)
// unrolled version for length 8
{
 { // start initial loop
  { // fi = 0  gi = 1
   Type g0, f0, f1, g1;
   sumdiff(f[0], f[1], f0, g0);
   sumdiff(f[2], f[3], f1, g1);
   sumdiff(f0, f1);
   sumdiff(g0, g1);
   Type s1, c1, s2, c2;
   sumdiff(f[4], f[5], s1, c1);
   sumdiff(f[6], f[7], s2, c2);
   sumdiff(s1, s2);
   sumdiff(f0, s1, f[0], f[4]);
   sumdiff(f1, s2, f[2], f[6]);
   c1 *= SQRT2;
   c2 *= SQRT2;
   sumdiff(g0, c1, f[1], f[5]);
   sumdiff(g1, c2, f[3], f[7]);
  }
 } // end initial loop
}
// opcount by generator:  #mult=2=0.25/pt   #add=22=2.75/pt
```

Generated DIF FHT codes for lengths up to 64 are given in [FXT: file fht/shortfhtdifcore.h].

The generated codes can be of great use when one wants to spot parts of the original code that need further optimization. Especially repeated trigonometric values and unused symmetries tend to be apparent in the unrolled code.

It is a good idea to let the generator count the number of operations (multiplications, additions, loads and stores) of the code it emits. It is even better if those numbers are compared to the corresponding values found in the compiled assembler code.

Checking the generated machine code

It is possible to have GCC produce the assembler code with the original source interlaced which is a great tool with code optimization. The necessary commands are (include- and warning flags omitted)

```
# create assembler code:
c++ -S -fverbose-asm -g -O2  test.cc -o test.s
# create asm interlaced with source lines:
as -alhnd test.s  > test.lst
```

For example, the generated length-4 DIT FHT core from [FXT: file fht/shortfhtditcore.h] is

```
template <typename Type>
inline void fht_dit_core_4(Type *f)
// unrolled version for length 4
{
   Type f0, f1, f2, f3;
   sumdiff(f[0], f[1], f0, f1);
   sumdiff(f[2], f[3], f2, f3);
   sumdiff(f0, f2, f[0], f[2]);
   sumdiff(f1, f3, f[1], f[3]);
}
```

With `Type` set to `double` the generated assembler is (some editing for readability)

```
  16:test.cc **** void fht_dit_core_4(double *f)
  17:test.cc **** {
  18:test.cc ****     double f0, f1, f2, f3;
  19:test.cc ****     sumdiff(f[0], f[1], f0, f1);
  49 0000 660F120F            movlpd  (%rdi), %xmm1   #* f, tmp63
  50 0004 660F1247            movlpd  8(%rdi), %xmm0  #, tmp64
  50      08
  20:test.cc ****     sumdiff(f[2], f[3], f2, f3);
  52 0009 660F1257            movlpd  16(%rdi), %xmm2 #, tmp67
  52      10
  54 000e F20F10D9            movsd   %xmm1, %xmm3    # tmp63, f0
  56 0012 F20F5CC8            subsd   %xmm0, %xmm1    # tmp64, f1
  59 0016 F20F10E2            movsd   %xmm2, %xmm4    # tmp67, f2
  62 001a F20F58D8            addsd   %xmm0, %xmm3    # tmp64, f0
  64 001e 660F1247            movlpd  24(%rdi), %xmm0 #, tmp68
  64      18
  65 0023 F20F58E0            addsd   %xmm0, %xmm4    # tmp68, f2
  66 0027 F20F5CD0            subsd   %xmm0, %xmm2    # tmp68, f3
  21:test.cc ****     sumdiff(f0, f2, f[0], f[2]);
  69 002b F20F10C3            movsd   %xmm3, %xmm0    # f0, tmp71
  70 002f F20F58C4            addsd   %xmm4, %xmm0    # f2, tmp71
  71 0033 F20F5CDC            subsd   %xmm4, %xmm3    # f2, f0
  72 0037 F20F1107            movsd   %xmm0, (%rdi)   # tmp71,* f
  22:test.cc ****     sumdiff(f1, f3, f[1], f[3]);
  74 003b F20F10C1            movsd   %xmm1, %xmm0    # f1, tmp73
  75 003f F20F5CCA            subsd   %xmm2, %xmm1    # f3, f1
  77 0043 F20F115F            movsd   %xmm3, 16(%rdi) # f0,
  77      10
  79 0048 F20F58C2            addsd   %xmm2, %xmm0    # f3, tmp73
  80 004c F20F114F            movsd   %xmm1, 24(%rdi) # f1,
  80      18
  81 0051 F20F1147            movsd   %xmm0, 8(%rdi)  # tmp73,
  81      08
  23:test.cc **** }
```

Note that the assembler code is not always in sync with the corresponding source lines, especially with higher levels of optimization.

12.14 Eigenvectors of the Fourier and Hartley transform *

For $a_S := a + \overline{a}$, the symmetric part of a sequence a:

$$\mathcal{F}\big[\,\mathcal{F}\big[\,a_S\,\big]\,\big] \;=\; a_S \tag{12.21}$$

Now let $u_+ := a_S + \mathcal{F}\big[\,a_S\,\big]$ and $u_- := a_S - \mathcal{F}\big[\,a_S\,\big]$ then

$$\mathcal{F}\big[\,u_+\,\big] \;=\; \mathcal{F}\big[\,a_S\,\big] + a_S = \;\; a_S + \mathcal{F}\big[\,a_S\,\big] \;= +1 \cdot u_+ \tag{12.22a}$$
$$\mathcal{F}\big[\,u_-\,\big] \;=\; \mathcal{F}\big[\,a_S\,\big] - a_S = -(a_S - \mathcal{F}\big[\,a_S\,\big]) = -1 \cdot u_- \tag{12.22b}$$

Both u_+ and u_- are symmetric.

For $a_A := a - \overline{a}$, the antisymmetric part of a, we have

$$\mathcal{F}\big[\,\mathcal{F}\big[\,a_A\,\big]\,\big] \;=\; -a_A \tag{12.23}$$

Therefore with $v_+ := a_A + i\,\mathcal{F}\big[\,a_A\,\big]$ and $v_- := a_A - i\,\mathcal{F}\big[\,a_A\,\big]$:

$$\mathcal{F}\big[\,v_+\,\big] \;=\; \mathcal{F}\big[\,a_A\,\big] - i\,a_A = -i\,(a_A + i\,\mathcal{F}\big[\,a_A\,\big]) = -i \cdot v_+ \tag{12.24a}$$
$$\mathcal{F}\big[\,v_-\,\big] \;=\; \mathcal{F}\big[\,a_A\,\big] + i\,a_A = +i\,(a_A - i\,\mathcal{F}\big[\,a_A\,\big]) = +i \cdot v_- \tag{12.24b}$$

Both v_+ and v_- are antisymmetric. u_+, u_-, v_+ and v_- are *eigenvectors* of the FT, with *eigenvalues* $+1$, -1, $-i$ and $+i$ respectively. The eigenvectors are pairwise perpendicular.

Using

$$a \;=\; \frac{1}{2}\,(u_+ + u_- + v_+ + v_-) \tag{12.25}$$

we can, for a given sequence, find a transform that is a 'square root' of the FT: Compute u_+, u_-, v_+ and v_-. Then for $\lambda \in \mathbb{R}$ one can define a transform $\mathcal{F}^\lambda\,[a]$ as

$$\mathcal{F}^\lambda\,[a] \;=\; \frac{1}{2}\Big((+1)^\lambda\,u_+ + (-1)^\lambda\,u_- + (-i)^\lambda\,v_+ + (+i)^\lambda\,v_-\Big) \tag{12.26}$$

Then $\mathcal{F}^0\,[a]$ is the identity and $\mathcal{F}^1\,[a]$ is the usual FT. The transform $\mathcal{F}^{1/2}\,[a]$ is a transform so that $\mathcal{F}^{1/2}\,\big[\mathcal{F}^{1/2}\,[a]\big] = \mathcal{F}\big[\,a\,\big]$, that is, a 'square root' of the FT. The transform $\mathcal{F}^{1/2}\,[a]$ is not unique as the expressions $\pm 1^{1/2}$ and $\pm i^{1/2}$ aren't.

The eigenvectors of the Hartley Transform are

$$u_+ \;:=\; a + \mathcal{H}\,[a] \tag{12.27a}$$
$$u_- \;:=\; a - \mathcal{H}\,[a] \tag{12.27b}$$

The eigenvalues are ± 1, one has $\mathcal{H}\,[u_+] = +1 \cdot u_+$ and $\mathcal{H}\,[u_-] = -1 \cdot u_-$.

Chapter 13

Number theoretic transforms

In this chapter we introduce the number theoretic transforms (NTTs). After understanding the necessary concepts from number theory (see also chapter 21) it turns out that the routines for the fast NTTs are rather straightforward translations of the FFT algorithms. We give radix-2 and radix-4 routines but there should be no difficulty to translate any given (complex valued) FFT algorithm into the equivalent NTT algorithm. For the translation of real valued FFT (or FHT) routines one needs an idea about how to express sines and cosines in modular arithmetic, this is presented in sections 21.11.6 and 21.11.7.

As no rounding errors occur with the underlying modular arithmetic the main application of NTTs is the fast computation of exact convolutions.

13.1 Basic concepts from number theory

How to make a number theoretic transform out of your FFT:
'*Replace* $exp(\pm 2\pi i/n)$ *by a primitive* n*-th root of unity, done.*'

We want to implement FFTs in $\mathbb{Z}/m\mathbb{Z}$ (the ring of integers modulo some integer m) instead of $\mathbb{C}$, the (field of the) complex numbers. These FFTs are called *number theoretic transforms* (NTTs), *mod* m *FFTs* or (if m is a prime) *prime modulus transforms*.

There is a restriction for the choice of m: For a length n NTT we need a primitive n-th root of unity. A number r is called an n-th root of unity if $r^n = 1$. It is called a *primitive* n*-th root* if $r^k \neq 1 \; \forall \, k < n$.

In $\mathbb{C}$ matters are simple: $e^{\pm 2\pi i/n}$ is a primitive n-th root of unity for arbitrary n. $e^{2\pi i/21}$ is a 21-th root of unity. $r = e^{2\pi i/3}$ is also 21-th root of unity but not a primitive root, because $r^3 = 1$. A primitive n-th root of 1 in $\mathbb{Z}/m\mathbb{Z}$ is also called an *element of order* n. The 'cyclic' property of the elements r of order n lies in the heart of all FFT algorithms: $r^{n+k} = r^k$.

In $\mathbb{Z}/m\mathbb{Z}$ things are not that simple: for a given modulus m primitive n-th roots of unity do not exist for arbitrary n. They exist for some maximal order R only. Roots of unity of an order different from R are available only for the divisors d_i of R: r^{R/d_i} is a d_i-th root of unity because $(r^{R/d_i})^{d_i} = r^R = 1$.

Therefore n must divide R, the first condition for NTTs:

$$n \backslash R \quad \Longleftrightarrow \quad \exists \; \sqrt[n]{1} \tag{13.1}$$

The operations needed in FFTs are addition, subtraction and multiplication. Division is not needed, except for division by n for the final normalization after transform and back-transform. Division by n is

multiplication by the inverse of n. Hence n must be invertible in $\mathbb{Z}/m\mathbb{Z}$: n must be coprime[1] to m, the second condition for NTTs:

$$n \perp m \iff \exists\, n^{-1} \text{ in } \mathbb{Z}/m\mathbb{Z} \tag{13.2}$$

We restrict our attention to prime moduli, though NTTs are also possible with composite moduli. If the modulus is a prime p then $\mathbb{Z}/p\mathbb{Z}$ is the field $\mathbb{F}_p = GF(p)$: All elements except 0 have inverses and 'division is possible'. Thereby the second condition is trivially fulfilled for all FFT lengths $n < p$: a prime p is coprime to all integers $n < p$.

Roots of unity are available for the maximal order $R = p-1$ and its divisors: Therefore the first condition on n for a length-n mod p FFT being possible is that n divides $p-1$. This restricts the choice for p to primes of the form $p = v\,n + 1$: For length-$n = 2^k$ FFTs one will use primes like $p = 3 \cdot 5 \cdot 2^{27} + 1$ (31 bits), $p = 13 \cdot 2^{28} + 1$ (32 bits), $p = 3 \cdot 29 \cdot 2^{56} + 1$ (63 bits) or $p = 27 \cdot 2^{59} + 1$ (64 bits). Primes of that form are not 'exceptional', see [Lipson].

Primes suitable with NTTs can be generated using [FXT: demo mod/fftprimes-demo.cc]. A shortened sample output is

```
arg 1: 62 == wb  [word bits, wb<=63]  default=62
arg 2: 0.01 == deltab  [results are in the range [wb-deltab, wb]]  default=0.01
 minb = 61.99 = wb-0.01
arg 3: 44 == minx  [log_2(min(fftlen))]  default=44
  ---- x = 44: -----
4580495072570638337 = 0x3f91300000000001 = 1 + 2^44 * 83 * 3137  (61.9902 bits)
4581058022524059649 = 0x3f93300000000001 = 1 + 2^44 * 3 * 11 * 13 * 607  (61.9904 bits)
4582113536867246O9 = 0x3f96f00000000001 = 1 + 2^44 * 3 * 7 * 79 * 157  (61.9907 bits)
4585702359639785473 = 0x3fa3b00000000001 = 1 + 2^44 * 3^2 * 11 * 2633  (61.9918 bits)
4587039365779161089 = 0x3fa8700000000001 = 1 + 2^44 * 7 * 193^2  (61.9923 bits)
4587391209500049409 = 0x3fa9b00000000001 = 1 + 2^44 * 3 * 17 * 5113  (61.9924 bits)
4588130081313914881 = 0x3fac500000000001 = 1 + 2^44 * 3 * 5 * 17387  (61.9926 bits)
4589572640569556993 = 0x3fb1700000000001 = 1 + 2^44 * 11 * 37 * 641  (61.9931 bits)
[--snip--]
4610999923171655681 = 0x3ffd900000000001 = 1 + 2^44 * 5 * 19 * 31 * 89  (61.9998 bits)
4611105476287922177 = 0x3ffdf00000000001 = 1 + 2^44 * 262111  (61.9998 bits)
  ---- x = 45: -----
4580336742896238593 = 0x3f90a00000000001 = 1 + 2^45 * 29 * 67^2  (61.9902 bits)
4581533011547258881 = 0x3f94e00000000001 = 1 + 2^45 * 3 * 5 * 8681  (61.9905 bits)
4584347761314365441 = 0x3f9ee00000000001 = 1 + 2^45 * 5 * 11 * 23 * 103  (61.9914 bits)
4587655092290715649 = 0x3faaa00000000001 = 1 + 2^45 * 3 * 7^2 * 887  (61.9925 bits)
[--snip--]
  ---- x = 48: -----
4585508845593296897 = 0x3fa3000000000001 = 1 + 2^48 * 11 * 1481  (61.9918 bits)
  ---- x = 49: -----
4582975570802900993 = 0x3f9a000000000001 = 1 + 2^49 * 7 * 1163  (61.991 bits)
4595360469778169857 = 0x3fc6000000000001 = 1 + 2^49 * 3^2 * 907  (61.9949 bits)
  ---- x = 50: -----
4601552919265804289 = 0x3fdc000000000001 = 1 + 2^50 * 61 * 67  (61.9968 bits)
```

13.2 Implementation of NTTs

To implement NTTs ('mod-m FFTs') one basically must implement modulo-m arithmetics and replace $e^{\pm 2\pi i/n}$ by an n-th root of unity in $\mathbb{Z}/m\mathbb{Z}$ in the code. The C++ class implementing modular arithmetics in FXT is [FXT: `class mod` in mod/mod.h].

For the back-transform one uses the (mod m) inverse r^{-1} of r (an element of order n) that was used for the forward transform. To check whether r^{-1} exists if $\gcd(r, m) = 1$ (in a field all elements except zero are invertible so the test can often be omitted). To compute the inverse modulo m one can use the relation $r^{-1} = r^{\varphi(m)-1} \pmod m$ which is $r^{-1} = r^{m-2} \pmod m$ for m prime.

While the notion of the Fourier transform as a 'decomposition into frequencies' seems to be meaningless for NTTs the algorithms are denoted with 'decimation in time/frequency' in analogy to those in the complex domain.

[1] n coprime to $m \iff \gcd(n, m) = 1$

The nice feature of NTTs is that there is no loss of precision in the transform as there is always with the floating point FFTs. Using the trigonometric recursion in its most naive form is mandatory, as the computation of roots of unity is expensive.

13.2.1 Radix-2 DIT NTT

Pseudo code for the radix-2 decimation in time (DIT) NTT (to be called with `ldn=log2(n)`):

```
procedure mod_fft_dit2(f[], ldn, is)
// mod_type f[0..2**ldn-1]
{
    n := 2**ldn

    rn := element_of_order(n)  // (mod_type)

    if is<0 then  rn := rn**(-1)

    revbin_permute(f[], n)

    for ldm:=1 to ldn
    {
        m  := 2**ldm
        mh := m/2

        dw := rn**(2**(ldn-ldm))   // (mod_type)
        w  := 1                    // (mod_type)

        for j:=0 to mh-1
        {
            for r:=0 to n-m step m
            {
                t1 := r+j
                t2 := t1+mh

                v := f[t2]*w  // (mod_type)
                u := f[t1]    // (mod_type)

                f[t1] := u+v
                f[t2] := u-v
            }

            w := w*dw  // trig recursion
        }
    }
}
```

Like shown in section 8.3.1 on page 256 it is a good idea to extract the `ldm==1` stage of the outermost loop: Replace

```
    for ldm:=1 to ldn
    {
```

by

```
    for r:=0 to n-1 step 2
    {
        {f[r], f[r+1]} := {f[r]+f[r+1], f[r]-f[r+1]}
    }
    for ldm:=2 to ldn
    {
```

The C++ implementation is given in [FXT: file ntt/nttdit2.cc]:

```
void
ntt_dit2(mod *f, ulong ldn, int is)
// Radix-2 decimation in time (DIT) NTT
{
    const ulong n = 1UL<<ldn;
    revbin_permute(f, n);

    for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);

    for (ulong ldm=2; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);

        const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
        mod w = (mod::one);
```

```
        for (ulong j=0; j<mh; ++j)
        {
            for (ulong r=0; r<n; r+=m)
            {
                const ulong t1 = r + j;
                const ulong t2 = t1 + mh;

                mod v = f[t2] * w;
                mod u = f[t1];

                f[t1] = u + v;
                f[t2] = u - v;
            }
            w *= dw;
        }
    }
}
```

The elements of order 2^k are precomputed upon initialization of the `mod` class. The call to `mod::root2pow()` is a simple table lookup.

13.2.2 Radix-2 DIF NTT

Pseudo code for the radix-2 decimation in frequency (DIF) NTT:

```
procedure mod_fft_dif2(f[], ldn, is)
// mod_type f[0..2**ldn-1]
{
    n := 2**ldn
    dw := element_of_order(n)  // (mod_type)

    if is<0 then  dw := rn**(-1)

    for  ldm:=ldn to 1 step -1
    {
        m  := 2**ldm
        mh := m/2

        w := 1  // (mod_type)

        for j:=0 to mh-1
        {
            for r:=0 to n-m step m
            {
                t1 := r+j
                t2 := t1+mh

                v := f[t2]  // (mod_type)
                u := f[t1]  // (mod_type)

                f[t1] :=  u+v
                f[t2] := (u-v)*w
            }

            w := w*dw  // trig recursion
        }

        dw := dw*dw
    }

    revbin_permute(f[], n)
}
```

As in section 8.3.2 on page 259 extract the `ldm==1` stage of the outermost loop:
Replace the line

```
    for  ldm:=ldn to 1 step -1
```

by

```
    for  ldm:=ldn to 2 step -1
```

and insert

```
    for r:=0 to n-1 step 2
    {
        {f[r], f[r+1]} := {f[r]+f[r+1], f[r]-f[r+1]}
    }
```

before the call of `revbin_permute(f[],n)`.

The C++ implementation is given in [FXT: file ntt/nttdif2.cc]:

```
void
ntt_dif2(mod *f, ulong ldn, int is)
// Radix-2 decimation in frequency (DIF) NTT
{
    const ulong n = (1UL<<ldn);
    mod dw = mod::root2pow( is>0 ? ldn : -ldn );

    for (ulong ldm=ldn; ldm>1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);

        mod w = mod::one;

        for (ulong j=0; j<mh; ++j)
        {
            for (ulong r=0; r<n; r+=m)
            {
                const ulong t1 = r + j;
                const ulong t2 = t1 + mh;

                mod v = f[t2];
                mod u = f[t1];

                f[t1] = (u + v);
                f[t2] = (u - v) * w;
            }
            w *= dw;
        }
        dw *= dw;
    }

    for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);

    revbin_permute(f, n);
}
```

13.2.3 Radix-4 NTTs

The radix-4 versions of the NTT are straightforward translations of the routines that use complex numbers. We simply give the C++ implementations

Decimation in time (DIT) algorithm

Code for a radix-4 decimation in time (DIT) NTT [FXT: file ntt/nttdit4.cc]:

```
static const ulong LX = 2;

void
ntt_dit4(mod *f, ulong ldn, int is)
{
    const ulong n = (1UL<<ldn);

    revbin_permute(f, n);

    if ( ldn & 1 )  // n is not a power of 4, need a radix-2 step
    {
        for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);
    }

    const mod imag = mod::root2pow( is>0 ? 2 : -2 );

    ulong ldm = LX + (ldn&1);
    for ( ; ldm<=ldn ; ldm+=LX)
    {
        const ulong m = (1UL<<ldm);
        const ulong m4 = (m>>LX);

        const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
        mod w = (mod::one);
        mod w2 = w;
        mod w3 = w;

        for (ulong j=0; j<m4; j++)
```

```
        {
            for (ulong r=0, i0=j+r; r<n; r+=m, i0+=m)
            {
                const ulong i1 = i0 + m4;
                const ulong i2 = i1 + m4;
                const ulong i3 = i2 + m4;

                mod a0 = f[i0];
                mod a2 = f[i1] * w2;
                mod a1 = f[i2] * w;
                mod a3 = f[i3] * w3;

                mod t02 = a0 + a2;
                mod t13 = a1 + a3;

                f[i0] = t02 + t13;
                f[i2] = t02 - t13;

                t02 = a0 - a2;
                t13 = a1 - a3;
                t13 *= imag;

                f[i1] = t02 + t13;
                f[i3] = t02 - t13;
            }

            w *= dw;
            w2 = w * w;
            w3 = w * w2;
        }
    }
}
```

Decimation in frequency (DIF) algorithm

Code for a radix-4 decimation in frequency (DIT) NTT [FXT: file ntt/nttdif4.cc]:

```
static const ulong LX = 2;

void
ntt_dif4(mod *f, ulong ldn, int is)
// Radix-4 decimation in frequency (DIF) NTT
{
    const ulong n = (1UL<<ldn);

    const mod imag = mod::root2pow( is>0 ? 2 : -2 );

    for (ulong ldm=ldn; ldm>=LX; ldm-=LX)
    {
        const ulong m = (1UL<<ldm);
        const ulong m4 = (m>>LX);

        const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
        mod w = (mod::one);
        mod w2 = w;
        mod w3 = w;

        for (ulong j=0; j<m4; j++)
        {
            for (ulong r=0, i0=j+r; r<n; r+=m, i0+=m)
            {
                const ulong i1 = i0 + m4;
                const ulong i2 = i1 + m4;
                const ulong i3 = i2 + m4;

                mod a0 = f[i0];
                mod a1 = f[i1];
                mod a2 = f[i2];
                mod a3 = f[i3];

                mod t02 = a0 + a2;
                mod t13 = a1 + a3;

                f[i0] = (t02 + t13);
                f[i1] = (t02 - t13) * w2;

                t02 = a0 - a2;
                t13 = a1 - a3;
                t13 *= imag;

                f[i2] = (t02 + t13) * w;
                f[i3] = (t02 - t13) * w3;
```

```
            }
            w *= dw;
            w2 = w * w;
            w3 = w * w2;
        }
    }
    if ( ldn & 1 )  // n is not a power of 4, need a radix-2 step
    {
        for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);
    }
    revbin_permute(f, n);
}
```

13.3 Convolution with NTTs

The NTTs are natural candidates for the computation of *exact* integer convolutions, as used in high precision multiplication algorithms. One must keep in mind that 'everything is mod m', the largest value that can be represented is $m-1$. One can simply choose a big enough m to get rid of this problem. If m does not fit into a single machine word this may slow down the computation unacceptably. The way out is to choose m as the product of several coprime moduli that are all just below machine word size, compute the convolutions for each modulus and finally use the Chinese Remainder Theorem (see section 21.4 on page 527) to obtain the convolution modulo m.

If using length-n FFTs for convolution there must be an inverse element for n. This imposes the condition $\gcd(n, m) = 1$, the modulus must be prime to n. For length-2^k FFTs this simply means that m must be odd.

C++ code for the NTT based exact convolution can be found in [FXT: file ntt/nttcnvl.cc]. The routines are virtually identical to their comples equivalents, for example, a routine for cyclic self-convolution can be given as

```
void ntt_auto_convolution(mod *f, ulong ldn)
{
    check_two_invertible();  // so we can normalize later
    int is = +1;
    MOD_FFT(f, ldn, is);  // transform
    const ulong n = (1UL<<ldn);
    for (ulong i=0; i<n; ++i)  f[i] *= f[i];  // multiply element-wise
    MOD_FFT(f, ldn, -is);  // inverse transform
    multiply_val(f, n, (mod(n)).inv() ); // normalize
}
```

As explained on page 287 for complex FFTs the revbin permutations can be avoided.

For further applications of the NTT see the survey article [55] which also has a good bibliography.

Part III

Arithmetical algorithms

Chapter 14

Fast multiplication and exponentiation

This chapter describes multiplication algorithms that are asymptotically better than the well-known scheme which uses proportional N^2 operations for the multiplication of two N-digit numbers. In addition, the two most simple schemes for binary exponentiation are given.

14.1 Asymptotics of algorithms

An important feature of an algorithm is the number of operations that must be performed for the completion of a task of a certain size N. The quantity N should be some reasonable quantity that grows strictly with the size of the task. For high precision computations one will take the length of the numbers counted in decimal digits or bits. For computations with square matrices one may take for N the number of rows. An operation is typically a (machine word) multiplication plus an addition, one could also simply count machine instructions.

We now consider the computational work (cost) of a given algorithm. An algorithm is said to have some asymptotics $f(N)$ if it needs proportional $f(N)$ operations for a task of size N. We use the symbol $\sim f(N)$ to express proportionality.

Examples:

- Addition of two N-digit numbers needs $\sim N$ operations (here: machine word addition).
- Ordinary multiplication of two N-digit numbers needs $\sim N^2$ operations.
- Counting elements equal to a given value in an unsorted array of length N costs $\sim N$ operations. The operations are reads from memory and comparisons.
- Deciding whether a sorted array of length N contains a given value costs $\sim \log(N)$ operations when binary search is used.
- The Fast Fourier Transform (FFT) of a length-N sequence needs $\sim N \log(N)$ operations (a straightforward implementation of the Fourier Transform, i.e. computing N sums each of length N would be $\sim N^2$).
- Matrix multiplication (of two $N \times N$ matrices by the obvious algorithm) is $\sim N^3$ (N^2 sums each of N products).
- When the problem size for matrix multiplication is taken to be $M = N^2$, the number of elements, then the cost is 'only' $\sim M^{3/2}$.

- Deciding whether a N-digit binary number is even or odd costs 1 operation (lookup whether the lowest bit is zero).

The algorithm with the 'best' asymptotics wins for some, possibly huge, N. For smaller N another algorithm will be superior. For the exact break-even point the constants omitted elsewhere are, of course, important. The approximate cost of an algorithm is expressed by the symbol $\approx C \cdot f(N)$.

Example: Let the algorithm `mult1` take $\approx 1.0 \cdot N^2$ operations, `mult2` take $\approx 8.0 \cdot N \log_2(N)$ operations. Then, for $N < 64$ `mult1` is faster and for $N > 64$ `mult2` is faster. Completely different algorithms may be optimal for the same task at different problem sizes. The 'hidden constants' can be so large that the asymptotically better algorithm is never practical for any feasible problem size.

We have simplified the considerations by assuming that for a given algorithm $\sim f(N)$ there is a constant C so that

$$\lim_{N \to \infty} \text{work}(N) \quad = \quad C \tag{14.1}$$

where work(N) is the actual cost for problem size N.

With many algorithms it is only possible to give lower and upper (asymptotic) bounds, especially if the program flow depends on the processed data.

We further will not consider the space requirements as the only two basic cases will occur: in-place algorithms (space requirement ~ 1) and algorithms were a small (constant) multiple of the size of the data processed is needed.

Note that it is possible to construct algorithms where the computational cost alone is not of much value: Integer factorization (up to a certain maximal value) costs just one lookup if a precomputed table of factorizations is stored in an (insanely) big array.

For a (much) more fine-grained approach to measuring the costs of algorithms see [Int2Alg].

14.2 The Karatsuba algorithm

Ordinary multiplication is $\sim N^2$. Assuming the hidden constant equals one the computation of the product of two million-digit numbers would require $\approx 10^{12}$ operations, taking about 1 day on a machine that does 10 million operations per second (MIPS). But there are better ways ...

Split the numbers U and V (assumed to have approximately the same length/precision) into two pieces

$$\begin{aligned} U &= U_0 + B\,U_1 \\ V &= V_0 + B\,V_1 \end{aligned} \tag{14.2}$$

where B is a power of the radix[1] (or base) close to the half length of U and V. The straightforward multiplication that needs 4 multiplications with half precision for one multiplication with full precision:

$$U\,V \quad = \quad U_0\,V_0 + B\,(U_0\,V_1 + V_0\,U_1) + B^2\,U_1\,V_1 \tag{14.3}$$

If we use the relation

$$U\,V \quad = \quad (1+B)\,U_0\,V_0 + B\,(U_1 - U_0)\,(V_0 - V_1) + (B + B^2)\,U_1\,V_1 \tag{14.4}$$

we need 3 multiplications with half precision for one multiplication with full precision. Applying the scheme recursively until the numbers to multiply are of machine size we obtain an algorithm whose asymptotic cost is $\sim N^{\log_2(3)} \approx N^{1.585}$.

[1] For decimal numbers the radix is 10.

For squaring use either

$$U^2 \quad = \quad (1+B)\,U_0^2 - B\,(U_1-U_0)^2 + (B+B^2)\,U_1^2 \tag{14.5}$$

or

$$U^2 \quad = \quad (1-B)\,U_0^2 + B\,(U_1+U_0)^2 + (-B+B^2)\,U_1^2 \tag{14.6}$$

One can extend the above idea by splitting U and V into more than two pieces each, the resulting algorithm is called Toom Cook algorithm. However the algorithm becomes more complicated for 'finer' splittings.

We now assume that the hidden constant equals two as there is more bookkeeping overhead than with the simple algorithm. Computing the product of two million-digit numbers would require $\approx 2 \cdot (10^6)^{1.585} \approx 3200 \cdot 10^6$ operations, taking about 10 minutes on the 10 MIPS machine.

14.3 Fast multiplication via FFT

Multiplication of two numbers is essentially a convolution of the sequences of their digits. The (linear) convolution of the two sequences $a_k, b_k, k = 0 \ldots N-1$ is defined as the sequence c where

$$c_k \quad := \quad \sum_{i,j=0;\ i+j=k}^{N-1} a_i b_j \qquad k = 0 \ldots 2N-2 \tag{14.7}$$

A number written in radix R as

$$a_P\ a_{P-1}\ \ldots\ a_2\ a_1\ a_0\ .\ a_{-1}\ a_{-2}\ \ldots\ a_{-p+1}\ a_{-p} \tag{14.8}$$

denotes a quantity of

$$\sum_{i=-p}^{P} a_i \cdot R^i \quad = \quad a_P \cdot R^P + a_{P-1} \cdot R^{P-1} + \cdots + a_{-p} \cdot R^{-p}. \tag{14.9}$$

That means, the digits can be considered as coefficients of a polynomial in R. For example, with decimal numbers one has $R = 10$ and $123.4 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1}$. The product of two numbers is almost the polynomial product

$$\sum_{k=0}^{2N-2} c_k\,R^k \quad := \quad \sum_{i=0}^{N-1} a_i\,R^i \cdot \sum_{j=0}^{N-1} b_j\,R^j \tag{14.10}$$

The c_k are found by comparing coefficients. One easily checks that the c_k must satisfy the convolution equation 14.7.

As the c_k can be greater than 'nine' (that is, $R-1$), the result has to be 'fixed' using *carry* operations: Go from right to left, replace c_k by $c_k \% R$ and add $(c_k - c_k \% R)/R$ to its left neighbor.

An example: usually one would multiply the numbers 82 and 34 as follows:

$$\begin{array}{rrrr} & 82 & \times & 34 \\ \hline & 3 & {}^{3}2 & 8 \\ 2 & {}^{2}4 & 6 & \\ \hline = 2 & 7 & 8 & 8 \end{array}$$

We just said that the carries can be delayed to the end of the computation:

	82	×	34
		32	8
	24	6	
	24	38	8
$= 2$	$^2\,7$	$^3\,8$	8

... which is really polynomial multiplication (which in turn is a convolution of the coefficients):

	$(8\,x+2)$	×	$(3\,x+4)$
		$32\,x$	8
	$24\,x^2$	$6\,x$	
$=$	$24\,x^2$	$+38\,x$	$+8$

Convolution can be done efficiently using the Fast Fourier Transform (FFT): Convolution is a simple (element-wise) multiplication in Fourier space. The FFT itself takes $N \cdot \log N$ operations. Instead of the direct convolution ($\sim N^2$) one proceeds like this:

- Compute the FFTs of both factors.
- Multiply the transformed sequences element-wise.
- Compute inverse transform of the product.

To understand why this actually works note that (1) the multiplication of two polynomials can be achieved by the (more complicated) scheme:

- evaluate both polynomials at sufficiently many points (at least one more point than the degree of the product polynomial c: $\deg c = \deg a + \deg b$)
- element-wise multiply the values found
- find the polynomial corresponding to those (product-)values

and (2) that *the FFT is an algorithm for the parallel evaluation of a given polynomial at many points, namely the roots of unity.* (3) the inverse FFT is an algorithm to find (the coefficients of) a polynomial whose values are given at the roots of unity.

You might be surprised if you always thought of the FFT as an algorithm for the 'decomposition into frequencies'. There is no problem with either of these notions.

Re-launching our example, we use the fourth roots of unity ± 1 and $\pm i$:

	$a = (8\,x+2)$	×	$b = (3\,x+4)$	$c = a\,b$
$+1$	$+10$		$+7$	$+70$
$+i$	$+8i+2$		$+3i+4$	$+38i-16$
-1	-6		$+1$	-6
$-i$	$-8i+2$		$-3i+4$	$-38i-16$
				$c = (24\,x^2+38\,x+8)$

This table has to be read like this: first the given polynomials a and b are evaluated at the points given in the left column, thereby the columns below a and b are filled. Then the values are multiplied to fill the column below c, giving the values of c at the points. Finally, the actual polynomial c is found from those values, resulting in the lower right entry. You may find it instructive to verify that a 4-point FFT really evaluates a, b by transforming the sequences 0, 0, 8, 2 and 0, 0, 3, 4 by hand. The backward transform of 70, $38i-16$, -6, $-38i-16$ should produce the final result given for c.

The operation count is dominated by that of the FFTs (the element-wise multiplication is of course $\sim N$), so the whole fast convolution algorithm takes $\sim N \cdot \log N$ operations. The following carry operation is also $\sim N$ and can therefore be neglected when counting operations.

Assume the hidden constant equals five. Multiplying our million-digit numbers will need about

$$5 \cdot 10^6 \log_2(10^6) \quad \approx \quad 5 \cdot 10^6 \cdot 20 \; = \; 10^8$$

operations, taking approximately 10 seconds on a 10 Mips machine.

Strictly speaking $N \cdot \log N$ is not really the truth: it has to be $N \cdot \log N \cdot \log\log N$. This is because the sums in the convolutions have to be represented as exact integers. The biggest term C that can possibly occur is approximately $N\,R^2$ for a number with N digits (see next section). Therefore, working with some fixed radix R one has to compute the FFTs with $\log N$ bits precision, leading to an operation count of $N \cdot \log N \cdot \log N$. The slightly better $N \cdot \log N \cdot \log\log N$ can be obtained by recursive use of FFT multiplies. For realistic applications (where the sums in the convolution all fit into the machine type floating point numbers) it is safe to think of FFT multiplication being proportional $N \cdot \log N$.

For a survey of multiplication methods, some mathematical background and further references see [19]. Several alternative multiplication algorithms are given in [DEK2], chapter 4.3.3 ('How fast can we multiply?'). See [72] on how far the idea "polynomials for numbers" can be carried and where it fails.

14.4 Radix/precision considerations with FFT multiplication

This section describes the dependencies between the radix of the number and the achievable precision when using FFT multiplication.

We use (unsigned) 16 bit words for the digits. Thereby the radix of the numbers can be in the range 2, 3, ..., 65536($= 2^{16}$). When working in base ten one will actually use 'super-digits' of base 10,000, the largest power of ten that fits into a word. These super-digits are called `LIMB`s in `hfloat`.

With very large precision one cannot actually use the greatest power of the desired base. This is due to the fact that the components of the convolution must be representable as integer numbers with the data type used for the FFTs: The cumulative sums c_k have to be represented precisely enough to distinguish every (integer) quantity from the next bigger (or smaller) value. The highest possible value for a c_k will appear in the middle of the product and when multiplicand and multiplier consist of 'nines' (that is $R-1$) only. For radix R and a precision of N `LIMB`s Let the maximal possible value be C, then

$$C \quad = \quad N\,(R-1)^2 \tag{14.11}$$

Note that with FFT based convolution the *absolute value* of the central term can in fact equal $|C| = N^2\,(R-1)^2$. But there is no need to distinguish that many integers. After dividing by N we are back at relation 14.11.

The number of bits to represent C exactly is the integer greater or equal to

$$\log_2(N\,(R-1)^2) \quad = \quad \log_2 N + 2\,\log_2(R-1) \tag{14.12}$$

Due to numerical errors there must be a few more bits for safety. If computations are made using double-precision floating point numbers (C-type `double`) one typically has a mantissa of 53 bits[2] then we need to have

$$M \quad \geq \quad \log_2 N + 2\,\log_2(R-1) + S \tag{14.13}$$

where M :=mantissa-bits and S :=safety-bits. Using $\log_2(R-1) < \log_2(R)$:

$$N_{max}(R) \quad = \quad 2^{M-S-2\,\log_2(R)} \tag{14.14}$$

Suppose we have $M = 53$ mantissa-bits and require $S = 3$ safety-bits. With base 2 numbers one could use radix $R = 2^{16}$ for precisions up to a length of $N_{max} = 2^{53-3-2\cdot 16} = 256k$ `LIMB`s. Corresponding are 4096 kilo bits and = 1024 kilo hex digits. For greater lengths smaller radices have to be used according to the following table (extra horizontal line at the 16 bit limit for `LIMB`s):

[2] Of which only the 52 least significant bits are physically present, the most significant bit is implied to be always set.

Radix R	max # LIMBs	max # hex digits	max # bits
$2^{10} = 1024$	$1048,576\,k$	$2621,440\,k$	$10240\,M$
$2^{11} = 2048$	$262,144\,k$	$720,896\,k$	$2816\,M$
$2^{12} = 4096$	$65,536\,k$	$196,608\,k$	$768\,M$
$2^{13} = 8192$	$16384\,k$	$53,248\,k$	$208\,M$
$2^{14} = 16384$	$4096\,k$	$14,336\,k$	$56\,M$
$2^{15} = 32768$	$1024\,k$	$3840\,k$	$15\,M$
$2^{16} = 65536$	$256\,k$	$1024\,k$	$4\,M$
$2^{17} = 128\,k$	$64\,k$	$272\,k$	$1062\,k$
$2^{18} = 256\,k$	$16\,k$	$72\,k$	$281\,k$
$2^{19} = 512\,k$	$4\,k$	$19\,k$	$74\,k$
$2^{20} = 1\,M$	$1\,k$	$5\,k$	$19\,k$
$2^{21} = 2\,M$	256	1300	5120

For decimal numbers:

Radix R	max # LIMBs	max # digits	max # bits
10^2	$110\,G$	$220\,G$	$730\,G$
10^3	$1100\,M$	$3300\,M$	$11\,G$
10^4	$11\,M$	$44\,M$	$146\,M$
10^5	$110\,k$	$550\,k$	$1826\,k$
10^6	$1\,k$	6,597	$22\,k$
10^7	11	77	255

Summarizing:

- For decimal digits and precisions up to 11 million LIMBs use radix 10,000. (corresponding to more than 44 million decimal digits), for even greater precisions choose radix 1,000.
- For hexadecimal digits and precisions up to 256,000 LIMBs use radix 65,536 (corresponding to more than 1 million hexadecimal digits), for even greater precisions choose radix 4,096.

14.5 The sum-of-digits test

With high-precision calculations it is mandatory to add a sanity check to the multiplication routines. That is, false results due to loss of accuracy should (with high probability) be detected via the *sum-of-digits test*:

When computing the product $c = a \cdot b$ with radix-R numbers then compute the values ('sums of digits') $s_a = a \bmod (R-1)$ and $s_b = b \bmod (R-1)$ after the multiplication compute $s_c = c \bmod (R-1)$ and compare s_c to $s_m = s_a \cdot s_b \bmod (R-1)$. If $s_c \neq s_m$ then an error has occurred in the computation of c.

The sum-of-digits function s_a can for a radix-R, length-n number a be computed as [FXT: file mult/auxil.cc]:

```
ulong
sum_of_digits(const LIMB *a, ulong n, ulong nine, ulong s)
{
    for (ulong k=0; k<n; ++k)
    {
        s += a[k];
        if  ( s>=nine )  s -= nine;
    }
    return s;
}
```

where the variable `nine` has to be set to $R-1$ and `s` to zero. The latter is needed when a carry with the most significant digit of the result occurs.

The computation of $s_m = s_a \cdot s_b$ is done in

```
ulong
mult_sum_of_digits(const LIMB *a, ulong an,
                   const LIMB *b, ulong bn,
                   ulong nine)
{
    ulong qsa = sum_of_digits(a, an, nine, 0);
    ulong qsb = sum_of_digits(b, bn, nine, 0);
    ulong qsm = (qsa*qsb) % nine;
    return  qsm;
}
```

The checks in multiplication routine [FXT: file mult/fxtmultiply.cc] can be outlined as:

```
fxt_multiply(const LIMB *a, ulong an,
             const LIMB *b, ulong bn,
             LIMB *c, ulong cn,
             uint rx)
{
    const ulong nine = rx-1;
    ulong qsm=0, qsp=0;
    qsm = mult_sum_of_digits(a, an, b, bn, nine);

    // Multiply: c=a*b
    // If carrying through c gives an additional (leading) digit,
    // then set cy to that value,  else set cy=0.

    qsp = sum_of_digits(g, n, nine, cy);
    if ( qsm!=qsp )  { /* FAILED */ }
}
```

If we assume that a failed multiplication produces 'random' digits in c then the probability that a failed multiplication goes unnoticed equals $1/R$.

Omitting the sum-of-digits test is *not* an option: The situation that some number contains mainly 'nines' in the course of a high-precision calculation is very common. Thereby insufficient precision in the FFT-multiplication will almost certainly result in an error.

14.6 Binary exponentiation

The method of *binary exponentiation* (or *binary powering*) allows to compute the e-th power of a number a using about $\log(e)$ multiplications and squarings. The term 'number' can be replaced by about anything one can multiply. That includes integers, floating point numbers, polynomials, matrices, integer remainders modulo some modulus, polynomials modulo a polynomial and so on. In fact, the given algorithms work for any group: we do not need commutativity but $a^n \cdot a^m = a^{n+m}$ must hold (power-associativity).

Right-to-left powering

The algorithm uses the binary expansion of the exponent: Let $e \geq 0$, write e the base 2 as $e = [e_j, e_{j-1}, \ldots, e_1, e_0]$, $e_i \in \{0, 1\}$. Then

$$a^e = a^{1\cdot e_0}\, a^{2\cdot e_1}\, a^{4\cdot e_2}\, a^{8\cdot e_3}\, \cdots\, a^{2^j\, e_j} \tag{14.15a}$$

$$= 1\,(a^1)^{e_0}\,(a^2)^{e_1}\,(a^4)^{e_2}\,\cdots\,(a^{2^j})^{e_j} \tag{14.15b}$$

We initialize a variable t by one, generate the powers $s_i = a^{2^i}$ by successive squarings $s_i = s_{i-1}^2 = (a^{2^{i-1}})^2$ and multiply t by s_i if e_i equals one. The following C++ code computes the e-th power of the (double precision) number a:

```
double power_r2l(double a, ulong e)
{
    double t = 1;
    if ( e )
    {
```

```
        double s = a;
        while ( 1 )
        {
            if ( e & 1 )  t *= s;
            e /= 2;
            if ( 0==e )  break;
            s *= s;
        }
    }
    return t;
}
```

A trivial optimization is to avoid the multiplication by 1 if the exponent is a power of two:

```
double power_r2l(double a, ulong e)
{
    if ( 0==e )  return 1;
    double s = a;
    while ( 0==(e&1) )
    {
        s *= s;
        e /= 2;
    }
    a = s;
    while ( 0!=(e/=2) )
    {
        s *= s;
        if ( e & 1 )  a *= s;
    }
    return  a;
}
```

The demo [FXT: demo arith/power-r2l-demo.cc] shows the quantities that occur with the computation of $p = 2^{38}$:

```
    arg 1: 2 == a  [number to exponentiate]  default=2
    arg 2: 38 == e  [exponent]  default=38
     e = 1..11.
    0                          2                         2
    1                          4                         2
    1                         16                        64
    0                        256                        64
    0                      65536                        64
    1                 4294967296             274877906944
    p=a**e = 274877906944
```

In the right-to-left powering scheme the exponent is scanned starting from the lowest bit.

Left-to-right powering

The left-to-right binary powering algorithm scans the exponent starting from the highest bits. We use the facts that $a^{2k} = (a^k)^2$ and $a^{2k+1} = (a^k)^2\, a$. Implementation is simple:

```
double power_l2r(double a, ulong e)
{
    if ( 0==e )  return 1;
    double s = a;
    ulong b = highest_bit(e);
    while ( b>1 )
    {
        b >>= 1;
        s *= s;
        if ( e & b )  s *= a;
    }
    return s;
}
```

The demo [FXT: demo arith/power-l2r-demo.cc] shows the quantities that occur with the computation of $p = 2^{38}$ when the left-to-right scan is used:

```
arg 1: 2 == a  [number to exponentiate]  default=2
arg 2: 38 == e  [exponent]  default=38
 e = 1..11.
1                          2
0                          4                          4
0                         16                         16
1                        256                        512
1                     262144                     524288
0             274877906944               274877906944
p=a**e = 274877906944
```

All multiplications apart from the squarings happen with the unchanged value of a. This is an advantage if a is a small (integer) value so that the multiplications are cheap. As a slightly extreme example, if one computes $7^{7^7} \approx +0.3759823526783 \cdot 10^{695975}$ to full precision, then the left-to-right powering is about 3 times faster. If a is a full-precision number (and multiplication is done via FFTs) then the FFT of a only need to be computed once. Thereby all multiplications except for the first count a squarings. This technique is called *FFT caching*.

In practice the given powering algorithms are 'good enough'. There are schemes that improve further. For repeated power computations, especially for very large exponents the schemes based on the so-called *addition chains* lead to better algorithms, see [DEK2]. The so-called 'flexible window powering method' is described and analyzed in [38]. A readable survey of exponentiation methods is given in [47]. An algorithm for the efficient computation of products of powers (of different numbers) is described in [20]. These optimized exponentiation algorithms are mainly used with cryptographic applications.

Techniques for accelerating computations of factorials and binomial coefficients are described in [69].

14.6.1 Cost of binary exponentiation of full-precision numbers

e	e (radix 2)	#S	#M	#F	#C
1	1	0	0	0	
2	....1.	1	0	2	
3	11	1	1	5	
4	...1..	2	0	4	
5	...1.1	2	1	7	
6	...11.	2	1	7	
7	...111	2	2	10	9
8	..1...	3	0	6	
9	..1..1	3	1	9	
10	..1.1.	3	1	9	
11	..1.11	3	2	12	11
12	..11..	3	1	9	
13	..11.1	3	2	12	11
14	..111.	3	2	12	11
15	..1111	3	3	15	13
16	.1....	4	0	8	
17	.1...1	4	1	11	
18	.1..1.	4	1	11	
19	.1..11	4	2	14	13
20	.1.1..	4	1	11	

e	e (radix 2)	#S	#M	#F	#C
21	.1.1.1	4	2	14	13
22	.1.11.	4	2	14	13
23	.1.111	4	3	17	15
24	.11...	4	1	11	
25	.11..1	4	2	14	13
26	.11.1.	4	2	14	13
27	.11.11	4	3	17	15
28	.111..	4	2	14	13
29	.111.1	4	3	17	15
30	.1111.	4	3	17	15
31	.11111	4	4	20	17
32	1.....	5	0	10	
33	1....1	5	1	13	
34	1...1.	5	1	13	
35	1...11	5	2	16	15
36	1..1..	5	1	13	
37	1..1.1	5	2	16	15
38	1..11.	5	2	16	15
39	1..111	5	3	19	17
40	1.1...	5	1	13	

Figure 14.1: Cost of binary powering of full-precision numbers for small exponents e in terms of squarings (#S), multiplications (#M) and FFTs (#F). If the left-to-right exponentiation algorithm with FFT caching needs less FFTs then the number is given under (#C).

With full-precision numbers the cost of binary powering is the same for both the left-to-right and the

right-to-left algorithm. As an example, to raise x to the 26-th power, note that $e = 26 = 11010_2$ and we can write

$$x^{26} \;=\; x^{16} \cdot x^8 \cdot x^2 \;=\; (((x^2)^2)^2)^2 \cdot ((x^2)^2)^2 \cdot (x^2) \tag{14.16}$$

Here we need four squarings and two multiplications. In general one needs $\lfloor \log_2 e \rfloor$ squarings and $h(e) - 1$ multiplications were $h(e)$ is the number of set bits in the binary expansion of e. Figure 14.1 on the preceding page lists the cost of the exponentiation for small exponents e in terms of squarings and multiplications and, assuming a squaring costs two FFTs and multiplication three, in terms of FFTs. The table was created with the program [FXT: demo arith/power-costs-demo.cc].

Chapter 15

Root extraction

We describe methods to compute the inverse, square root, and higher roots of a given number. The computation of any of these costs just the equivalent of a few full-precision multiplications.

15.1 Division, square root and cube root

15.1.1 Inverse and division

The ordinary division algorithm is far too expensive for numbers of extreme precision. Instead one replaces the division $\frac{a}{d}$ by the multiplication of a with the inverse of d. The inverse of d is computed by finding a starting approximation $x_0 \approx \frac{1}{d}$ and then iterating

$$x_{k+1} \quad = \quad x_k + x_k\,(1 - d\,x_k) \tag{15.1}$$

until the desired precision is reached. The convergence is quadratic (second order), which means that the number of correct digits is doubled with each step: if $x_k = \frac{1}{d}(1+e)$ then $x_{k+1} = \frac{1}{d}\,\left(1 - e^2\right)$.

Moreover, each step only requires computations with twice the number of digits that were correct at its beginning. Still better: the multiplication $x_k(\dots)$ needs only to be done with half of the current precision as it computes the correcting digits (which alter only the less significant half of the digits). Thus, at each step we have 1.5 multiplications of the current precision: one full precision multiplication for $d\,x_k$ plus a half precision multiplication for $x_k(\dots)$. The total work[1] amounts to $1.5 + 1.5/2 + 1.5/4 + \dots = 1.5 \cdot \sum_{n=0}^{N} \frac{1}{2^n}$ which is less than 3 full precision multiplications. Together with the final multiplication a division costs as much as 4 multiplications.

The numerical example given in figure 15.1 on the next page shows the first steps of the computation of an inverse starting from a two-digit initial approximation.

The achieved precision can be determined by the absolute value of $(1 - d\,x_k)$. In hfloat, when the achieved precision is below a certain limit a third order correction is used to assure maximum precision at the last step:

$$x_{k+1} \quad = \quad x_k + x_k\,(1 - d\,x_k) + x_k\,(1 - d\,x_k)^2 \tag{15.6}$$

One should in general *not* use algebraically equivalent forms like $x_{k+1} = 2\,x_k - d\,x_k^2$ (for the second order iteration) because computationally there is a difference: Cancellation can occur and the information on the achieved precision is not found easily.

[1] The asymptotics of the multiplication is set to $\sim N$ (instead of $N \log(N)$) for the estimates made here, this gives a realistic picture for large N.

$$d := 3.1415926 \tag{15.2a}$$
$$x_0 = 0.31 \quad \text{[initial 2-digit approximation for } 1/d\text{]} \tag{15.2b}$$

$$d \cdot x_0 = 3.141 \cdot 0.3100 = 0.9737 \tag{15.3a}$$
$$y_0 = 1.000 - d \cdot x_0 = 0.02629 \tag{15.3b}$$
$$x_0 \cdot y_0 = 0.3100 \cdot 0.02629 = 0.0081(49) \tag{15.3c}$$
$$x_1 = x_0 + x_0 \cdot y_0 = 0.3100 + 0.0081 = 0.3181 \tag{15.3d}$$

$$d \cdot x_1 = 3.1415926 \cdot 0.31810000 = 0.9993406 \tag{15.4a}$$
$$y_1 = 1.0000000 - d \cdot x_1 = 0.0006594 \tag{15.4b}$$
$$x_1 \cdot y_1 = 0.31810000 \cdot 0.0006594 = 0.0002097(5500) \tag{15.4c}$$
$$x_2 = x_1 + x_1 \cdot y_1 = 0.31810000 + 0.0002097 = 0.31830975 \tag{15.4d}$$

$$d \cdot x_2 = 3.1415926 \cdot 0.31830975 = 0.99999955 \tag{15.5a}$$
$$y_2 = 1.0000000 - d \cdot x_2 = 0.00000014 \tag{15.5b}$$
$$x_2 \cdot y_2 = 0.31830975 \cdot 0.00000014 = 0.000000044 \tag{15.5c}$$
$$x_3 = x_2 + x_2 \cdot y_2 = 0.31830975 + 0.000000044 = 0.31830979399 \tag{15.5d}$$

Figure 15.1: First steps of the computation of the inverse of π.

15.1.2 Inverse square root

Computation of inverse square roots can be done using a similar scheme: Find a starting approximation $x_0 \approx \frac{1}{\sqrt{d}}$ then iterate

$$x_{k+1} = x_k + x_k \frac{(1 - d\,x_k^2)}{2} \tag{15.7}$$

until the desired precision is reached. Convergence is again second order: if $x_k = \frac{1}{\sqrt{d}}(1 + e)$ then

$$x_{k+1} = \frac{1}{\sqrt{d}} \left(1 - \frac{3}{2}e^2 - \frac{1}{2}e^3\right) \tag{15.8}$$

When the achieved precision is below a certain limit a third order correction should be applied:

$$x_{k+1} = x_k + x_k \frac{(1 - d\,x_k^2)}{2} + x_k \frac{3\,(1 - d\,x_k^2)^2}{8} \tag{15.9}$$

To compute the square root first compute $\frac{1}{\sqrt{d}}$ then a final multiply with d gives $\sqrt{d}$.

Similar considerations as above (with squaring considered as expensive as multiplication[2]) give an operation count of 4 multiplications for computing $\frac{1}{\sqrt{d}}$ and 5 for $\sqrt{d}$. Note that this algorithm is considerably better than iterating $x_{k+1} := \frac{1}{2}\,(x_k + \frac{d}{x_k})$ because no long divisions are involved.

A unified routine that implements the computation of the inverse a-th roots is given in [hfloat: src/hf/itiroot.cc]. The general form of the division-less iteration for the a-th root of d is, up to third

[2]Indeed it costs about $\frac{2}{3}$ of a multiplication.

order:

$$x_{k+1} \;=\; x_k \left(1 + \frac{(1 - d\,x_k^a)}{a} + \frac{(1+a)\,(1 - d\,x_k^a)^2}{2\,a^2}\right) \tag{15.10}$$

The initial approximation is obtained using ordinary floating point numbers (type `double`) with special precautions to avoid overflow with exponents that cannot be represented with `double`s. Third order corrections are made whenever the achieved precision falls below a certain limit.

15.1.3 Cube root extraction

We use the relation $d^{1/3} = d\,(d^2)^{-1/3}$, that is we compute the inverse third root of d^2 using the iteration

$$x_{k+1} \;=\; x_k + x_k \frac{(1 - d^2\,x_k^3)}{3} \tag{15.11}$$

and finally multiply with d. Convergence is second order: If $x_k = \frac{1}{\sqrt[3]{d}}(1+e)$ then

$$x_{k+1} \;=\; \frac{1}{\sqrt[3]{d}} \left(1 - 2e^2 - \frac{4}{3}e^3 - \frac{1}{3}e^4\right) \tag{15.12}$$

15.1.4 Improved iteration for the square root

Actually, the 'simple' version of the square root iteration ($x_{k+1} := \frac{1}{2}\,(x_k + \frac{d}{x_k})$) can be used for practical purposes when rewritten as a *coupled iteration* for both $\sqrt{d}$ and its inverse. Using for $\sqrt{d}$ the iteration

$$x_{k+1} \;=\; x_k - \frac{(x_k^2 - d)}{2\,x_k} \tag{15.13}$$

$$\;=\; x_k - v_{k+1}\,\frac{(x_k^2 - d)}{2} \quad \text{where} \quad v \approx 1/x \tag{15.14}$$

and for the auxiliary $v \approx 1/\sqrt{d}$ the iteration

$$v_{k+1} = v_k + v_k\,(1 - x_k\,v_k) \tag{15.15}$$

where one starts with approximations

$$x_0 \;\approx\; \sqrt{d} \tag{15.16}$$

$$v_0 \;\approx\; 1/x_0 \tag{15.17}$$

and the v-iteration step precedes that for x. When carefully implemented this method turns out to be significantly more efficient than the preceding version. [hfloat: src/hf/itsqrt.cc] The idea is due to Schönhage.

15.1.5 A different view on the iterations

Let p be a prime and assume you know the inverse x_0 of a given number d modulo p. With (the iteration for the inverse, relation 15.1 on page 399) $\Phi(x) := x\,(1 + (1 - d\,x))$ the number $x_1 := \Phi(x_0)$ is the inverse of d modulo p^2. Modulo p^2 we know that $x_0\,d \equiv (1 + k\,p)$ so we can write $x_0 \equiv 1/d\,(1 + k\,p)$, thereby

$$\Phi(x_0) \;=\; \Phi\left(\frac{1}{d}\,(1 + k\,p)\right) \;=\; \frac{1}{d}\,\left(1 - k\,p^2\right) \;\equiv\; \frac{1}{d} \bmod p^2 \tag{15.18}$$

The very same computation (with $x_1 = 1/d\,(1 + j\,p^2)$) shows that for $x_2 := \Phi(x_1)$ one has $x_2 \equiv 1/d \bmod p^4$. Each application of $\Phi(.)$ doubles the exponent of the modulus.

The equivalent scheme works for root extraction. An example for the inverse square root: with $p = 17$ and $x_0 = 3$ we have $x_0^2\,2 \equiv 1$ ($x_0 \equiv 1/\sqrt{d} \bmod p$ where $d = 2$). Now use the iteration $\Phi(x) := x\,(1+(1-d\,x)/2)$ to compute $x_1 = \Phi(x_0)$ to obtain $x_1 = -45/2 \equiv 122 \bmod p$ and observe that $x_1^2\,d \equiv 1 \bmod p^2$. Compute $x_2 = \Phi(x_1)$ to obtain $x_2 = -1815665 \equiv 21797 \bmod p^2$ and check that $x_2^2\,d \equiv 1 \bmod p^4$.

We will not go into the details (of the theory of p-adic numbers) but note that pari/gp can work with them:

```
? 1/sqrt(2+O(17^5))
  3 + 7*17 + 7*17^2 + 4*17^3 + 11*17^4 + O(17^5)
\\ Note that 21797 = 3 + 7*17 + 7*17^2 + 4*17^3 + 11*17^4
\\       and   122 = 3 + 7*17
```

Section 1.24 on page 52 describes the case $p = 2$.

15.2 Square root extraction for rationals

A general formula for an k-th order ($k \geq 2$) iteration for $\sqrt{d}$ is

$$\Phi_k(x) \;=\; \sqrt{d}\,\frac{\left(x+\sqrt{d}\right)^k + \left(x-\sqrt{d}\right)^k}{\left(x+\sqrt{d}\right)^k - \left(x-\sqrt{d}\right)^k} = \sqrt{d}\,\frac{\left(p+q\sqrt{d}\right)^k + \left(p-q\sqrt{d}\right)^k}{\left(p+q\sqrt{d}\right)^k - \left(p-q\sqrt{d}\right)^k} \tag{15.19}$$

where $x = p/q$.

All $\sqrt{d}$ vanish when expanded:

$$\Phi_2(x) \;=\; \frac{x^2+d}{2\,x} \;=\; \frac{p^2+d\,q^2}{2\,p\,q} \tag{15.20a}$$

$$\Phi_3(x) \;=\; x\,\frac{x^2+3\,d}{3\,x^2+d} \;=\; \frac{p}{q}\,\frac{p^2+3\,d\,q^2}{3\,p^2+d\,q^2} \tag{15.20b}$$

$$\Phi_4(x) \;=\; \frac{x^4+6\,d\,x^2+d^2}{4\,x^3+4\,d\,x} \;=\; \frac{p^4+6\,d\,p^2\,q^2+d^2\,q^4}{4\,p^3\,q+4\,d\,p\,q^3} \tag{15.20c}$$

$$\Phi_5(x) \;=\; x\,\frac{x^4+10\,d\,x^2+5\,d^2}{5\,x^4+10\,d\,x^2+d^2} \;=\; \frac{p}{q}\,\frac{p^4+10\,d\,p^2\,q^2+5\,d^2\,q^4}{5\,p^4+10\,d\,p^2\,q^2+d^2\,q^4} \tag{15.20d}$$

$$\Phi_k(x) \;=\; x\,\frac{\sum_{j=0}^{\lfloor k/2\rfloor}\binom{k}{2\,j}\,x^{k-2j}\,d^j}{\sum_{j=0}^{\lfloor k/2\rfloor}\binom{k}{2\,j+1}\,x^{k-2j-1}\,d^j} \tag{15.20e}$$

There is a nice expression for the error behavior of the k-th order iteration:

$$\Phi_k(\sqrt{d}\cdot\frac{1+e}{1-e}) \;=\; \sqrt{d}\cdot\frac{1+e^k}{1-e^k} \tag{15.21}$$

An equivalent form of relation 15.19 comes from the theory of continued fractions:

$$\Phi_k(x) \;=\; \sqrt{d}\,\cot\left(k\,\operatorname{arccot}\frac{x}{\sqrt{d}}\right) \tag{15.22}$$

With this relation it is immediate that

$$\Phi_m(\Phi_n(x)) \;=\; \Phi_{mn}(x) \tag{15.23}$$

More iterations via Padé approximants

The iterations can also be obtained using Padé-approximants. Let $P_{[i,j]}(z)$ be the Padé-expansion of $\sqrt{z}$ around $z = 1$ of order $[i, j]$. An iteration of order $i+j+1$ is given by $x\,P_{[i,j]}(\frac{d}{x^2})$. Different combinations of i and j result in alternative iterations:

$$[i,j] \quad \mapsto \quad x\,P_{[i,j]}(\frac{d}{x^2}) \tag{15.24a}$$

$$[1,0] \quad \mapsto \quad \frac{x^2+d}{2x} \tag{15.24b}$$

$$[0,1] \quad \mapsto \quad \frac{2x^3}{3x^2-d} \tag{15.24c}$$

$$[1,1] \quad \mapsto \quad x\,\frac{x^2+3d}{3x^2+d} \tag{15.24d}$$

$$[2,0] \quad \mapsto \quad \frac{3x^4+6dx^2-3d^2}{8x^3} \tag{15.24e}$$

$$[0,2] \quad \mapsto \quad \frac{8x^5}{15x^4-10dx^2+3d^2} \tag{15.24f}$$

Still other forms are obtained by using $\frac{d}{x}\,P_{[i,j]}(\frac{x^2}{d})$:

$$[i,j] \quad \mapsto \quad \frac{d}{x}\,P_{[i,j]}(\frac{x^2}{d}) \tag{15.25a}$$

$$[1,0] \quad \mapsto \quad \frac{x^2+d}{2x} \tag{15.25b}$$

$$[0,1] \quad \mapsto \quad \frac{2d^2}{3dx-x^3} \tag{15.25c}$$

$$[1,1] \quad \mapsto \quad \frac{d\,(d+3x^3)}{x\,(3d+x^2)} \tag{15.25d}$$

$$[2,0] \quad \mapsto \quad \frac{-x^4+6dx^2+3d^2}{8xd} \tag{15.25e}$$

$$[0,2] \quad \mapsto \quad \frac{8d^3}{3x^4-10dx^2+15d^2} \tag{15.25f}$$

Products involving $\Phi_k(y)$ for $d = 1$ *

A product expression involving the rational iterations of order 2^n for $d = 1$ is given by

$$y\,\left(\frac{y^2+1}{2y}\right)^{1/2}\,\left(\frac{y^4+6y^2+1}{4y^3+4y}\right)^{1/4}\,\ldots\,(\Phi_{2^k}(y))^{1/2^k}\,\ldots \quad = \quad \left(\frac{y+1}{2}\right)^2 \tag{15.26}$$

where $\Phi_1(y) = y$. The relation can be deduced from the following relation (given by R.W.Gosper) by setting $x = \operatorname{arccoth}(y)$:

$$\coth(x)\,\coth(2x)^{1/2}\,\coth(4x)^{1/4}\,\ldots\,\coth(2^k x)^{1/2^k}\,\ldots \quad = \quad \frac{\exp(2x)}{4\,\sinh^2(x)} \tag{15.27}$$

An equivalent relation is obtained by setting $y = (1+e)/(1-e)$ and using relation 15.21 on the preceding page:

$$\prod_{k=0}^{\infty}\left(\frac{1+e^{2^k}}{1-e^{2^k}}\right)^{1/2^k} \quad = \quad \frac{1}{(1-e)^2} \tag{15.28}$$

Compare with relation 24.13a on page 635. More generally, one has

$$\prod_{k=0}^{\infty} \left(\frac{\sum_{j=0}^{n-1} e^{j \cdot n^k}}{1 - e^{n^k}} \right)^{1/2^k} = \frac{1}{(1-e)^2} \tag{15.29}$$

For example, with $n = 3$:

$$\prod_{k=0}^{\infty} \left(\frac{1 + e^{3^k} + e^{2 \cdot 3^k}}{1 - e^{3^k}} \right)^{1/2^k} = \frac{1}{(1-e)^2} \tag{15.30}$$

These are valid for $e < 1$. In a similar spirit,

$$\prod_{k=0}^{\infty} \left(\frac{1 + e^{3^k} + e^{2 \cdot 3^k}}{\left(1 - e^{3^k}\right)^2} \right)^{1/3^k} = \frac{1}{(1-e)^3} \tag{15.31}$$

Which generalizes as

$$\prod_{k=0}^{\infty} \left(\frac{\sum_{j=0}^{n-1} e^{j \cdot n^k}}{\left(1 - e^{n^k}\right)^{X-1}} \right)^{1/X^k} = \frac{1}{(1-e)^X} \tag{15.32}$$

Extraction of higher roots for rationals

The Padé idea can be adapted for higher roots: use the expansion of $\sqrt[a]{z}$ around $z = 1$ then $x\, P_{[i,j]}(\frac{d}{x^a})$ produces an order $i + j + 1$ iteration for $\sqrt[a]{z}$. A second order iteration is given by

$$\Phi_2(x) = x + \frac{d - x^a}{a\, x^{a-1}} = \frac{(a-1)\, x^a + d}{a\, x^{a-1}} = \frac{1}{a} \left((a-1)\, x + \frac{d}{x^{a-1}} \right) \tag{15.33}$$

A third order iteration for $\sqrt[a]{d}$ is

$$\Phi_3(x) = x \cdot \frac{\alpha\, x^a + \beta\, d}{\beta\, x^a + \alpha\, d} = \frac{p}{q} \cdot \frac{\alpha\, p^a + \beta\, q^a d}{\beta\, p^a + \alpha\, q^a d} \tag{15.34}$$

where $x = p/q$, $\alpha = a - 1$ and $\beta = a + 1$.

With $1/\sqrt[a]{x}$ and $x\, P_{[i,0]}(x^a\, d)$ division-free iterations for the inverse a-th root of d are obtained, see section 15.3. In section 16.5 on page 425 the Padé idea is pursued for arbitrary functions f.

15.3 Iterations for the inverse a-th root

There is a nice general formula that allows to build iterations with arbitrary order of convergence for $\sqrt[a]{d} = d^{-1/a}$ that involve no long division.

One uses the identity

$$d^{-1/a} = x\, (1 - (1 - x^a\, d))^{-1/a} \tag{15.35}$$

$$= x\, (1-y)^{-1/a} \quad \text{where} \quad y := (1 - x^a\, d) \tag{15.36}$$

Taylor expansion gives

$$d^{-1/a} = x \sum_{k=0}^{\infty} (1/a)^{\overline{k}}\, y^k \tag{15.37}$$

where $z^{\overline{k}} := z\,(z+1)\,(z+2)\,\ldots\,(z+k-1)$ (and $z^{\overline{0}} := 1$, $z^{\overline{k}}$ is called the rising factorial power). Written out:

$$d^{-1/a} = x\,\frac{1}{\sqrt[a]{1-y}} = x\left(1+\frac{y}{a}+\frac{(1+a)\,y^2}{2\,a^2}+\frac{(1+a)(1+2a)\,y^3}{6\,a^3}+ + \frac{(1+a)(1+2a)(1+3a)\,y^4}{24\,a^4}+\cdots+\frac{\prod_{k=1}^{n-1}(1+k\,a)}{n!\,a^n}\,y^n+\ldots\right) \tag{15.38}$$

A n-th order iteration for $d^{-1/a}$ is obtained by truncating the above series after the $(n-1)$-th term:

$$\Phi_n(x) := x\sum_{k=0}^{n-1}(1/a)^{\overline{k}}\,y^k \tag{15.39}$$

$$x_{k+1} = \Phi_n(x_k) \tag{15.40}$$

Convergence is n-th order:

$$\Phi_n(d^{-1/a}(1+e)) = d^{-1/a}(1+O(e^n)) \tag{15.41}$$

For example, the second order iteration is

$$\Phi_2(x) := x + x\frac{(1-dx^a)}{a} \tag{15.42}$$

Convergence is indeed quadratic: If $x = \frac{1}{\sqrt[a]{d}}\,(1+e)$ then

$$\Phi_2(x) = \frac{1}{\sqrt[a]{d}}\left((1+e)\left[(1+e)^a-(a+1)\right]\right) \tag{15.43}$$

$$= \frac{1}{\sqrt[a]{d}}\left(1-\frac{a+1}{2}e^2+O(e^3)\right) \tag{15.44}$$

Iterations for the inverse

Set $a=1$, $y=1-d\,x$ to compute the inverse of d.

$$\frac{1}{d} = x\,\frac{1}{1-y} \tag{15.45a}$$

$$\Phi_k(x) = x\left(1+y+y^2+y^3+y^4+\cdots+y^{k-1}\right) \tag{15.45b}$$

$\Phi_2(x) = x\,(1+y)$ is the second order iteration 15.1 on page 399.

Composition is particularly simple with the iterations for the inverse:

$$\Phi_{n\,m} = \Phi_n(\Phi_m) \tag{15.46}$$

There are simple closed forms for this iteration:

$$\Phi_k = \frac{1-y^k}{d} = x\,\frac{1-y^k}{1-y} \tag{15.47a}$$

$$\Phi_\infty = 1+x+x^2+x^3+x^4+\ldots \tag{15.47b}$$

$$= x\,(1+y)\,(1+y^2)\,(1+y^4)\,(1+y^8)\,\ldots \tag{15.47c}$$

$$= x\,(1+y+y^2)\,(1+y^3+y^6)\,(1+y^9+y^{18})\,\ldots \tag{15.47d}$$

The expression for the convergence of the k-th order iteration is

$$\Phi_k\left(\frac{1}{d}\,(1+e)\right) = \frac{1}{d}\left(1-(-e)^k\right) \tag{15.48}$$

The iteration converges if one has $|e| < 1$ for the start value $x_0 = \frac{1}{d}(1+e)$. That is, the basin of attraction is the open disc of radius $r = 1/d$ around the point $1/d$, independent of the order k. For other iterations, the basin of attraction usually has a fractal boundary and further depends on the order.

Iterations for the inverse square root

Set $a = 2$, $y = 1 - d\,x^2$ to compute the inverse square root of d.

$$\frac{1}{\sqrt{d}} = x\,\frac{1}{\sqrt{1-y}} \tag{15.49a}$$

$$= x\left(1 + \frac{y}{2} + \frac{3\,y^2}{8} + \frac{5\,y^3}{16} + \frac{35\,y^4}{128} + \cdots + \frac{\binom{2k}{k}\,y^k}{4^k} + \ldots\right) \tag{15.49b}$$

$$\Phi_{k+1}(x) = x\left(1 + \frac{y}{2} + \frac{3\,y^2}{8} + \ldots + \frac{\binom{2k}{k}\,y^k}{4^k}\right) \tag{15.49c}$$

$\Phi_2(x) = x\,(1 + y/2)$ is the second order iteration 15.7 on page 400.

Computation of the a-th root

The obvious way to compute the a-root is to use the iteration for the inverse root and invert afterwards. Another possibility is to compute the inverse a-th root of d^{a-1} and multiply with d afterwards:

$$\left[(d)^{a-1}\right]^{-1/a} d = d^{(1-a)/a}\,d = d^{1/a} \tag{15.50}$$

If a is small the cost is lower than with the final iteration for the inverse (which costs about 3 Multiplications or 9 FFTs). The powering-method is not more expensive than inversion if the rightmost column in figure 14.1 on page 397 for $e = a - 1$ is less or equal 6. In case the iteration for the inverse involves a loss of precision the method might be preferred also if its cost is higher.

Error expressions for inverse square root iterations *

An expression for the error behavior of the n-th order iteration similar to 'nice' formula 15.21 on page 402 is

$$F_n := \Phi_n(d^{-1/2}\frac{1+e}{1-e})/d^{-1/2} \tag{15.51a}$$

$$= \frac{\sum_{k=0}^{n}\binom{2n-1}{k}(-e)^k - \sum_{k=n+1}^{2n+1}\binom{2n-1}{k}(-e)^k}{(1-e)^{2n-1}} \tag{15.51b}$$

Now define $c := \frac{F_n - 1}{F_n + 1}$, then

$$F_n = \frac{1+c}{1-c} \quad \text{where} \quad c = e^n\,\frac{\sum_{k=0}^{n}\binom{2n-1}{n-k}(-e)^k}{\sum_{k=0}^{n}\binom{2n-1}{k}(-e)^k} \tag{15.51c}$$

For example, with $n = 2$ we obtain

$$\begin{aligned} F_2 &:= \Phi_2(d^{-1/2}\frac{1+e}{1-e}) \,/\, d^{-1/2} && (15.52a)\\ &= \frac{1-3\,e-3\,e^2+e^3}{1-3\,e+3\,e^2-e^3} && (15.52b)\\ &= 1-6\,e^2-16\,e^3-30\,e^4-48\,e^5-70\,e^6-\ldots && (15.52c)\\ &= -1+\frac{6}{(e-1)^2}+\frac{4}{(e-1)^3} && (15.52d)\\ F_2 &= \frac{1+c}{1-c} \quad \text{where} \quad c = e^2\,\frac{e-3}{1-3\,e} && (15.52e) \end{aligned}$$

The coefficients of the Taylor expansion of F_n in e are always integers.

For $n = 4$:

$$\begin{aligned} F_4 &:= \Phi_4(d^{-1/2}\frac{1+e}{1-e}) \,/\, d^{-1/2} && (15.53a)\\ &= \frac{1-7\,e+21\,e^2-35\,e^3-35\,e^4+21\,e^5-7\,e^6+e^7}{1-7\,e+21\,e^2-35\,e^3+35\,e^4-21\,e^5+7\,e^6-e^7} && (15.53b)\\ &= 1-70\,e^4-448\,e^5-1680\,e^6-4800\,e^7-11550\,e^8-\ldots && (15.53c)\\ &= -1+\frac{70}{(e-1)^4}+\frac{168}{(e-1)^5}+\frac{140}{(e-1)^6}+\frac{40}{(e-1)^7} && (15.53d)\\ F_4 &= \frac{1+c}{1-c} \quad \text{where} \quad c = e^4\,\frac{e^3-7\,e^2+21\,e-35}{1-7\,e+21\,e^2-35\,e^3} && (15.53e) \end{aligned}$$

Two curious formulas related to the error behavior of Φ_2 are

$$\Phi_2\left(\frac{1}{\sqrt{d}}\left[e+\frac{1}{e}\right]\right) = \frac{1}{\sqrt{d}}\left[-\frac{1}{2}\cdot\left(e^3+\frac{1}{e^3}\right)\right] \quad (15.54)$$

$$\Phi_2\left(\frac{1}{\sqrt{d}}\left[e-\frac{2}{3}\,\frac{1}{e}\right]\right) = \frac{1}{\sqrt{d}}\left[\frac{1}{2}\cdot\left(e^3-\frac{2}{3}\,\frac{1}{e^3}\right)\right] \quad (15.55)$$

15.4 Initial approximations for iterations

With the iterative schemes one alway needs an initial approximation for the root that is to be computed. Assume we want to compute $f(d)$, for example, $f(d) = \sqrt{d}$ or $f(d) = \exp(d)$. One can convert the high precision number d to a machine floating point number and use the FPU to compute an initial approximation. However, when d cannot be represented with a machine float, the method fails. The method will also fail if the result causes an overflow, which is likely to happen with $f(d) = \exp(d)$.

15.4.1 Inverse roots

With $f(d) = d^{1/a}$ one can use the following technique. Write d in the form

$$d = M \cdot R^X$$

were M is the mantissa, R the radix and X the exponent. We have $0 \le M < 1$ and $X \in \mathbb{Z}$. Now use

$$d^{1/a} = M^{1/a} \cdot R^{X/a} = M^{1/a} \cdot R^{Y/a} \cdot R^Z \quad (15.56)$$

where $Z = \lfloor X/a \rfloor$ and $Y = X - a \cdot Z$ (so $X = a \cdot Z + Y$).

The algorithm computes the three quantities in relation 15.56 separately and finally computes the product as result. A C++ implementation is given in [hfloat: src/hf/itiroot.cc]:

```
void
approx_invpow(const hfloat &d, hfloat &c, long a)
{
    double dd;
    dt_mantissa_to_double(*(d.data()), dd);
    dd = pow(dd, 1.0/(double)a);  // M^(1/a)

    long  Z = d.exp() / a;    // Z = X / a
    long  Y = d.exp() - a*Z;  // Y = X % a

    double tt = pow((double)d.radix(),(double)Y/a);  // R^(Y/a)
    dd *= tt;         // M^(1/a) * R^(Y/a)

    d2hfloat(dd, c);  // c =  M^(1/a) * R^(Y/a)
    c.exp( c.exp()+Z );   // c *= R^(Z)
}
```

One can also subtract $a \cdot Z$ from the exponent before the iteration and add Z to the exponent afterwards: $\left(R^{X-a\,Z}\right)^{1/a} = R^{Y/a} = R^{X/a}/R^{Z}$. In that case the initial approximation can be computed via the straight forward approach.

15.4.2 Exponential function

With $f(d) = \exp(d)$ write

$$\exp(d) \quad = \quad M \cdot R^X$$

Then use $X = \lfloor d/\log(R) \rfloor$ and $M = \exp(d - X \cdot \log R)$. Note that d must fit into a machine float which is not a real restriction as the $\exp(d)$ will not fit into a `hfloat` already with smaller values.

A C++ implementation is given in [hfloat: src/tz/itexp.cc]:

```
void
approx_exp(const hfloat &d, hfloat &c)
{
    double dd;
    hfloat2d(d,dd);

    double lr = log( hfloat::radix() );
    double X = floor( dd/lr );
    double M = exp( dd-X*lr );

    d2hfloat(M,c);
    c.exp( c.exp()+(long)X );
}
```

The iteration for computation of the exponential function is given in section 17.3.

15.5 Some applications of the matrix square root

15.5.1 Re-orthogonalization

A task from graphics applications: a rotation matrix A that deviates from being orthogonal[3] shall be transformed to the closest orthogonal matrix E. One has (see [90])

$$E \quad = \quad A\,(A^T A)^{-\frac{1}{2}} \tag{15.57}$$

With the division-free iteration for the inverse square root

$$\Phi(x) \quad = \quad x\left(1 + \frac{1}{2}\left(1 - dx^2\right) + \frac{3}{8}\left(1 - dx^2\right)^2 + \frac{5}{16}\left(1 - dx^2\right)^3 + \ldots\right) \tag{15.58}$$

[3]Typically due to cumulative errors resulting from many multiplications with rotation matrices.

at hand the given task is pretty easy: As $A^T A$ is close to unity (the identity matrix) we can use the (second order) iteration with $d = A^T A$ and $x = 1$

$$(A^T A)^{-\frac{1}{2}} \approx \left(1 + \frac{1 - A^T A}{2}\right) \tag{15.59}$$

and multiply by A to get a 'closer-to-orthogonal' matrix A_+:

$$A_+ = A\left(1 + \frac{1 - A^T A}{2}\right) \approx E \tag{15.60}$$

The step can be repeated with A_+ (or higher orders can be used) if necessary. Note the identical equation would be obtained when trying to compute the inverse square root of 1:

$$x_+ = x\left(1 + \frac{1 - x^2}{2}\right) \to 1 \tag{15.61}$$

It is instructive to write things down in the *singular value decomposition* (SVD) representation

$$A = U\,\Omega\,V^T \tag{15.62}$$

where U and V are orthogonal and Ω is a diagonal matrix with non-negative entries (see [108]). The SVD is a decomposition of the action of the matrix as: rotation – element wise stretching – rotation.

Now

$$A^T A = \left(V\Omega U^T\right)\,\left(U\Omega V^T\right) = V\Omega^2 V^T \tag{15.63}$$

and positive exponents of A are 'absorbed' as powers of Ω

$$\left(U\Omega V^T\right)^n = U\Omega^n V^T \tag{15.64}$$

while for negative exponents

$$\left(U\Omega V^T\right)^{-n} = V\Omega^{-n} U^T \tag{15.65}$$

Thereby

$$(A^T A)^{-\frac{1}{2}} = \left(\left(V\Omega U^T\right)\left(U\Omega V^T\right)\right)^{-\frac{1}{2}} = \left(V\Omega^2 V^T\right)^{-\frac{1}{2}} = V\Omega^{-1}V^T \tag{15.66}$$

and we have

$$A\,(A^T A)^{-\frac{1}{2}} = \left(U\Omega V^T\right)\,\left(V\Omega^{-1}V^T\right) = U\,V^T \tag{15.67}$$

that is, the 'stretching part' was removed.

A numerical example: Let

$$A = \begin{bmatrix} +1.0000000 & +1.0000000 & +0.7500000 \\ -0.5000000 & +1.5000000 & +1.0000000 \\ +0.7500000 & +0.5000000 & -1.0000000 \end{bmatrix} \tag{15.68}$$

then

$$E = \begin{bmatrix} +0.803114165 & +0.291073143 & +0.519888513 \\ -0.486897253 & +0.823533541 & +0.291073143 \\ +0.343422053 & +0.486897252 & -0.803114166 \end{bmatrix} \tag{15.69}$$

and $E\,E^T = 1$.

15.5.2 Polar decomposition

The so-called *polar decomposition* of a matrix A is a representation of the form

$$A \;=\; E\,R \tag{15.70}$$

where the matrix E is orthogonal and $R = R^T$. It is an analogue to the representation of a complex number $z \in \mathbb{C}$ as $z = e^{i\,\phi}\,r$ (identify $R \sim r$ and $E \sim e^{i\,\phi}$).

The polar decomposition can be defined by

$$A \;=\; E\,R \;:=\; \Big(A(A^T A)^{-1/2}\Big)\,\Big((A^T A)^{1/2}\Big) \tag{15.71}$$

where R is the (unique, positive semidefinite) square root $R = (A^T A)^{1/2}$ and $E = A(A^T A)^{-1/2}$.

The matrix E is computed as before:

$$E \;=\; A \cdot \left(1 + \frac{1 - A^T A}{2}\right) \cdot \left(1 + \frac{1 - A_+^T A_+}{2}\right) \cdot \ldots \tag{15.72}$$

The matrix R equals $E^{-1}A = E^T A$, that is

$$\begin{aligned} A &= E\,R \;=\; E\,\big(E^T A\big) && (15.73)\\ &= U\,V^T\,\big(V\,\Omega\,V^T\big) && (15.74)\end{aligned}$$

Compute the polar decomposition as

$$\begin{aligned} E_0 &= A && (15.75a)\\ Y_k &= \left(1 + \frac{1 - E_k^T\,E_k}{2}\right) && (15.75b)\\ E_{k+1} &= E_k\,Y_k \quad \to E && (15.75c)\\ R_{k+1} &= E_{k+1}^T A \quad \to R && (15.75d)\\ E_{k+1}\,R_{k+1} &\to A && (15.75e)\end{aligned}$$

Higher orders can be added in the computation of Y_k. If you prefer $z = r\,e^{i\,\phi}$ over $e^{i\,\phi}\,r$ then iterate as above but set $R' = A\,E^T$ so that

$$\begin{aligned} A &= R'\,E \;=\; \big(AE^T\big)\,E && (15.76)\\ &= \big(U\,\Omega\,U^T\big)\,U\,V^T && (15.77)\end{aligned}$$

Numerical example: Let

$$A \;=\; \begin{bmatrix} +1.00000 & +1.00000 & +0.75000 \\ -0.50000 & +1.50000 & +1.00000 \\ +0.75000 & +0.50000 & -1.00000 \end{bmatrix} \tag{15.78}$$

then

$$\begin{aligned} A &= E\,R && (15.79a)\\ &= \begin{bmatrix} +0.80311 & +0.29107 & +0.51988 \\ -0.48689 & +0.82353 & +0.29107 \\ +0.34342 & +0.48689 & -0.80311 \end{bmatrix} \begin{bmatrix} +1.30412 & +0.24447 & -0.22798 \\ +0.24447 & +1.76982 & +0.55494 \\ -0.22798 & +0.55494 & +1.48410 \end{bmatrix} && (15.79b)\\ A &= R'\,E && (15.79c)\\ &= \begin{bmatrix} +1.48410 & +0.55494 & +0.22798 \\ +0.55494 & +1.76982 & -0.24447 \\ +0.22798 & -0.24447 & +1.30412 \end{bmatrix} \begin{bmatrix} +0.80311 & +0.29107 & +0.51988 \\ -0.48689 & +0.82353 & +0.29107 \\ +0.34342 & +0.48689 & -0.80311 \end{bmatrix} && (15.79d)\end{aligned}$$

15.5.3 Sign decomposition

The *sign decomposition* can be defined as

$$A \;=\; S\,N \;=\; \left(A(A^2)^{-1/2}\right)\,\left((A^2)^{1/2}\right) \tag{15.80}$$

where $N = (A^2)^{-1/2}$ and $S = A\,(A^2)^{-1/2}$. The square root has to be chosen such that all its eigenvalues have positive real parts. The sign decomposition is undefined if A has eigenvalues on the imaginary axis. The matrix S is its own inverse (its eigenvalues are ± 1). The matrices A, S and N commute pairwise: $SN = NS$, $AN = NA$ and $AS = SA$.

Use

$$\begin{aligned} S_0 &= A && (15.81a)\\ Y_k &= \left(1 + \frac{1 - S_k^2}{2}\right) && (15.81b)\\ S_{k+1} &= S_k\,Y_k \quad \to S && (15.81c)\\ N_{k+1} &= S_{k+1}\,A \quad \to N && (15.81d) \end{aligned}$$

Numerical example: Let

$$A \;=\; \begin{bmatrix} +1.00000 & +1.00000 & +0.75000 \\ -0.50000 & +1.50000 & +1.00000 \\ +0.75000 & +0.50000 & -1.00000 \end{bmatrix} \tag{15.82}$$

then

$$\begin{aligned} A &= S\,N && (15.83a)\\ &= \begin{bmatrix} +0.90071 & -0.01706 & +0.29453 \\ -0.24065 & +0.95862 & +0.71389 \\ +0.62679 & +0.10775 & -0.85933 \end{bmatrix} \begin{bmatrix} +1.13014 & +1.02237 & +0.36392 \\ -0.18454 & +1.55423 & +0.06423 \\ -0.07158 & +0.35875 & +1.43718 \end{bmatrix} && (15.83b) \end{aligned}$$

where $S\,S = 1$.

See [59] and also [60].

15.5.4 Pseudo-inverse

While we are at it: Define a matrix A^+ as

$$A^+ \;:=\; (AA^T)^{-1}A^T = \left(V\Omega^{-2}V^T\right)\,\left(V\Omega U^T\right) = V\Omega^{-1}U^T \tag{15.84}$$

This looks suspiciously like the inverse of A. In fact, this is the *pseudo-inverse* of A:

$$A^+\,A \;=\; \left(V\,\Omega^{-1}\,U^T\right)\,\left(U\,\Omega\,V^T\right) = 1 \quad \text{but wait} \tag{15.85}$$

A^+ has the nice property to exist even if A^{-1} does not. If A^{-1} exists, it is identical to A^+. If not, $A^+\,A \neq 1$ but A^+ will give the best possible (in a least-square sense) solution $x^+ = A^+b$ of the equation $A\,x = b$ (see [Int2Alg], p.770). To find $(A\,A^T)^{-1}$ use the iteration for the inverse:

$$\Phi(x) = x\,\left(1 + (1 - dx) + (1 - dx)^2 + \dots\right) \tag{15.86}$$

with $d = A\,A^T$ and the start value $x_0 = 2 - n\,(A\,A^T)/\,||A\,A^T||^2$ where n is the dimension of A.

A pari/gp implementation of the pseudo-inverse using the SVD:

```
matpseudoinv(A)=
\\ Return pseudo-inverse of A
{
    local(t, x, U, d, V);
    t = matSVD(A);
    U = t[1];  d = t[2];  V = t[3];
    for (k=1, matsize(d)[1],
        x=d[k,k];  if (x>1e-15, d[k,k]=1/x, d[k,k]=0);
    );
    return( V*d*U~ );
}
```

Where the SVD is computed with the help of a routine (`qfjacobi()`) that returns the eigenvectors of a real symmetric matrix:

```
matSVDcore(A)=
\\ Singular value decomposition:
\\ Return [U, d, V] so that U*d*V~==A
\\ d is a diagonal matrix
\\ U, V are orthogonal
{
    local(U, d, V);  \\ returned quantities
    local(t, R, d1);
    R = conj(A~)*A;  \\ R==V*d^2*V~
    t = qfjacobi( R );  \\ fails with eigenvalues==zero
    V = t[2];
    d = real(sqrt(t[1]));
    d1 = d;
    for (k=1, length(d1), t=d1[k]; if (abs(t)>1e-16, t=1/t, t=0); d1[k]=t );
    d1 = matdiagonal(d1):
    d = matdiagonal(d);
    U = (A*V*d1);
    return( [U, d, V] );
}
```

The core routine is always called with a matrix A whose number of rows is greater or equal to its number of rows. Thereby we make sure that when the main routine with argument A computes U, d, V so that $A = U\,d\,V^T$ then with argument A^T we obtain X, d, Y where $A^T = X\,d\,Y^T$, $X = V$ and $Y = U$.

```
matSVD(A)=
{
    local(tq, t, U, d, V);
    t = matsize(A);
    tq=0;  if ( t[1]<t[2],  tq=1; A=A~; );
    t = matSVDcore(A);
    d = t[2];
    if ( tq,
        U = t[3];  V=t[1];
    , /* else */
        U = t[1];  V=t[3];
    );
    return( [U, d, V] );
}
```

For a numerical example see figure 15.2 on the next page. The connection between the SVD of a matrix a and the eigenvectors of $A^T\,A$ is described in [67].

15.6 Goldschmidt's algorithm

A framework for the so-called *Goldschmidt algorithm* can be stated as follows. Initialize

$$x_0 = d^A \tag{15.87a}$$
$$E_0 = d^B \tag{15.87b}$$

```
? A
  [+1.00 +1.00 +0.75 +2.00]
  [-0.50 +1.50 +1.00 +3.00]
  [+0.75 +0.50 -1.00 -3.00]

? t=matSVD(A); U=t[1]; d=t[2]; V=t[3];
? U
  [+0.644401153492 +0.438818890 +0.6262468643]
  [-0.695372132379 +0.676976586 +0.2411644651]
  [-0.318126941467 -0.590881276 +0.7413869203]

? d
  [+0.95641003 0 0]
  [0 +5.09161169 0]
  [0 0 +1.74234618]

? V
  [+0.787833655771 -0.067332385426 +0.60935354299]
  [-0.583139548860 +0.227598489665 +0.77980313700]
  [+0.110889336609 +0.313647647860 -0.01752654388]
  [+0.164225309908 +0.919396775264 -0.14243646797]

? Ax=matpseudoinv(A)
  [+0.744034618880 -0.497415792829 +0.005046325813]
  [-0.093004327360 +0.562176974103 +0.499369209273]
  [+0.095446097914 -0.041347314730 -0.080741213017]
  [+0.138692567521 -0.016875347613 -0.221929812661]

? A*Ax
  [+1.0000000000000 +3.78653234 E-29 -4.41762106 E-29]
  [+2.52435489 E-29 +1.0000000000000 -2.52435489 E-29]
  [-2.52435489 E-29 -2.52435489 E-29 +1.0000000000000]

? Ax*A
  [+0.9965272596551 +0.0004340925431 +0.0555638455173 -0.0193171181681]
  [+0.0004340925431 +0.9999457384321 -0.0069454806896 +0.0024146397710]
  [+0.0555638455173 -0.0069454806896 +0.1109784717229 +0.3090738906900]
  [-0.0193171181681 +0.0024146397710 +0.3090738906900 +0.8925485301897]
```

Figure 15.2: Numerical example for the pseudo-inverse computed by the SVD. We use a 3×4 matrix which is definitely not invertible. A working precision of 25 decimal digits was used, so $A\,A^+ = 1$ to within that precision. On the other hand, $A^+\,A$ is not close to the unit matrix.

---

then iterate

$$P_{k+1} = 1 + \frac{1 - E_k}{a} \tag{15.87c}$$

$$x_{k+1} = x_k\,P_k^b \quad \to d^{A - B\,b/a} \tag{15.87d}$$

$$E_{k+1} = E_k\,P_k^a \quad \to 1 \tag{15.87e}$$

The algorithm converges quadratically. The updates for x and E (last two relations) can be computed independently. The iteration is not self-correcting, so the computations have to be carried out with full precision in all steps.

An invariant of the algorithm is given by x_k^a / E_k^b:

$$\frac{x_{k+1}^a}{E_{k+1}^b} = \frac{(x_k \cdot P_k^b)^a}{(E_k \cdot P_k^a)^b} = \frac{x_k^a}{E_k^b} \tag{15.88a}$$

Using

$$\frac{x_0^a}{E_0^b} = \frac{d^{A\,a}}{d^{B\,b}} = d^{A\,a - B\,b} \tag{15.88b}$$

and, as E converges to 1, we find that

$$\frac{x_\infty^a}{E_\infty^b} = x_\infty^a = \frac{x_0^a}{E_0^b} \tag{15.88c}$$

That is,

$$x_\infty \;=\; \left(\frac{x_0^a}{E_0^b}\right)^{1/a} \;=\; \frac{x_0}{E_0^{b/a}} \;=\; \frac{d^A}{d^{b\,B/a}} \;=\; d^{A-B\,b/a} \tag{15.88d}$$

One can now look for interesting special cases, b is set to one in what follows.

15.6.1 An algorithm for the a-th root

Solving $A - B/a = 1/a$ gives $B = Aa - 1$ and especially $A = 0,\ B = 1$. That is, set

$$x_0 \;=\; d \tag{15.89a}$$
$$E_0 \;=\; d^{a-1} \tag{15.89b}$$

then iterate:

$$P_k \;:=\; 1 + \frac{1 - E_k}{a} \quad \to 1 \tag{15.89c}$$
$$x_{k+1} \;:=\; x_k \cdot P_k \tag{15.89d}$$
$$E_{k+1} \;:=\; E_k \cdot P_k^a \quad \to 1 \tag{15.89e}$$

until x close enough to

$$x_\infty \;=\; d^{\frac{1}{a}} \tag{15.89f}$$

Computation of the square root

For $a = 2$ one gets an algorithm for the computation of the square root:

$$\sqrt{d} \;=\; d \prod_{k=0}^{\infty} \frac{3 - E_k}{2} \tag{15.90}$$

where $E_0 = d$, $E_{k+1} := E_k \,(\frac{3-E_k}{2})^2$.

15.6.2 An algorithm for the inverse a-th root

Solving $A - B/a = -1/a$ gives $B = Aa + 1$ and especially $A = 1,\ B = a - 1$. That is, set

$$x_0 \;=\; 1 \tag{15.91a}$$
$$E_0 \;=\; d \tag{15.91b}$$

then iterate as in formulas 15.89c..15.89e until x close enough to

$$x_\infty \;=\; d^{-\frac{1}{a}} \tag{15.91c}$$

Computation of the inverse

With $a = 1$ one gets an algorithm for the inverse ($P_k = 1 + (1 - E_k) = 2 - E_k$, so):

$$\frac{1}{d} \;=\; \prod_{k=0}^{\infty} (2 - E_k) \tag{15.92}$$

where $E_0 = d$, $E_{k+1} := E_k \,(2 - E_k)$.

Computation of the inverse square root

For $a = 2$ one gets an algorithm for the inverse square root ($P_k = 1 + (1 - E_k)/2 = (3 - E_k)/2$):

$$\frac{1}{\sqrt{d}} \quad = \quad \prod_{k=0}^{\infty} \frac{3 - E_k}{2} \tag{15.93}$$

where $E_0 = d$, $E_{k+1} := E_k \, (\frac{3-E_k}{2})^2$. See also [96].

15.6.3 Higher order algorithms for the inverse a-th root

Higher order iterations are obtained by appending higher terms to the expression $\left(1 + \frac{1-E_k}{a}\right)$ in the definitions of P_{k+1} as suggested by equation 15.38 and the identification $y = 1 - E$:

$$\begin{aligned} E_{k+1} \quad = \quad & E_k \left[1 + \frac{1 - E_k}{a} \qquad \text{[second order]} \right. \\ & + \frac{(1+a)\,(1-E_k)^2}{2\,a^2} \qquad \text{[third order]} \\ & + \frac{(1+a)\,(1+2a)\,(1-E_k)^3}{6\,a^3} \qquad \text{[fourth order]} \\ & + \ \ldots \ + \\ & \left. + \ \frac{(1+a)\,(1+2a)\ldots(1+(n-1)\,a)\,(1-E_k)^n}{n!\,a^n} \right]^a \qquad \text{[order } (n+1)\text{]} \end{aligned} \tag{15.94}$$

As an example, the inverse fourth root of $d = 2$ can be obtained via the third order algorithm

$$x_0 \quad = \quad 1 \tag{15.95a}$$

$$E_0 \quad = \quad d \ = \ 2 \tag{15.95b}$$

$$E_{k+1} \quad = \quad E_k \left(\frac{45 - 18\,E_k + 5\,E_k^2}{32} \right)^4 \ =: \ E_k \, P_k^4 \tag{15.95c}$$

$$x_{k+1} \quad = \quad x_k \, P_k \tag{15.95d}$$

Figure 15.3 on the following page shows the numerical values of x_k, E_k and P_k up to step $k = 6$. The approximate precision in bits of x_k is computed as $b_k = -\frac{\log(|1-E_k|)}{\log(2)}$.

15.7 Products for the a-th root

Rewrite the well-known product form

$$\frac{1}{1-y} \quad = \quad (1+y)\,(1+y^2)\,(1+y^4)\,(1+y^8)\,\ldots \tag{15.96}$$

as

$$\frac{1}{1-y} \quad = \quad \prod_{k>0} (1 + Y_k) \quad \text{where} \tag{15.97a}$$

$$Y_1 \quad := \quad y \qquad Y_{k+1} \ := \ Y_k^2 \tag{15.97b}$$

We give product forms for a-th roots and their inverses that generalize the relations above.

$x_0 = 1$ $E_0 = 2.0$ $P_0 = 0.90625$ $b_0 = 0.0$
$x_1 = 0.90625$ $E_1 = 1.3490314483642578125$ $P_1 = 0.93177697415069360431516543030738830566406 25$ $b_1 = 1.5185$
$x_2 = 0.84442288282406607891061867121607065200805664 0625$ $E_2 = 1.01688061936626457410433320872039209492188542219092968$ $P_2 = 0.99582436942565084187883150540345532399967 18905629355821$ $b_2 = 5.8884$
$x_3 = 0.84089688481686585202128466050063877105975287 18627830956$ $E_3 = 1.00000223363323283559583877024921007574068879685671957$ $P_3 = 0.99999944159247134069773211917093099758090030134706071 62$ $b_3 = 18.772$
$x_4 = 0.84089641525371454412926831199731186378490804857313364 97$ $E_4 = 1.00000000000000000052236770943197147970437318824842246 37$ $P_4 = 0.99999999999999999869408072642007130500262990214776549 07$ $b_4 = 57.409$
$x_5 = 0.84089641525371454303112547623321489504003426235678452 49$ $E_5 = 1.00000000000000000000000000000000000000000000000000000 067$ $P_5 = 0.99999999999999999999999999999999999999999999999999999 9833$ $b_5 = 173.32$
$1/\sqrt[4]{2} = 0.84089641525371454303112547623321489504003426235678451 08\ldots$

Figure 15.3: Numerical quantities occurring in the computation of $1/\sqrt[4]{2}$ using a third order Goldschmidt algorithm.

15.7.1 Second order products

Products for the square root and its inverse

For the inverse square root:

$$\frac{1}{\sqrt{1-y}} = \prod_{k>0}(1+Y_k) \quad \text{where} \tag{15.98a}$$

$$Y_1 := \frac{y}{2} \qquad Y_{k+1} := Y_k^2\left(\frac{3}{2}+Y_k\right) \tag{15.98b}$$

For the square root:

$$\sqrt{1-y} = \prod_{k>0}(1+Y_k) \quad \text{where} \tag{15.99a}$$

$$Y_1 := -\frac{y}{2} \qquad Y_{k+1} := -\frac{1}{2}\left(\frac{Y_k}{1+Y_k}\right)^2 \tag{15.99b}$$

Products for the a-th root and its inverse

The relation for the inverse a-th root is

$$\frac{1}{\sqrt[a]{1-y}} \;=\; (1-y)^{-1/a} \;=\; \prod_{k>0}(1+Y_k) \quad \text{where} \tag{15.100a}$$

$$Y_1 \;:=\; \frac{y}{a} \qquad Y_{k+1} \;:=\; \frac{1}{a}\left(1-(1-a\,Y_k)\,(Y_k+1)^a\right) \tag{15.100b}$$

Alternatively,

$$\frac{1}{\sqrt[a]{d}} \;=\; x\,(1-y)^{-1/a} \;=\; x\,\prod_{k>0}(1+Y_k) \tag{15.101}$$

with $y := 1-d\,x^a$ and the definitions 15.100b for Y_k.

For the a-th root:

$$\sqrt[a]{1-y} \;=\; (1-y)^{1/a} \;=\; \prod_{k>0}(1+Y_k) \quad \text{where} \tag{15.102a}$$

$$Y_1 \;:=\; \frac{y}{a} \qquad Y_{k+1} \;:=\; \frac{1}{a}\,\frac{(1+a\,Y_k)-(1+Y_k)^a}{(1+Y_k)^a} \tag{15.102b}$$

15.7.2 Products of arbitrary order

We want to obtain an n-th order product for the inverse a-th root

$$\frac{1}{\sqrt[a]{1-y}} \;=\; \prod_{k>0}(1+T(Y_k)) \quad \text{where} \tag{15.103a}$$

$$Y_1 \;:=\; y \qquad Y_{k+1} \;=\; N(Y_k) \tag{15.103b}$$

The functions $T(.)$ and $N(.)$ have to be determined. Set

$$[1-y]^{-\frac{1}{a}} \;=\; (1+T(Y_1))\,\left[(1+T(Y_1))^a\,(1-y)\right]^{-\frac{1}{a}} \tag{15.104a}$$

$$=:\; (1+T(Y_1))\,[1-Y_2]^{-\frac{1}{a}} \tag{15.104b}$$

where $1+T(Y_1)$ is the Taylor expansion

$$[1-y]^{-\frac{1}{a}} \;=\; 1+\frac{y}{a}+\frac{(1+a)\,y^2}{2\,a^2}+\frac{(1+a)(1+2a)\,y^3}{6\,a^3}+\ldots+\frac{\prod_{k=1}^{n-1}(1+k\,a)}{n!\,a^n}\,y^n+\ldots \tag{15.105}$$

up to order $n-1$. The Taylor expansion of Y_2 starts with a term $\sim y^n$. Using $Y_{k+1}=N(Y_k)$ as suggested by the relation between Y_2 and Y_1 gives a product with n-th order convergence.

For example, for a third order product for $1/\sqrt{1-y}$, set

$$T(Y) \;:=\; \frac{1}{2}\,Y+\frac{3}{8}\,Y^2 \tag{15.106}$$

Now solve $(1+T(Y_1))^2\,(1-y)=(1-Y_2)$ for Y_2 to get

$$Y_2 \;=\; \frac{5\,y^3}{8}+\frac{15\,y^4}{64}+\frac{9\,y^5}{64} \;=:\; N(y) \tag{15.107}$$

Then, finally, ($Y_1 := y$ and)

$$\frac{1}{\sqrt{1-y}} \;=\; \prod_{k>0}(1+T(Y_k)) \quad \text{where} \quad T(Y) \;:=\; \frac{1}{2}\,Y+\frac{3}{8}\,Y^2 \tag{15.108a}$$

$$Y_{k+1} \;=\; N(Y_k) \;:=\; \frac{Y_k^3}{64}\left(40+15\,Y_k+9\,Y_k^2\right) \tag{15.108b}$$

Replace relation 15.108b by $Y_{k+1} = 1 - (1 + T(Y_k))^a \, (1 - Y_k)$ to get the general formula for the inverse a-th root.

The second order products lead to expressions that are especially nice.

For $\frac{1}{\sqrt[a]{1-y}}$ one finds:

$$\frac{1}{\sqrt[a]{1-y}} = \prod_{k>0} (1 + T(Y_k)) \quad \text{where} \quad T(y) := +\frac{y}{a} \tag{15.109a}$$

$$Y_{k+1} = N(Y_k) := 1 - \left(1 + \frac{y}{a}\right)^a (1 - y) \tag{15.109b}$$

For $\frac{1}{\sqrt[a]{1+y}}$:

$$\frac{1}{\sqrt[a]{1+y}} = \prod_{k>0} (1 + T(Y_k)) \quad \text{where} \quad T(y) := -\frac{y}{a} \tag{15.110a}$$

$$Y_{k+1} = N(Y_k) := \left(1 - \frac{y}{a}\right)^a (1 + y) - 1 \tag{15.110b}$$

For $\sqrt[a]{1-y}$ we get (using the corresponding Taylor expansion):

$$\sqrt[a]{1-y} = \prod_{k>0} (1 + T(Y_k)) \quad \text{where} \quad T(y) := -\frac{y}{a} \tag{15.111a}$$

$$Y_{k+1} = N(Y_k) := \frac{(a - Y_k)^a - (1 - Y_k)\, a^a}{(a - Y_k)^a} = 1 - \frac{(1 - Y_k)\, a^a}{(a - Y_k)^a} \tag{15.111b}$$

For $\sqrt[a]{1+y}$:

$$\sqrt[a]{1+y} = \prod_{k>0} (1 + T(Y_k)) \quad \text{where} \quad T(y) := +\frac{y}{a} \tag{15.112a}$$

$$Y_{k+1} = N(Y_k) := \frac{(1 + Y_k)\, a^a - (a + Y_k)^a}{(a + Y_k)^a} = \frac{(1 + Y_k)\, a^a}{(a + Y_k)^a} - 1 \tag{15.112b}$$

The third order product for $\frac{1}{\sqrt[a]{1-y}}$ is:

$$\frac{1}{\sqrt[a]{1-y}} = \prod_{k>0} (1 + T(Y_k)) \quad \text{where} \quad T(y) = +\frac{y}{a} + \frac{y^2\,(1+a)}{2\,a^2} \tag{15.113a}$$

$$Y_{k+1} = N(Y_k) := 1 - \left(1 + \frac{y}{a} + \frac{y^2\,(1+a)}{2\,a^2}\right)^a (1 - y) \tag{15.113b}$$

$$= 1 - (1 + T(y))^a \, (1 - y) \tag{15.113c}$$

Chapter 16

Iterations for the inversion of a function

In this chapter we study some general expressions for iterations for the zero of a function. Two schemes that give (arbitrary order, one-point) iterations, Householder's formula and Schröder's formula, are given. Several methods to construct alternative iterations are described. Moreover, iterations that also converge for multiple roots and a technique that allows to turn a linear iteration into a super-linear one are presented.

16.1 Iterations and their rate of convergence

An *iteration* for a zero r (or root, $f(r) = 0$) of a function $f(x)$ are themselves functions $\Phi(x)$ that, when 'used' like

$$x_{k+1} \quad = \quad \Phi(x_k) \tag{16.1}$$

will make x_k converge towards the root: $x_\infty = r$. Convergence is subject to the condition that x_0 was chosen close enough to r. The function $\Phi(x)$ must (and can) be constructed so that it has an attracting fixed point where $f(x)$ has a zero:

$$\Phi(r) \quad = \quad r \qquad \text{(fixed point)} \tag{16.2}$$

$$|\Phi'(r)| \quad < \quad 1 \qquad \text{(attracting)} \tag{16.3}$$

This type of iteration is called a *one-point iteration*. There are also *multi-point iterations*, these are of the form $x_{k+1} = \Phi(x_k, x_{k-1}, \ldots, x_{k-j}), j \geq 1$. The best known example is the two-point iteration

$$\Phi(x_k, x_{k-1}) \quad = \quad \frac{x_{k-1}\, f(x_k) - x_k\, f(x_{k-1})}{f(x_k) - f(x_{k-1})} \tag{16.4}$$

We are mainly concerned with one-point iterations in what follows.

The order of convergence (or simply *order*) of a given iteration can be defined as follows: let $x = r \cdot (1 + e)$ with $|e| \ll 1$ and $\Phi(x) = r \cdot (1 + \alpha e^n + O(e^{n+1}))$, then the iteration Φ is called *linear* (or first order) if $n = 1$ (and $|\alpha| < 1$). A linear iteration improves the result by (roughly) adding a constant amount of correct digits with every step.

A *super-linear* iteration does better than that: The number of correct digits grows exponentially (to the base n) at each step. Super-linear convergence of order n should better be called *exponential of order n*. Iterations of second order ($n = 2$) are often called *quadratic* (or *quadratically convergent*), those of third

order *cubic* iterations. Fourth, fifth and sixth order iterations are called *quartic*, *quintic* and *sextic* and so on.

It is conceivable to find iterations that do converge better than linear but less than exponential to any base: Imagine an iteration that produces proportional k^2 digits at step k (this is *not* quadratical convergence which produces proportional 2^k correct digits at step k). That case is not covered by the 'order-n' notion just introduced. However, those (super-linear but sub-exponential) iterations are not usually encountered and we actually won't meet one. In fact, the constructions used in this chapter cannot produce such an iteration.

For $n \geq 2$ the function Φ has a *super-attracting fixed point* at r: $\Phi'(r) = 0$. For an iteration of order n one has

$$\Phi'(r) = 0, \quad \Phi''(r) = 0, \quad \ldots, \quad \Phi^{(n-1)}(r) = 0 \tag{16.5}$$

There seems to be no standard term for emphasizing the number of derivatives vanishing at the fixed point: *attracting of order n* might be appropriate.

To any iteration of order n for a function f one can add a term $f(x)^n \cdot \varphi(x)$ (where $\varphi(x)$ is an arbitrary function that is analytic in a neighborhood of the root) without changing the order of convergence. That term is assumed to be zero in what follows. The statement can easily be checked by verifying that the first $n-1$ derivatives of $\Phi_n(x) + f(x)^n \cdot \varphi(x)$, evaluated at the root r, equal zero.

Any two one-point iterations of the same order n differ by a term $f(x)^n \cdot \varphi(x)$.

Any two iterations of the same order n differ by a term $(x-r)^n \nu(x)$ where $\nu(x)$ is a function that is finite at r (see [Hous], p. 174, ex.3).

Any one-point iteration of order n must explicitly evaluate f, f', $\ldots$, $f^{(n-1)}$ (see [Traub]).

16.2 Schröder's formula

Let $n \geq 2$ then the expression

$$S_n(x) \quad := \quad x + \sum_{t=1}^{n-1} (-1)^t \, \frac{f(x)^t}{t!} \, \left(\frac{1}{f'(x)} \, \partial \right)^{t-1} \, \frac{1}{f'(x)} \tag{16.6}$$

gives a n-th order iteration for a (simple) root r of f (see [94] p.13). This is, explicitly,

$$\begin{aligned} S \quad = \quad x \quad & - \frac{f}{1!\, f'} \quad - \frac{f^2}{2!\, f'^3} \cdot f'' \quad - \frac{f^3}{3!\, f'^5} \cdot \left(3f''^2 - f' f''' \right) \\ & - \frac{f^4}{4!\, f'^7} \cdot \left(15 f''^3 - 10 f' f'' f''' + f'^2 f'''' \right) \\ & - \frac{f^5}{5!\, f'^9} \cdot \left(105 f''^4 - 105 f' f''^2 f''' + 10 f'^2 f'''^2 + 15 f'^2 f'' f'''' - f'^3 f''''' \right) - \ldots \end{aligned} \tag{16.7}$$

The second order iteration is the well-known formula by Newton. The third order iteration obtained upon truncation after the third term on the right hand side, written as

$$S_3 \quad = \quad x - \frac{f}{f'} \left(1 + \frac{f f''}{2 f'^2} \right) \tag{16.8}$$

is sometimes referred to as 'Householder's method'. Approximating the second term on the rhs. as $\frac{f}{f'} \left(1 - \frac{f f''}{2 f'^2} \right)^{-1}$ gives Halley's formula.

Cite from [94], (p.16, translation has a typo in the first formula):

If we denote the general term by

$$-\frac{f^a}{a!}\frac{\chi_a}{f'^{2a-1}} \tag{16.9a}$$

the numbers χ_a can be easily computed by the recurrence

$$\chi_{a+1} = (2a-1)f''\chi_a - f'\partial\chi_a \tag{16.9b}$$

.

An alternative recursion is given in [Traub] (p.83): Express S_n as

$$S_n(x) = x - \sum_{t=1}^{n-1} Y_t \left(\frac{f}{f'}\right)^t \tag{16.10}$$

Then

$$Y_1 = 1 \tag{16.11a}$$

$$t\,Y_t = 2\,(t-1)\,\frac{f''}{2f'}\,Y_{t-1} - Y'_{t-1} \tag{16.11b}$$

Formula 16.6 on the preceding page with $f(x) = 1/x^a - d$ gives the 'division-free' iteration 15.39 on page 405 for arbitrary order. For $f(x) = \log(x) - d$ one obtains the iteration 17.3 on page 435.

For $f(x) = x^2 - d$ one gets

$$S(x) = x - \left(\frac{x^2-d}{2x} + \frac{\left(x^2-d\right)^2}{8x^3} + \frac{\left(x^2-d\right)^3}{16x^5} + \frac{5\left(x^2-d\right)^4}{128x^7} + \ldots\right) \tag{16.12}$$

$$= x - \left(y + \frac{1}{2x}\cdot y^2 + \frac{2}{(2x)^2}\cdot y^3 + \frac{5}{(2x)^3}\cdot y^4 + \ldots\right) \quad \text{where} \quad y := \frac{x^2-d}{2x} \tag{16.13}$$

$$= x - 2x\cdot\left(Y + Y^2 + 2\,Y^3 + 5\,Y^4 + 14\,Y^5 + 42\,Y^6 + \ldots\right) \quad \text{where} \quad Y := \frac{x^2-d}{(2x)^2} \tag{16.14}$$

The coefficients of the powers of Y are the Catalan numbers, see page 161.

A simple derivation of Schröder's formula *

Starting point is the Taylor series of a function f around x_0:

$$f(x) = \sum_{k=0}^{\infty} \frac{1}{k!} f^{(k)}(x_0)\,(x-x_0)^k \tag{16.15}$$

$$= f(x_0) + f'(x_0)\,(x-x_0) + \frac{1}{2}f''(x_0)\,(x-x_0)^2 + \frac{1}{6}f'''(x_0)\,(x-x_0)^3 + \ldots \tag{16.16}$$

Now let $f(x_0) = y_0$ and r be the zero of f (that is, $f(r) = 0$). We expand the inverse $g = f^{-1}$ around y_0

$$g(0) = \sum_{k=0}^{\infty} \frac{1}{k!} g^{(k)}(y_0)\,(0-y_0)^k \tag{16.17}$$

$$= g(y_0) + g'(y_0)\,(0-y_0) + \frac{1}{2}g''(y_0)\,(0-y_0)^2 + \frac{1}{6}g'''(y_0)\,(0-y_0)^3 + \ldots \tag{16.18}$$

Using $x_0 = g(y_0)$ and $g(0) = r$ we get

$$r = x_0 - g'(y_0)\,f(x_0) + \frac{1}{2}g''(y_0)\,f(x_0)^2 - \frac{1}{6}g'''(y_0)\,f(x_0)^3 + \ldots \tag{16.19}$$

Remains to express the derivatives of the inverse g in terms of (derivatives of) f. Not a difficult task, note that

$$f \circ g = \mathrm{id} \qquad \text{that is:} \quad f(g(x)) = x \tag{16.20}$$

and derive (chain rule) to get $g'(f(x))\,f'(x) = 1$, so $g'(y) = \frac{1}{f'(x)}$. Derive $f(g(x)) - x$ multiple times and set the expressions to zero (arguments y of g and x of f are omitted for readability):

$$\begin{aligned}
1 &= f'g' && (16.21a)\\
0 &= g'f'' + f'^2 g'' && (16.21b)\\
0 &= g'f''' + 3f'f''g'' + f'^3 g''' && (16.21c)\\
0 &= g'f'''' + 4f'g''f''' + 3f''^2 g'' + 6f'^2 f''g''' + f'^4 g'''' && (16.21d)
\end{aligned}$$

This system of linear equations in the derivatives of g can be solved successively for g', g'', g''', ...:

$$\begin{aligned}
g' &= \frac{1}{f'} && (16.22a)\\
g'' &= -\frac{f''}{f'^3} && (16.22b)\\
g''' &= \frac{1}{f'^5}\left(3f''^2 - f'f'''\right) && (16.22c)\\
g'''' &= \frac{1}{f'^7}\left(10f'f''f''' - 15f''^3 - f'^2 f''''\right) && (16.22d)\\
g''''' &= \frac{1}{f'^9}\left(105f''^4 - f'^3 f''''' - 105f'f''^2 f''' + 15f'^2 f''f'''' + 10f'^2 f'''^2\right) && (16.22e)
\end{aligned}$$

Thereby equation 16.19 can be written as

$$\begin{aligned}
r &= x - \frac{1}{f'}f + \frac{1}{2}\left(-\frac{f''}{f'^3}\right)f^2 - \frac{1}{6}\left(\frac{1}{f'^5}\left(3f''^2 - f'f'''\right)\right)f^3 + \ldots && (16.23)\\
&= x - \frac{f}{1!\,f'} - \frac{f^2}{2!\,f'^3}\cdot f'' - \frac{f^3}{3!\,f'^5}\cdot\left(3f''^2 - f'f'''\right) - \ldots
\end{aligned}$$

which is Schröder's iteration (equation 16.7 on page 420).

16.3 Householder's formula

For $n \geq 2$ the expression

$$H_n(x) \;:=\; x + (n-1)\,\frac{\left(\frac{g(x)}{f(x)}\right)^{(n-2)}}{\left(\frac{g(x)}{f(x)}\right)^{(n-1)}} \tag{16.24}$$

gives a n-th order iteration for a (simple) root r of f. The function $g(x)$ must be analytic near the root and is set to 1 in what follows (see [Hous] p.169). We have

$$\begin{aligned}
H_2(x) &= x - \frac{f}{f'} && (16.25a)\\
H_3(x) &= x - \frac{2ff'}{2f'^2 - ff''} && (16.25b)\\
H_4(x) &= x - \frac{3f(ff'' - 2f'^2)}{6ff'f'' - 6f'^3 - f^2 f'''} && (16.25c)\\
H_5(x) &= x + \frac{4f\left(6f'^3 - 6ff'f'' + f^2 f'''\right)}{f^3 f'''' - 24f'^4 + 36ff'^2 f'' - 8f^2 f'f''' - 6f^2 f''^2} && (16.25d)
\end{aligned}$$

The second order variant is Newton's formula, the third order iteration is called Halley's formula.

In [66] Kalantari and Gerlach define the iteration

$$B_m(x) = x - f(x)\,\frac{D_{m-2}(x)}{D_{m-1}(x)} \tag{16.26a}$$

where $m \geq 2$, $D_0 = 1$, $D_1 = f'$ and

$$D_m(x) \quad = \quad \det \begin{pmatrix} f'(x) & \frac{f''(x)}{2!} & \dots & \frac{f^{(m-1)}(x)}{(m-1)!} & \frac{f^{(m)}(x)}{m!} \\ f(x) & f'(x) & \ddots & \ddots & \frac{f^{(m-1)}(x)}{(m-1)!} \\ 0 & f(x) & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \frac{f''(x)}{2!} \\ 0 & 0 & \ddots & f(x) & f'(x) \end{pmatrix} \tag{16.26b}$$

The iteration turns out to be identical to the one of Householder ($B_n = H_n$). A recursive definition for $D_m(x)$ is given by

$$D_m(x) = \sum_{i=1}^{m} (-1)^{i-1} f(x)^{i-1} \frac{f^{(i)}(x)}{i!} D_{m-i}(x) \tag{16.27}$$

The well-known derivation of Halley's formula by applying Newton's formula to $f/\sqrt{f'}$ can be generalized to produce m-order iterations as follows: Let $F_1(x) = f(x)$ and for $m \geq 2$ let

$$F_m(x) \quad = \quad \frac{F_{m-1}(x)}{F'_{m-1}(x)^{1/m}} \tag{16.28a}$$

$$H_m(x) \quad = \quad x - \frac{F_{m-1}(x)}{F'_{m-1}(x)} \tag{16.28b}$$

That this recursion indeed gives the Householder iteration is shown in [66].

An alternative recursive formulation (also given in [66], ascribed to Ford/Pennline) is

$$Q_2(x) \quad = \quad 1 \tag{16.29a}$$

$$Q_{m+1} \quad = \quad f'(x)\,Q_m(x) - \frac{1}{m-1}\,f(x)\,Q'_m(x) \tag{16.29b}$$

$$H_m \quad = \quad x - f(x)\,\frac{Q_m(x)}{Q_{m-1}(x)} \tag{16.29c}$$

The Taylor series of the k-th order Householder iteration around $f = 0$ up to order $k-1$ gives the k-th order Schröder iteration.

16.4 Dealing with multiple roots

The iterations given so far will not converge at the stated order if f has a multiple root at r. As an example consider the function

$$f(x) \quad = \quad (x^2 - d)^m \quad \text{where} \quad m \in \mathbb{N} \tag{16.30}$$

The iteration $\Phi(x) = x - f/f'$ is

$$\Phi_?(x) \quad = \quad x - \frac{x^2 - d}{m2x} \tag{16.31}$$

Its convergence is only linear for $m > 1$: $\Phi(\sqrt{d}(1+e)) = \sqrt{d}(1 + \frac{m-1}{m}e + O(e^2))$

Householder ([Hous] p.161 ex.6) gives

$$\Phi_2(x) = x - m \cdot \frac{f}{f'} \tag{16.32}$$

as a second order iteration for a root of known multiplicity m. Note that with the example above we get a quadratic iteration.

For roots of unknown multiplicity use the general expressions for iterations with $F := f/f'$ instead of f. Both F and f have the same set of roots, but all roots of F are simple. To illustrate this consider a function f that has a root of multiplicity m at r: $f(x) := (x-r)^m\, h(x)$ with $h(r) \neq 0$. Then

$$\begin{aligned} f'(x) &= m\,(x-r)^{m-1}\, h(x) + (x-r)^m\, h'(x) && (16.33a)\\ &= (x-r)^{m-1}\,\Big(m\,h(x) + (x-r)\,h'(x)\Big) && (16.33b)\end{aligned}$$

and

$$F(x) = f(x)/f'(x) \quad = \quad (x-r)\,\frac{h(x)}{m\,h(x) + (x-r)\,h'(x)} \tag{16.34}$$

The fraction on the right hand side does not vanish at the root r.

Plugging $F = f/f'$ into Householder's formula (relation 16.24) we get the following iterations denoted by $H_k^{\%}$, the iterations H_k are given for comparison:

$$\begin{aligned} H_2(x) &= x - \frac{f}{f'} && (16.35a)\\ H_2^{\%}(x) &= x - \frac{ff'}{f'^2 - ff''} && (16.35b)\\ H_3(x) &= x - \frac{2ff'}{2f'^2 - ff''} && (16.35c)\\ H_3^{\%}(x) &= x + \frac{2f^2f'' - 2ff'^2}{2f'^3 - 3ff'f'' + f^2f'''} && (16.35d)\\ H_4(x) &= x + \frac{3f^2f'' - 6ff'^2}{6f'^3 - 6ff'f'' + f^2f'''} && (16.35e)\\ H_4^{\%}(x) &= x + \frac{6ff'^3 + 3f^3f''' - 9f^2f'f''}{f^3f'''' - 6f'^4 + 12ff'^2f'' - 4f^2f'f''' - 3f^2f''^2} && (16.35f)\\ H_5(x) &= x + \frac{24ff'^3 + 4f^3f''' - 24f^2f'f''}{f^3f'''' - 24f'^4 + 36ff'^2f'' - 8f^2f'f''' - 6f^2f''^2} && (16.35g)\end{aligned}$$

The terms in the numerators and denominators of $H_k^{\%}$ and H_{k+1} are identical up to the integral constants. A alternative form for $H_k^{\%}$ is given by

$$H_k^{\%} \quad = \quad x + (k-1)\,\frac{(\log(f))^{(k-1)}}{(\log(f))^{(k)}} \tag{16.36}$$

Schröder's formula (relation 16.6), when inserting f/f', becomes:

$$\begin{aligned} S^{\%}(x) \quad = \quad & x - \frac{ff'}{(f'^2 - ff'')} - \frac{f^2f'\left(ff'f''' - 2ff''^2 + f'^2f''\right)}{2\left(ff'' - f'^2\right)^3} - \\ & - \frac{f^3f'\left(2ff'^3f''f''' \pm \ldots - 3f^2f'^2f''''^2\right)}{6\left(ff'' - f'^2\right)^5} - \frac{f^4f'\left(3f'^8f'''' \pm \ldots - 36f^3f'^2f''^2f''''^2\right)}{24\left(ff'' - f'^2\right)^7} - \\ & - \ldots - \frac{f^kf'(\ldots)}{k!\left(ff'' - f'^2\right)^{2k-1}} \end{aligned} \tag{16.37}$$

Checking convergence with the example function (relation 16.30) we began with: The second order iteration is

$$\Phi_2^{\%}(x) \;=\; S_2 \;=\; H_2 \;=\; x + x\frac{d-x^2}{d+x^2} = \frac{2\,d\,x}{x^2+d} \tag{16.38}$$

Convergence is indeed second order, as

$$\Phi_2^{\%}(\sqrt{d}\cdot\frac{1-e}{1+e}) \;=\; \sqrt{d}\cdot\frac{1-e^2}{1+e^2} \tag{16.39}$$

which holds independent of m. The expression is similar to relation 15.21.

In general, with $H_k^{\%}$ one obtains for the square root:

$$\Phi_k^{\%}(\sqrt{d}\cdot\frac{1-e}{1+e}) \;=\; \sqrt{d}\cdot\frac{1-e^k}{1+e^k} \tag{16.40}$$

Using Schröder's third order formula for f/f' with f as in 16.30 we get a nice *fourth* order iteration for $\sqrt{d}$:

$$S_3^{\%}(x) \;=\; x + x\,\frac{d-x^2}{d+x^2} + x\,d\,\frac{(d-x^2)^2}{(d+x^2)^3} \tag{16.41a}$$

$$S_3^{\%}(\sqrt{d}\,\frac{1-e}{1+e}) \;=\; \sqrt{d}\,\frac{1+3e^2-3e^4-e^6}{1+3e^2+3e^4+e^6} \tag{16.41b}$$

$$\;=\; \sqrt{d}\,\frac{1-c}{1+c} \quad\text{where}\quad c = e^4\,\frac{e^2+3}{3e^2+1} \tag{16.41c}$$

In general, the $(1+a\,k)$-th order Schröder iteration for $1/\sqrt[a]{d}$ obtained through f/f' has an order of convergence that exceeds the expected one by one.

For $f(x) = 1 - d\,x^2$ we obtain (third order Schröder):

$$S_3^{\%}(x) \;=\; x + x\,\frac{1-dx^2}{1+dx^2} + x\,\frac{(1-dx^2)^2}{(1+dx^2)^3} \tag{16.42}$$

The iteration also has 4th order convergence and the error expression $S_3^{\%}(\frac{1}{\sqrt{d}}\,\frac{1-e}{1+e})$ is the same (replacing $\sqrt{d}$ by $1/\sqrt{d}$) as in relation 16.41b.

16.5 More iterations

Rational iterations from Padé approximants

The $[i,j]$-th Padé approximant of Φ_n in f gives an iteration of order $p = i+j+1$ (if $n \geq p$). Write $P_{[i,j]}$ for an iteration (of order $i+j+1$) that is obtained using the Padé approximant $[i,j]$. For the second order (where the well-known Newton formula corresponds to $P_{[1,0]}(x) = x - \frac{f}{f'}$) this method gives one alternative form, namely

$$P_{[0,1]}(x) \;=\; x^2\,\frac{f'}{f + x\,f'} \;=\; x - \frac{x\,f}{f + x\,f'} \;=\; x - \frac{x\,f}{(x\,f)'} \;=\; x\left(1 + \frac{f}{x\,f'}\right)^{-1} \tag{16.43}$$

For the third order one finds $P_{[2,0]}(x) = S_3(x)$, $P_{[1,1]}(x) = H_3(x)$ and

$$\begin{aligned} P_{[0,2]}(x) &= \frac{2x^3 f'^3}{2f^2 f' + 2xff'^2 + xf^2 f'' + 2x^2 f'^3} && (16.44a)\\ &= x - \frac{xf\left(2ff' + xff'' + 2xf'^2\right)}{\left(2f^2 f' + 2xff'^2 + xf^2 f'' + 2x^2 f'^3\right)} && (16.44b)\\ &= x\left(1 + \frac{f}{xf'} + \frac{f^2}{x^2 f'^2} + \frac{f^2 f''}{2xf'^3}\right)^{-1} && (16.44c)\\ &= x\left(1 + \frac{f}{(x\,f')} + \frac{f^2(x^2\,f'')}{2(x\,f')^3} + \frac{f^2}{(x\,f')^2}\right)^{-1} && (16.44d) \end{aligned}$$

Alternatively one can use the Padé approximant $A_{[i,j]}$ of $(\Phi(x)-x)/f$ in f where $\Phi(x)$ is a given iteration of order $\geq i+j+2$. Then $\Phi^+_{[i,j]} := x + f \cdot A_{[i,j]}$ gives an iteration which has order $n = i+j+2$.

$$\begin{aligned} \Phi^+_{[0,1]}(x) &= x - \frac{2ff'}{2f'^2 - ff''} \;=\; H_3(x) && (16.45a)\\ \Phi^+_{[1,0]}(x) &= x - \frac{f\left(f\,f'' + 2\,f'^2\right)}{2\,f'^3} \;=\; S_3(x) && (16.45b) \end{aligned}$$

The iterations Φ^+ of order n are expressions in $x, f, f', \ldots, f^{(n-1)}$. Fourth order iterations are

$$\begin{aligned} \Phi^+_{[2,0]}(x) &= x - \frac{f\left(6\,f'^4 + 3\,f\,f'^2\,f'' - f^2\,f'\,f''' + 3\,f^2\,f''^2\right)}{6\,f'^5} && (16.46a)\\ &= x - \frac{f}{f'} - \frac{f^2}{2\,f'^3}\cdot f'' - \frac{f^3}{6\,f'^5}\cdot\left(3f''^2 - f'f'''\right) && (16.46b)\\ &= x - \frac{f^2}{f'^2}\left(\frac{f'}{f} + \frac{f''}{2\,f'} + \frac{f\left(3\,f''^2 - f'\,f'''\right)}{6\,f'^3}\right) && (16.46c)\\ \Phi^+_{[1,1]}(x) &= x - \frac{f\left(2\,f\,f'\,f''' - 3\,f\,f''^2 + 6\,f'^2\,f''\right)}{f'\left(2\,f\,f'\,f''' - 6\,f\,f''^2 + 6\,f'^2\,f''\right)} && (16.46d)\\ \Phi^+_{[0,2]}(x) &= x - \frac{12\,f\,f'^3}{\left(12\,f'^4 - 6\,f\,f'^2\,f'' + 2\,f^2\,f'\,f''' - 3\,f^2\,f''^2\right)} && (16.46e)\\ &= x - \left(\frac{f'}{f} - \frac{f''}{2\,f'} - \frac{f\left(3\,f''^2 - 2\,f'\,f'''\right)}{12\,f'^3}\right)^{-1} && (16.46f) \end{aligned}$$

$\Phi^+_{[n,0]}$ always coincides with Schröder's iteration. In general one obtains $n-1$ additional forms of iterations using the approximants $[0, n-2]$, $[1, n-3]$, $\ldots$, $[n-3, 1]$.

Neglecting in 16.46d terms that contain the third derivative one obtains the third order iteration

$$\Phi_3 \;=\; x - \frac{f\left(2\,f'^2 - f\,f''\right)}{f'\left(2\,f'^2 - 2\,f\,f''\right)} \qquad (16.47)$$

An iteration involving radicals

The third order iteration

$$\Phi_3(x) \quad = \quad x - \frac{1}{f''}\left(f' \pm \sqrt{f'^2 - 2ff''}\right) = x - \frac{f'}{f''}\left(1 \pm \sqrt{1 - 2\,\frac{f\,f''}{f'^2}}\right) \tag{16.48}$$

stems from directly solving the truncated Taylor expansion of $f(r) = 0$ around x.

$$f(r) \quad = \quad f(x) + f'(x)\,(r-x) + \frac{1}{2}f''(x)\,(r-x)^2 \tag{16.49}$$

For $f(x) = ax^2 + bx + c$ it gives the two solutions of the quadratic equation $f(x) = 0$; for other functions one gets an iterated square root expression for the roots.

The following form avoids possible cancellation

$$\Phi_3(x) \quad = \quad x - \frac{2\,u}{1 + \sqrt{1 - 4\,A\,u}} \quad \text{where} \quad u = f/f' \quad \text{and} \quad A = f''/(2f) \tag{16.50}$$

The relation is given in [Traub] (p.94). It can be obtained by observing that

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad = \quad \frac{-2c}{b + \sqrt{b^2 - 4ac}} \tag{16.51}$$

Iterations from iterations

Alternative rational forms can also be obtained in a way that generalizes the the method used for multiple roots: if we emphasize the so far notationally omitted dependency from the function f as $\Phi\{f\}$. The iteration $\Phi\{f\}$ has fixed points where f has a root r, so $x - \Phi\{f\}$ again has a root at r. Hence we can build more iterations that will converge to those roots as $\Phi\{x - \Phi\{f\}\}$. For dealing with multiple roots we used $\Phi\{x - \Phi_2\{f\}\}_k = \Phi\{f/f'\}$. An iteration $\Phi_k\{x - \Phi_j\{f\}\}$ can only be expected to have a k-th order convergence.

16.6 Improvements by the delta squared process

Given a sequence of partial sums x_k the so-called *delta squared process* computes a new sequence x_k^* of extrapolated sums:

$$x_k^* \quad = \quad x_{k+2} - \frac{(x_{k+2} - x_{k+1})^2}{x_{k+2} - 2\,x_{k+1} + x_k} \tag{16.52}$$

The method is due to Aitken. The name delta squared is due to the fact that the formula can be written symbolically as

$$x^* = x - \frac{(\Delta x)^2}{(\Delta^2 x)} \tag{16.53}$$

where Δ is the difference operator.

Note that the mathematically equivalent form

$$x_k^* \quad = \quad \frac{x_k\,x_{k+2} - x_{k+1}^2}{x_{k+2} - 2\,x_{k+1} + x_k} \tag{16.54}$$

sometimes given should be avoided with numerical computations due to possible cancellation.

If $x_k = \sum_{i=0}^{k} a_i$ and the ratio of consecutive summands a_i is approximately constant (that is, a is close to a geometric series) then x^* converges significantly faster to x_∞ than x. Let us partly rewrite the formula using $x_k - x_{k-1} = a_k$:

$$x_k^* = x_{k+2} - \frac{(a_{k+2})^2}{a_{k+2} - a_{k+1}} \tag{16.55}$$

Then for a geometric series with $a_{k+1}/a_k = q$

$$x_k^* = x_{k+2} - \frac{(a_{k+2})^2}{a_{k+2} - a_{k+1}} = x_{k+2} - \frac{\left(a_0\, q^{k+2}\right)^2}{a_0\,(q^{k+2} - q^{k+1})} \tag{16.56a}$$

$$= a_0\,\frac{1-q^{k+3}}{1-q} + a_0\,q^{k+2}\cdot\frac{q^{k+2}}{q^{k+1}-q^{k+2}} = \frac{a_0}{1-q}\,\left(1 - q^{k+3} + q^{k+3}\right) \tag{16.56b}$$

$$= \frac{a_0}{1-q} \tag{16.56c}$$

which is the exact sum.

Why do we meet the delta squared here? Consider the sequence

$$x_0, \quad x_1 = \Phi(x_0), \quad x_2 = \Phi(x_1) = \Phi(\Phi(x_0)), \quad \ldots \tag{16.57}$$

of better and better approximations to some root r of a function f. Think of the x_k as partial sums of a series whose sum is the root r. Apply the idea to define an improved iteration Φ^* from a given one Φ:

$$\Phi^*(x) = \Phi(\Phi(x)) - \frac{[\Phi(\Phi(x)) - \Phi(x)]^2}{\Phi(\Phi(x)) - 2\,\Phi(x) + x} \tag{16.58}$$

The good news is that $\Phi*$ will give quadratic convergence even if Φ only gives linear convergence. As an example let us take $f(x) = (x^2 - d)^2$, forget that its root $\sqrt{d}$ is a double root and happily define $\Phi(x) = x - f(x)/f'(x) = x - (x^2 - d)/(4x)$. Convergence is only linear:

$$\Phi(\sqrt{d}\cdot(1+e)) = \sqrt{d}\cdot\left(1 + \frac{e}{2} + \frac{e^2}{4} + O(e^3)\right) \tag{16.59}$$

Then try

$$\Phi^*(x) = \frac{d\,(7\,x^2 + d)}{x\,(3\,x^2 + 5\,d)} \tag{16.60}$$

and find that it offers quadratic convergence

$$\Phi(\sqrt{d}\cdot(1+e)) = \sqrt{d}\cdot\left(1 - \frac{e^2}{4} + \frac{e^3}{16} + O(e^4)\right) \tag{16.61}$$

In general, if Φ_n has convergence of order $n > 1$ then Φ_n^* will be of order $2n - 1$, linear convergence ($n = 1$) is turned into second order (see [Hous]).

Chapter 17

Transcendental functions and the AGM

We describe fast algorithms for the computation of some transcendental functions and quantities to high precision.

17.1 The AGM

The *arithmetic geometric mean* (AGM) plays a central role in the high precision computation of logarithms and π.

The AGM(a, b) is defined as the limit of the iteration

$$a_{k+1} = \frac{a_k + b_k}{2} \tag{17.1a}$$

$$b_{k+1} = \sqrt{a_k\, b_k} \tag{17.1b}$$

starting with $a_0 = a$ and $b_0 = b$. Both of the values converge quadratically to a common limit. The related quantity c_k used in many AGM based computations is defined as

$$c_k^2 = a_k^2 - b_k^2 = (a_{k-1} - a_k)^2 \tag{17.2}$$

An alternative way for the computation for the AGM iteration is

$$c_{k+1} = \frac{a_k - b_k}{2} \tag{17.3a}$$

$$a_{k+1} = \frac{a_k + b_k}{2} \tag{17.3b}$$

$$b_{k+1} = \sqrt{a_{k+1}^2 - c_{k+1}^2} \tag{17.3c}$$

Schönhage gives the most economic variant of the AGM, which, apart from the square root, only needs

one squaring per step:

$$\begin{aligned}
A_0 &= a_0^2 && (17.4a)\\
B_0 &= b_0^2 && (17.4b)\\
t_0 &= 1-(A_0-B_0) && (17.4c)\\
S_k &= \frac{A_k+B_k}{4} && (17.4d)\\
b_k &= \sqrt{B_k} \qquad \text{[square root]} && (17.4e)\\
a_{k+1} &= \frac{a_k+b_k}{2} && (17.4f)\\
A_{k+1} &= a_{k+1}^2 \qquad \text{[squaring]} && (17.4g)\\
&= \left(\frac{\sqrt{A_k}+\sqrt{B_k}}{2}\right)^2 = \frac{A_k+B_k}{4}+\frac{\sqrt{A_k\,B_k}}{2} && (17.4h)\\
B_{k+1} &= 2\,(A_{k+1}-S_k) \;=\; b_{k+1}^2 && (17.4i)\\
c_{k+1}^2 &= A_{k+1}-B_{k+1} \;=\; a_{k+1}^2-b_{k+1}^2 && (17.4j)\\
t_{k+1} &= t_k-2^{k+1}\,c_{k+1}^2 && (17.4k)
\end{aligned}$$

Starting with $a_0=A_0=1$, $B_0=1/2$ one has $\pi\approx(2\,a_n^2)/t_n$.

The importance of the AGM is related to the fact that it allows to compute certain hypergeometric functions fast. Indeed, one has

$$F\left(\left.\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\right|1-\frac{b^2}{a^2}\right) \;=\; \frac{a}{\mathrm{AGM}(a,b)} \;=\; \frac{1}{\mathrm{AGM}(1,b/a)} \tag{17.5}$$

The relation is usually written as

$$F\left(\left.\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\right|k\right) \;=\; \frac{1}{\mathrm{AGM}(1,\sqrt{1-k})} \tag{17.6}$$

corresponding to $\mathrm{AGM}(1,k)$, that is, $a_0=1$, $b_0=k$, $c_0=\sqrt{1-k^2}$.

The quantity

$$R'(k) \;:=\; 1-\frac{1}{2}\sum_{n=0}^{\infty}2^n c_n^2 \tag{17.7}$$

together with the AGM leads to a fast algorithm for the function $F\left(\left.\begin{matrix}-\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\right|k\right)$, see section 17.5 on page 439.

Fourth order AGM

Combining two steps of the AGM iteration leads to the fourth order AGM iteration:

$$\begin{aligned}
\alpha_0 &= \sqrt{a_0} && (17.8a)\\
\beta_0 &= \sqrt{b_0} && (17.8b)\\
\alpha_{k+1} &= \frac{\alpha_k+\beta_k}{2} && (17.8c)\\
\beta_{k+1} &= \left(\frac{\alpha_k\,\beta_k\,(\alpha_k^2+\beta_k^2)}{2}\right)^{1/4} && (17.8d)\\
\gamma_k^4 &= \alpha_k^4-\beta_k^4 \qquad = c_{2k}^2 && (17.8e)
\end{aligned}$$

An alternative formulation of the 4th order AGM iteration is:

$$\gamma_{k+1} = \frac{\alpha_k - \beta_k}{2} \tag{17.9a}$$

$$\alpha_{k+1} = \frac{\alpha_k + \beta_k}{2} \tag{17.9b}$$

$$\beta_{k+1} = \left(\alpha_{k+1}^4 - \gamma_{k+1}^4\right)^{1/4} \tag{17.9c}$$

$$c_{2k}^2 + 2\,c_{2k+1}^2 = \alpha_{k-1}^4 - \left(\alpha_k^2 - \gamma_k^2\right)^2 \tag{17.9d}$$

We have $\alpha_k = \sqrt{a_{2k}}$, $\beta_k = \sqrt{b_{2k}}$ and

$$R'(k) = 1 - \sum_{n=0}^{\infty} 4^n \left(\alpha_n^4 - \left(\frac{\alpha_n^2 + \beta_n^2}{2}\right)^2\right) \tag{17.10}$$

17.2 Computation of the logarithm

The (natural) logarithm can be computed using the relation (see [PiAGM], p.221)

$$\left|\log(d) - R'(10^{-n}) + R'(10^{-n}\,d)\right| \leq \frac{n}{10^{2(n-1)}} \tag{17.11}$$

$$\log(d) \approx R'(10^{-n}) - R'(10^{-n}\,d) \tag{17.12}$$

which holds for $n \geq 3$ and $d \in]\frac{1}{2}, 1[$. Note that the first term on the rhs. is constant and can be saved for subsequent log-computations. Section 17.6 gives an algorithm for the computation of the logarithm that needs the precomputed value of π.

A C++ implementation is given in [hfloat: src/tz/log.cc]: Firstly, set $d = M \cdot R^X$ (for mantissa, radix and exponent), $\log(d)$ is computed as $\log(M) + X \cdot \log(R)$ where $\log(R)$ is computed only once (by an algorithm given in section 17.6). Secondly, if M is not in the interval [1/2, 3/2] a reduction is done via

$$\log(M) = \log\left(M\,s^f\right) - f\,\log(s)$$

Where $0 < M < 1$ is the mantissa, $s = \sqrt{2}$ and $f \in \mathbb{Z}$ so that $M\,s^f \in [1/2, 3/2]$. The quantity $\log(s) = \log(\sqrt{2})$ can be precomputed directly via the AGM.

A discussion of the AGM-based computation of logarithms can be found in [18].

With an efficient algorithm for exp() one can compute log() from exp() using

$$y := 1 - d\,e^{-x} \tag{17.13}$$

$$\log(d) = x + \log(1 - y) \tag{17.14}$$

$$= x + \log\left(1 - (1 - d\,e^{-x})\right) = x + \log\left(e^{-x}d\right) = x + (-x + \log(d)) \tag{17.15}$$

Then

$$\log(d) = x + \log(1 - y) = x - \left(y + \frac{y^2}{2} + \frac{y^3}{3} + \frac{y^4}{4} + \ldots\right) \tag{17.16}$$

Truncation of the series before the n-th power of y gives an iteration of order n:

$$x_{k+1} = \Phi_n(x_k) \quad \text{where} \tag{17.17a}$$

$$\Phi_n(x) := x - \left(y + \frac{y^2}{2} + \frac{y^3}{3} + \cdots + \frac{y^{n-1}}{n-1}\right) \tag{17.17b}$$

Padé approximants $P_{[i,j]}(z)$ of $\log(1-z)$ at $z=0$ produce iterations of order $i+j+1$. Compared to the power series based iteration one needs one additional long division but saves half of the exponentiations. This can be a substantial saving for high order iterations.

The approximants can be computed via the continued fraction expansion of $\log(1+z)$:

$$\log(1+z) \;=\; 0+\cfrac{c_1\,z}{1+\cfrac{c_2\,z}{1+\cfrac{c_3\,z}{1+\cfrac{c_4\,z}{1+\cdots}}}} \tag{17.18}$$

where $c_1 = 1$ and

$$c_k \;=\; \frac{k}{4(k-1)} \quad \text{if } k \text{ even}, \qquad c_k \;=\; \frac{k-1}{4k} \quad \text{else} \tag{17.19}$$

Using recursions 20.27a and 20.27b with $a_0 = 0$, $a_k = 1$ and $b_k = c_k \cdot z$ we get (fractions in lowest terms):

$$1 \;\mapsto\; P_{[1,0]} \;=\; \frac{z}{1} \tag{17.20a}$$

$$2 \;\mapsto\; P_{[1,1]} \;=\; \frac{2\,z}{2+\,z} \tag{17.20b}$$

$$3 \;\mapsto\; P_{[2,1]} \;=\; \frac{6\,z+\,z^2}{6+4\,z} \tag{17.20c}$$

$$4 \;\mapsto\; P_{[2,2]} \;=\; \frac{6\,z+3\,z^2}{6+6\,z+\,z^2} \tag{17.20d}$$

$$5 \;\mapsto\; P_{[3,2]} \;=\; \frac{30\,z+21\,z^2+\,z^3}{30+36\,z+9\,z^2} \tag{17.20e}$$

$$6 \;\mapsto\; P_{[3,3]} \;=\; \frac{60\,z+60\,z^2+11\,z^3}{60+90\,z+36\,z^2+3\,z^3} \tag{17.20f}$$

$$7 \;\mapsto\; P_{[4,3]} \;=\; \frac{420\,z+510\,z^2+140\,z^3+3\,z^4}{420+720\,z+360\,z^2+48\,z^3} \tag{17.20g}$$

$$8 \;\mapsto\; P_{[4,4]} \;=\; \frac{420\,z+630\,z^2+260\,z^3+25\,z^4}{420+840\,z+540\,z^2+120\,z^3+6\,z^4} \tag{17.20h}$$

The expressions are Padé approximants correct up to order k. For even k these are the diagonal approximants $[k/2, k/2]$ which satisfy the functional equation $\log(1/z) = -\log(z)$: $P(1/z-1) = -P(z-1)$. Further information like the error term of the diagonal approximants is given in [76].

Relations 20.32a and 20.32b lead to an algorithm that produces just the diagonal approximants. Set $P_0 = 0$, $Q_0 = 1$, $P_2 = z$, $Q_2 = 1 + z/2$ and for $k = 4, 6, \ldots, 2n$ compute

$$P_k \;=\; A_k\,P_{k-2} + B_k\,P_{k-4} \tag{17.21a}$$

$$Q_k \;=\; A_k\,Q_{k-2} + B_k\,Q_{k-4} \tag{17.21b}$$

where

$$A_k \;=\; 1 + z/2 \tag{17.21c}$$

$$B_k \;=\; \frac{z^2}{16}\,\frac{(k-2)^2}{1-(k-2)^2} \tag{17.21d}$$

Then P_{2n}/Q_{2n} is the Padé approximant $[n, n]$ of $\log(1+z)$. The following pari/gp function implements the algorithm:

```
log_pade(n, z='z)=
{ /* Return Pade approximant [n,n] of log(1+z) */
    local(P0,Q0,P2,Q2,tp,tq, t);
    if ( n<1, return(0) );
    P0=0;  Q0=1;
    P2=z;  Q2=1+z/2;
    forstep (k=4, 2*n, 2,
        Ak = 1+z/2;  \\ == +z*C(k-1)+z*C(k)+1;
        t = (k-2)^2;
        Bk = z^2/16*t/(1-t);  \\ == -z^2*C(k-1)*C(k-2);
        tp = Ak*P2 + Bk*P0;
        tq = Ak*Q2 + Bk*Q0;
        P0=P2; P2=tp;
        Q0=Q2; Q2=tq;
    );
    return( P2/Q2 );
}
```

Argument reduction for the logarithm

When the logarithm shall be computed for high but not extreme precision (up to several hundred decimal digits or so), the following scheme can beat the AGM algorithm. Use the functional equation for the logarithm

$$\log(z^a) \quad = \quad a\,\log(z) \tag{17.22}$$

to reduce the argument by setting $a = 1/N$. Now with N big enough the $z^{1/N}$ will be close to one: $r := z^{1/N} = 1+e$ where e is small. Then a few terms of the Taylor series of $\log(1+e) = e - 1/2\,e^2 + 1/3\,e^3 \pm \ldots$ suffice to compute the logarithm: Compute the logarithm of z as follows

1. set $r = z^{1/N}$ and $e = r - 1$
2. compute $l := \log(1 + e)$ to the desired precision using the Taylor series
3. return $L := N\,l$

One can of course also use a Padé approximant in step 2. For example, with argument $z = 2.0$, $N = 2^{32}$, and four terms of the Taylor series we obtain:

```
? z=2.0;   \\ argument for log()
? n=32; N=2^n;
? r=z^(1/N)  \\ compute by 32 sqrt extractions
  1.000000000161385904209659761203976631101985032744612016
? e=r-1; \\ small
  1.613859042096597612039766311019850327446120165053265785 E-10
? l=e-1/2*e^2+1/3*e^3-1/4*e^4  \\ approx log(1+e)
  1.613859041966370561665930136708022486594054133693140550 E-10
? L=N*l     \\ final result
  0.693147180559945309417232121458176568075406932265650365
? log(z)  \\ check with builtin log
  0.693147180559945309417232121458176568075500134360255254l
```

Similarly for the inverse tangent function. We use the equation

$$\arctan(z) \quad = \quad 2\,\arctan\left(\frac{z}{1+\sqrt{1+z^2}}\right) \tag{17.23}$$

Compute the inverse tangent of z as follows

1. set $r := z$
2. repeat n times: $r = r/(1+\sqrt{1+r^2})$ (for n big enough)
3. compute $a := \arctan(r)$ to the desired precision using the Taylor series
4. return $A := 2^n\,a$

We compute arctan(1.0) using $n = 16$ and four terms of the Taylor series:

```
? z=1.0;
? n=16;
? r=z;for(k=1,n,r=r/(1+sqrt(1+r^2)));r
  0.000011984224905930303478511634794650661319585128744025261 89
? a=r-1/3*r^3+1/5*r^5-1/7*r^7
  0.000011984224905356572107172559292905818497455676569676602 63
? A=2^n*a
  0.785398163397448309615660845819875721049255219670325829 9
? atan(z)  \\ check with builtin atan
  0.785398163397448309615660845819875721049292349843776455 2
```

All divisions in the reduction phase can be saved by using

$$\arctan(1/z) \quad = \quad 2\,\arctan\left(\frac{1}{1+\sqrt{1+z^2}}\right) \tag{17.24}$$

The inverse sine and cosine can be computed as

$$\arcsin(z) \quad = \quad \arctan\left(\frac{z}{1-z^2}\right) \tag{17.25a}$$

$$\arccos(z) \quad = \quad \arctan\left(\frac{1-z^2}{z}\right) \tag{17.25b}$$

A curious series for the logarithm *

We finally note two relations resembling the well-known series

$$\frac{1}{2}\,\log\left(\frac{1+x}{1-x}\right) \quad = \quad x+\frac{1}{3}\,x^3+\frac{1}{5}\,x^5+\ldots+\frac{1}{2k+1}\,x^{2k+1}+\ldots \tag{17.26}$$

The first is

$$\frac{1}{6}\,\log\left(\frac{1+3x+3x^2}{1-3x+3x^2}\right) \quad = \quad x-\frac{3^2}{5}\,x^5-\frac{3^3}{7}\,x^7+\frac{3^5}{11}\,x^{11}+\frac{3^6}{13}\,x^{13}\pm\ldots= \tag{17.27a}$$

$$= \quad \sum_{k=0}^{\infty}\left(+\frac{3^{6k+0}}{12k+1}\,x^{12k+1}-\frac{3^{6k+2}}{12k+5}\,x^{12k+5}-\frac{3^{6k+3}}{12k+7}\,x^{12k+7}+\frac{3^{6k+5}}{12k+11}\,x^{12k+11}\right) \tag{17.27b}$$

The second, given in [13], is

$$\frac{1}{3}\,\log\left(\frac{1+x+x^2}{1-2x+x^2}\right) \quad = \quad x+\frac{x^2}{2}+\frac{x^4}{4}+\frac{x^5}{5}\pm\ldots \;=\; \sum_{k=0}^{\infty}\left(+\frac{x^{3k+1}}{3k+1}+\frac{x^{3k+2}}{3k+2}\right) \tag{17.28}$$

Relation 17.27a can be brought in a similar form, namely

$$\frac{1}{2\sqrt{3}}\,\log\left(\frac{1+\sqrt{3}\,x+x^2}{1-\sqrt{3}\,x+x^2}\right) \quad = \quad x-\frac{x^5}{5}-\frac{x^7}{7}+\frac{x^{11}}{11}+\frac{x^{13}}{13}\pm\ldots= \tag{17.29a}$$

$$= \quad \sum_{k=0}^{\infty}\left(+\frac{x^{12k+1}}{12k+1}-\frac{x^{12k+5}}{12k+5}-\frac{x^{12k+7}}{12k+7}+\frac{x^{12k+11}}{12k+11}\right) \tag{17.29b}$$

17.3 Computation of the exponential function

The exponential function can be computed using the iteration that is obtained as follows:

$$\exp(d) \quad = \quad x\,\exp\left(d-\log(x)\right) \tag{17.30}$$

$$= \quad x\,\exp(y) \quad \text{where} \quad y := d-\log(x) \tag{17.31}$$

$$= \quad x\left(1+y+\frac{y^2}{2}+\frac{y^3}{3!}+\frac{y^4}{4!}+\ldots\right) \tag{17.32}$$

The corresponding n-th oder iteration is

$$x_{k+1} = \Phi_n(x_k) \quad \text{where} \tag{17.33a}$$

$$\Phi_n(x) := x_k \left(1 + y + \frac{y^2}{2} + \frac{y^3}{3!} + \ldots + \frac{y^{n-1}}{(n-1)!}\right) \tag{17.33b}$$

As the computation of logarithms is expensive one should use an iteration of higher order. The C++ implementation given in [hfloat: src/tz/itexp.cc] uses the iteration of order 20.

Padé series $P_{[i,j]}(z)$ of $\exp(z)$ around $z = 0$ produce iterations of order $i + j + 1$. For $i = j$ we obtain

$$P_{[1,1]} = \frac{2+z}{2-z} \tag{17.34a}$$

$$P_{[2,2]} = \frac{12+6z+z^2}{12-6z+z^2} \tag{17.34b}$$

$$P_{[3,3]} = \frac{120+60z+12z^2+z^3}{120-60z+12z^2-z^3} \tag{17.34c}$$

$$P_{[4,4]} = \frac{1680+840z+180z^2+20z^3+z^4}{1680-840z+180z^2-20z^3+z^4} \tag{17.34d}$$

Note that the functional equation $\exp(-z) = \frac{1}{\exp(z)}$ holds for the (diagonal) Padé approximants.

A closed form for the $[i, j]$-th Padé approximant of $\exp(z)$ is

$$P_{[i,j]}(z) = \left\{\sum_{k=0}^{i} \frac{\binom{i}{k}}{\binom{i+j}{k}} \frac{z^k}{k!}\right\} / \left\{\sum_{k=0}^{j} \frac{\binom{j}{k}}{\binom{i+j}{k}} \frac{(-z)^k}{k!}\right\} \tag{17.35}$$

The numerator for $i = j$, multiplied by $(2i)!/i!$ in order to avoid rational coefficients, equals

$$= \frac{(2i)!}{i!} \cdot \sum_{k=0}^{i} \frac{\binom{i}{k}}{\binom{2\,i}{k}} \frac{z^k}{k!} \tag{17.36}$$

The coefficients of the numerator and denominator in the diagonal approximant

$$P_{[i,i]} = \frac{\sum_{k=0}^{i} c_k\, z^k}{\sum_{k=0}^{i} c_k\, (-z)^k} \tag{17.37}$$

can be computed using $c_i = 1$ (the coefficient of the highest power of z) and the recursion

$$c_k = c_{k+1} \frac{(k+1)\,(2i-k)}{(i-k)} \tag{17.38}$$

It is usually preferable to generate the coefficients in the other direction. To do so compute the constant c_0

$$c_0 = \prod_{w=1}^{i} 4\,w - 2 \;=\; 2,\ 12,\ 120,\ 1680,\ 30240,\ \ldots \tag{17.39}$$

and use the recurrence

$$c_k = c_{k-1} \frac{i-k}{(2i-k)\,(k+1)} \tag{17.40}$$

We generate the coefficients for $1 \le i \le 8$:

```
? c0(i)=prod(w=1,i,4*w-2)
? qq(i,k)=(i-k)/((2*i-k)*(k+1))
? for (i=1, 8, c=c0(i); print1("["i,",",i,"]:  "); \
?   for (k=0, i, print1("  ", c); c*=qq(i,k)); print();)
  [1,1]:    2  1
  [2,2]:    12  6  1
  [3,3]:    120  60  12  1
  [4,4]:    1680  840  180  20  1
  [5,5]:    30240  15120  3360  420  30  1
  [6,6]:    665280  332640  75600  10080  840  42  1
  [7,7]:    17297280  8648640  1995840  277200  25200  1512  56  1
  [8,8]:    518918400  259459200  60540480  8648640  831600  55440  2520  72  1
```

Finally, the Padé approximant $P_{[i,j]}$ can be expressed as ratio of hypergeometric functions:

$$P_{[i,j]} \quad = \quad F\left(\begin{matrix}-i\\-i-j\end{matrix}\Big|\, z\right) / F\left(\begin{matrix}-j\\-i-j\end{matrix}\Big|\, -z\right) \tag{17.41}$$

This is relation 20.93 on page 503 with $a = -i$ and $b = -j$ where i and j are positive integers.

Argument reduction for the exponential function

As for the logarithm an argument reduction technique can be useful with moderate precisions. We do not use the functional equation for the exponential functional ($\exp(2z) = \exp(z)^2$) because the loss of precision when adding up the Taylor series (one plus a tiny quantity). Instead we use the functional equation for $E(z) := \exp(z) - 1$:

$$E(2z) \quad = \quad 2\,E(z) + E^2(z) \tag{17.42}$$

Compute the exponential function of z as follows

1. set $r = z/2^n$ (for n big enough)
2. compute $E := \exp(r) - 1$ to the desired precision using the Taylor series
3. repeat n times: $E = 2\,E + E^2$
4. return $E + 1$

We compute $\exp(1.0)$ using $n = 16$ and four terms of the Taylor series:

```
? z=1.0;
? n=16;
? r=z/2^n
  0.000015258789062500000000000000000000000000000000000000000000000
? E=r+1/2*r^2+1/6*r^3+1/24*r^4
  0.000015258905478413948140035729291360174405175105979045232l3
? for(k=1,n,E=2*E+E^2);E
  1.718281828459045235359059500690612039181007440391110888
? E+1
  2.718281828459045235359059500690612039181007440391110888
```

One can also compute the exponential function via the hyperbolic cosine using

$$\exp(z) \quad = \quad \cosh(z) + \sinh(z) \;=\; \cosh(z) + \sqrt{\cosh^2(z) - 1} \tag{17.43}$$

The advantage is that half of the coefficients of the Taylor series are zero. Again we do not use the function equation for the cosine ($\cos(2z) = 2\,\cos^2(z) - 1$) but that for $C(z) := \cos(z) - 1$:

$$C(2z) \quad = \quad 2\,(C(z) + 1)^2 - 2 \tag{17.44}$$

Compute the hyperbolic cosine as follows

1. set $r = z/2^n$ (for n big enough)
2. compute $C := \cosh(r) - 1$ to the desired precision using the Taylor series

3. repeat n times: $C = 2\,[C-1]^2 - 2$
4. return $C + 1$

We compute cosh(1.5) using $n = 16$ and four terms of the Taylor series:

```
? z=1.50;
? n=16;
? r=z/2^n
? C=1/2*r^2+1/24*r^4+1/720*r^6+1/40320*r^8
  0.000000000261934474122038277307663984994058290343310054242645592 3
? for(k=1,n,C=2*(C+1)^2-2);C
  1.352409615243247325767667965441644170173960682574839216
? C+1
  2.352409615243247325767667965441644170173960682574839216
? cosh(z) \\ check with builtin cosh
  2.352409615243247325767667965441644170173960748865373192
```

When the Taylor series for $\cos(z) - 1$ is used then the cosine can be obtained by the identical algorithm:

```
? z=1.50;
? n=16;
? r=z/2^n
  0.000022888183593750000000000000000000000000000000000000000000000
? C=-1/2*r^2+1/24*r^4-1/720*r^6+1/40320*r^8
  -0.000000000261934474099168387731797875839209846883179633343528818 1
? for(k=1,n,C=2*(C+1)^2-2);C
  -0.929262798332297089911810148565731290914908941381793162 3
? C+1
  0.070737201667702910088189851434268709085091058618206837 69
? cos(z)  \\ check with builtin cos
  0.070737201667702910088189851434268709085091027563346869 42
```

Compute the sine function as $\sin(z) = \sqrt{1 - \cos^2(z)}$ and the tangent function as $\tan(z) = \frac{\sin(z)}{\cos(z)}$.

17.4 Logarithms of power series *

The computation of the logarithm and related functions turns out to be surprisingly simple with power series.

Logarithm

Let $f(x)$ be a power series in x and $g(x) = \log(f(x))$. Then we have $\frac{d\,g(x)}{dx} = \frac{f'(x)}{f(x)}$. Thereby, symbolically,

$$g(x) \quad = \quad \log(f(x)) \;=\; \int \frac{f'(x)}{f(x)}\,dx \tag{17.45}$$

A few lines of pari/gp to demonstrate this:

```
? sp=8;default(seriesprecision,sp+1);
? f=taylor((1)/(1-x-x^2),x) /* shifted Fibonacci (with constant term) */
 1 + x + 2*x^2 + 3*x^3 + 5*x^4 + 8*x^5 + 13*x^6 + 21*x^7 + 34*x^8 + O(x^9)

? d=deriv(f,x)
 1 + 4*x + 9*x^2 + 20*x^3 + 40*x^4 + 78*x^5 + 147*x^6 + 272*x^7 + O(x^8)

? q=d/f  /* the only nontrivial computation */
 1 + 3*x + 4*x^2 + 7*x^3 + 11*x^4 + 18*x^5 + 29*x^6 + 47*x^7 + O(x^8)

? lf=intformal(q)
 x + 3/2*x^2 + 4/3*x^3 + 7/4*x^4 + 11/5*x^5 + 3*x^6 + 29/7*x^7 + 47/8*x^8 + O(x^9)

? f-exp(lf)  /* check with builtin exp() */
 O(x^9)
```

Inverse trigonometric functions

Now let $a(x) = \arctan(f(x))$. Then, symbolically,

$$a(x) \quad = \quad \int \frac{f'(x)}{1+f(x)^2}\, dx \tag{17.46}$$

Verification for the trivial case $f(x) = x$:

```
? sp=13;default(seriesprecision,sp+1);
? f=taylor(x,x)
 x + O(x^14)
? d=deriv(f,x)
 1 + O(x^13)
? q=d/(1+f^2)
 1 - x^2 + x^4 - x^6 + x^8 - x^10 + x^12 + O(x^13)
? af=intformal(q,x)
 x - 1/3*x^3 + 1/5*x^5 - 1/7*x^7 + 1/9*x^9 - 1/11*x^11 + 1/13*x^13 + O(x^14)
? f-tan(af)  /* check with builtin tan() */
 O(x^14)
```

For $s(x) = \arcsin(f(x))$:

$$s(x) \quad = \quad \int \frac{f'(x)}{\sqrt{1-f(x)^2}}\, dx \tag{17.47}$$

Exponential function

With $e(x) = \exp(f(x))$ we can use a scheme similar to the one shown in section 15.7 on page 415. We express a function $g(y)$ as

$$g(y) \quad = \quad \prod_{k=1}^{\infty} 1 + T(Y_k) \tag{17.48}$$

where $Y_1 = y$, $Y_{k+1} = N(Y_k)$ and $1 + T(y)$ is the truncated Taylor series of $g(.)$. A second order product is obtained by taking $1 + T(y) = 1 + y$ (the series of $\exp(y)$ truncated before the second term) and

$$N(y) \quad = \quad f^{-1}\left(\frac{f(y)}{1+T(y)}\right) \tag{17.49}$$

For $g(y) = \exp(y)$ one gets $N(y) = y - \log(1+y)$ and

$$\exp(y) \quad = \quad [1+y]\,[1+y-\log(1+y)]\,[1+y-\log(1+y)-\log(1+y-\log(1+y))]\,\ldots \tag{17.50}$$

$$= \quad \prod_{k=1}^{\infty} 1 + Y_k \tag{17.51}$$

where $Y_1 = y = f(x)$ and $Y_{k=1} = Y_k - \log(1+Y_k)$. The product $\prod_{k=1}^{N}$ is correct up to order y^{2^N-1}. The computation involves $N-2$ logarithms and $N-1$ multiplications.

Implementation in pari/gp:

```
texp(y, N=5)=
{
    local(Y, e, t);
    Y = y;  e = 1 + Y;
    for (k=2, N,
        t = deriv(1+Y,x)/(1+Y);
        t = intformal(t);  \\ here: t = log(1+Y);
        Y -= t;
        e *= (1+Y);
    );
    return( e );
}
```

Check:

```
? f=taylor((x)/(1-x-x^2),x)
  x + x^2 + 2*x^3 + 3*x^4 + 5*x^5 + 8*x^6 + 13*x^7 + 21*x^8 + ...
? e=exp(f)  /* builtin exp() */
  1 + x + 3/2*x^2 + 19/6*x^3 + 145/24*x^4 + 467/40*x^5 + 16051/720*x^6 + ...
? t=texp(f,4);
? t-e
  -1/32768*x^16 - 35/98304*x^17 - ...
```

17.5 The elliptic functions K and E

17.5.1 Elliptic K

The function K can be defined as

$$K(k) \;=\; \int_0^{\pi/2} \frac{d\,\vartheta}{\sqrt{1-k^2\,\sin^2\vartheta}} \;=\; \int_0^1 \frac{d\,t}{\sqrt{(1-t^2)\,(1-k^2t^2)}} \tag{17.52}$$

One has

$$K(k) \;=\; \frac{\pi}{2}\,F\left(\begin{matrix}\frac{1}{2},\frac{1}{2}\\ 1\end{matrix}\,\middle|\,k^2\right) \tag{17.53a}$$

$$\;=\; \frac{\pi}{2}\sum_{i=0}^{\infty}\left(\frac{(2\,i-1)!!}{2^i\,i!}\right)^2 k^{2i} \;=\; \frac{\pi}{2}\sum_{i=0}^{\infty}\left(\frac{\binom{2\,i}{i}}{4^i}\right)^2 k^{2i} \tag{17.53b}$$

$$\;=\; \frac{\pi}{2}\left(1+\left(\frac{1}{2}\right)^2 k^2+\left(\frac{1\cdot 3}{2\cdot 4}\right)^2 k^4+\left(\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\right)^2 k^6+\ldots\right) \tag{17.53c}$$

$$\;=\; \frac{\pi}{2}\left(1+\frac{1}{4}\,k^2+\frac{9}{64}\,k^4+\frac{25}{256}\,k^6+\frac{1225}{16384}\,k^8+\frac{3969}{65536}\,k^{10}+\ldots\right) \tag{17.53d}$$

See section 20.4.4 on page 508 for transformations in terms of hypergeometric functions.

A special value is

$$K(0) \;=\; \frac{\pi}{2} \tag{17.54}$$

Further, $\lim_{k\to 1-} K(k) = +\infty$.

The computational interesting form is $F\left(\begin{matrix}\frac{1}{2},\frac{1}{2}\\ 1\end{matrix}\,\middle|\,z\right) = 1/\operatorname{AGM}(1,\sqrt{1-z})$:

$$K(k) \;=\; \frac{\pi}{2\,\operatorname{AGM}(1,k')} \;=\; \frac{\pi}{2\,\operatorname{AGM}(1,\sqrt{1-k^2})} \tag{17.55a}$$

One defines $k' = \sqrt{1-k^2}$ and K' as

$$K'(k) \;:=\; K(k') \;=\; K(\sqrt{1-k^2}) \;=\; \frac{\pi}{2\,\operatorname{AGM}(1,k)} \tag{17.55b}$$

A C++ implementation of the AGM based computation is given in [hfloat: src/tz/elliptic-k.cc].

Product forms for K and K' that are also candidates for fast computations are

$$\frac{2}{\pi}\,K'(k_0) \;=\; \prod_{n=0}^{\infty}\frac{2}{1+k_n} \;=\; \prod_{n=1}^{\infty} 1+k'_n \tag{17.56}$$

$$\text{where}\quad k_{n+1} \;:=\; \frac{2\sqrt{k_n}}{1+k_n},\quad 0<k_0\le 1$$

$$\frac{2}{\pi}\,K'(k_0) \;=\; \prod_{n=0}^{\infty}\frac{1}{\sqrt{k_n}} \tag{17.57}$$

$$\text{where}\quad k_{n+1} \;:=\; \frac{1+k_n}{2\sqrt{k_n}},\quad 0<k_0\le 1$$

The second form seems computationally especially attractive since, apart from the multiplication with the main product, only a inverse square root needs to be computed per step. The product formulas follow directly from 17.55b (and $\mathrm{AGM}(a,b) = a\ \mathrm{AGM}(1,b/a)$):

$$\frac{1}{\mathrm{AGM}(1,k)} \;=\; \left[\mathrm{AGM}\left(\frac{1+k}{2},\sqrt{k}\right)\right]^{-1} \tag{17.58a}$$

$$\;=\; \left[\frac{1+k}{2}\ \mathrm{AGM}\left(1,\frac{2\sqrt{k}}{1+k}\right)\right]^{-1} \qquad \text{(first form)} \tag{17.58b}$$

$$\;=\; \left[\sqrt{k}\ \mathrm{AGM}\left(\frac{1+k}{2\sqrt{k}},1\right)\right]^{-1} \qquad \text{(second form)} \tag{17.58c}$$

Similarly,

$$\frac{2}{\pi}\,K(k_0) \;=\; \prod_{n=0}^{\infty}\frac{2}{1+k'_n} \;=\; \prod_{n=1}^{\infty} 1+k_n \tag{17.59}$$

$$\text{where}\quad k_{n+1} \;:=\; \frac{1-k'_n}{1+k'_n} \;=\; \frac{1-\sqrt{1-k_n^2}}{1+\sqrt{1-k_n^2}},\quad 0<k_0\le 1$$

17.5.2 Elliptic E

The function E can be defined as

$$E(k) \;=\; \int_0^{\pi/2}\sqrt{1-k^2\,\sin^2\vartheta}\,d\,\vartheta \;=\; \int_0^1 \frac{\sqrt{1-k^2t^2}}{\sqrt{1-t^2}}\,d\,t \tag{17.60}$$

One has

$$E(k) \;=\; \frac{\pi}{2}\,F\left(\begin{matrix}-\frac{1}{2},\frac{1}{2}\\ 1\end{matrix}\,\middle|\,k^2\right) \tag{17.61a}$$

$$\;=\; \frac{\pi}{2}\left(-\sum_{i=0}^{\infty}\left(\frac{(2\,i-1)!!}{2^i\,i!}\right)^2\frac{k^{2i}}{2i-1}\right) \;=\; \frac{\pi}{2}\sum_{i=0}^{\infty}\left(\frac{\binom{2\,i}{i}}{4^i}\right)^2\frac{k^{2i}}{2\,i-1} \tag{17.61b}$$

$$\;=\; \frac{\pi}{2}\left(1-\left(\frac{1}{2}\right)^2k^2-\left(\frac{1\cdot 3}{2\cdot 4}\right)^2\frac{k^4}{3}-\left(\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\right)^2\frac{k^6}{5}-\ldots\right) \tag{17.61c}$$

$$\;=\; \frac{\pi}{2}\left(1-\frac{1}{4}\,k^2-\frac{3}{64}\,k^4-\frac{5}{256}\,k^6-\frac{175}{16384}\,k^8-\frac{441}{65536}\,k^{10}-\ldots\right) \tag{17.61d}$$

Special values are

$$E(0) \;=\; \frac{\pi}{2} \qquad\qquad E(1) \;=\; 1 \tag{17.62}$$

The latter leads to a (slowly converging) series for $2/\pi$:

$$\frac{2}{\pi} \;=\; F\left(\begin{matrix}-\frac{1}{2},\frac{1}{2}\\ 1\end{matrix}\,\middle|\,1\right) \tag{17.63}$$

Similarly as for K' one defines

$$E'(k) \;:=\; E(k') \;=\; E(\sqrt{1-k^2}) \tag{17.64}$$

The key to fast computations is given by the relation

$$\frac{E}{K} \;=\; 1-\frac{1}{2}\sum_{n=0}^{\infty} 2^n {c'}_n^2 \tag{17.65}$$

The terms c' in the sum occur naturally during the computation of $\mathrm{AGM}(1,\sqrt{1-k^2}) = \mathrm{AGM}(1,k')$. One defines

$$R \;:=\; \frac{E}{K} \qquad R' \;:=\; \frac{E'}{K'} \tag{17.66}$$

Then E can be computed via

$$E(k) \;=\; R(k)\,K(k) \;=\; \frac{\pi}{2\ \mathrm{AGM}(1,\sqrt{1-k^2})}\cdot\left(1-\sum_{n=0}^{\infty} 2^{n-1}{c'}_n^2\right) \tag{17.67}$$

Legendre's relation between K and E is (arguments omitted for readability, choose your favorite form):

$$\frac{E}{K}+\frac{E'}{K'}-1 \;=\; \frac{\pi}{2\,K\,K'} \tag{17.68a}$$

$$E\,K'+E'\,K-K\,K' \;=\; \frac{\pi}{2} \tag{17.68b}$$

Equivalently,

$$\mathrm{AGM}(1,k) \;=\; \frac{E/K}{(1-E'/K')} \;=\; \frac{R}{(1-R')} \tag{17.69}$$

For $k=\frac{1}{\sqrt{2}} =: s$ we have $k=k'$, thereby $K=K'$ and $E=E'$, so

$$\frac{K(s)}{\pi}\left(\frac{2\,E(s)}{\pi}-\frac{K(s)}{\pi}\right) \;=\; \frac{1}{2\,\pi} \tag{17.70}$$

As expressions 17.55a on page 439 and 17.67 provide a fast AGM based computation of $\frac{K}{\pi}$ and $\frac{E}{\pi}$ the above formula can be used to compute π.

Using $E-K = k\,k'\,\frac{dK}{dk} - k^2\,K$ one can express the derivative of K in terms of E and K and thereby compute that quantity fast:

$$\frac{dK}{dk} \;=\; \frac{E-k'^2\,K}{k\,k'^2} \tag{17.71}$$

For the derivative of E we have

$$\frac{dE}{dk} \;=\; \frac{E-K}{k} \tag{17.72}$$

17.6 Computation of $\pi/\log(q)$

For the computation of the natural logarithm one can use the relation

$$\log(m\,r^x) \;=\; \log(m) + x\,\log(r) \tag{17.73}$$

where m is the mantissa and r the radix of the floating point numbers.

There is a nice way to compute the value of $\log(r)$ if the value of π has been precomputed. One defines

$$\Theta_2(q) \;=\; \sum_{n=-\infty}^{\infty} q^{(n+1/2)^2} \;=\; 2\left(q^{1/4} + q^{9/4} + q^{25/4} + q^{81/4} + q^{121/4} + \ldots\right) \tag{17.74a}$$

$$\;=\; 2\,q^{1/4}\left(1 + q^2 + q^6 + q^{12} + q^{20} + \ldots\right) \tag{17.74b}$$

$$\Theta_3(q) \;=\; \sum_{n=-\infty}^{\infty} q^{n^2} \;=\; 1 + 2\left(q + q^4 + q^9 + q^{16} + q^{25} + \ldots\right) \tag{17.74c}$$

$$\Theta_4(q) \;=\; \sum_{n=-\infty}^{\infty} (-1)^n\,q^{n^2} \;=\; 1 + 2\left(-q + q^4 - q^9 + q^{16} - q^{25} \pm \ldots\right) \tag{17.74d}$$

Then (see [PiAGM] p.225)

$$\log\frac{1}{q} \;=\; -\log q \;=\; \pi\,\frac{K'}{K} \tag{17.75}$$

so $q = \exp(-\pi\,K'/K)$ and

$$\frac{\pi}{\log(1/q)} \;=\; -\frac{\pi}{\log(q)} \;=\; \mathrm{AGM}\left(\Theta_3^2(q),\,\Theta_2^2(q)\right) \;=\; \frac{\mathrm{AGM}(1,\,k)}{\mathrm{AGM}(1,\,k')} \tag{17.76}$$

Computing $\Theta_3(q)$ is easy if $q = 1/r$:

$$\Theta_3(q) \;=\; 1 + 2\sum_{n=1}^{\infty} q^{n^2} \;=\; 2\,(1 + \sum_{n=1}^{\infty} q^{n^2}) - 1 \tag{17.77}$$

However, the computation of $\Theta_2(q)$ suggests to choose $q = 1/r^4 =: b^4$:

$$\Theta_2(q) \;=\; 0 + 2\sum_{n=0}^{\infty} q^{(n+1/2)^2} \;=\; 2\sum_{n=0}^{\infty} b^{4n^2+4n+1} \quad \text{where} \quad q = b^4 \tag{17.78}$$

$$\;=\; 2\,b\sum_{n=0}^{\infty} q^{n^2+n} \;=\; 2\,b\,(1 + \sum_{n=1}^{\infty} q^{n^2+n}) \tag{17.79}$$

Functions that compute $\Theta_2(b^4)$, $\Theta_3(b^4)$ and $\pi/\log(b)$ where b is the used radix are given in [hfloat: src/tz/pilogq.cc].

We list a few relations involving the theta functions (see, for example [ModAnl]):

$$\Theta_3^4(q) \;=\; \Theta_2^4(q) + \Theta_4^4(q) \tag{17.80}$$

$$\Theta_3(q) \;=\; \Theta_3(q^4) + \Theta_2(q^4) \qquad \Theta_4(q) \;=\; \Theta_3(q^4) - \Theta_2(q^4) \tag{17.81}$$

$$\Theta_2^2(q) \;=\; \frac{2\,k\,K}{\pi} \qquad \Theta_3^2(q) \;=\; \frac{2\,K}{\pi} \qquad \Theta_4^2(q) \;=\; \frac{2\,k'\,K}{\pi} \tag{17.82}$$

$$k \;=\; \frac{\Theta_2^2(q)}{\Theta_3^2(q)} \qquad\qquad k' \;=\; \frac{\Theta_4^2(q)}{\Theta_3^2(q)} \tag{17.83}$$

$$1 \;=\; \mathrm{AGM}\left(\Theta_3^2(q), \Theta_4^2(q)\right) \tag{17.84}$$

$$q \prod_{n=1}^{\infty} (1-q^n)^{24} \;=\; \frac{256}{\pi^{12}}\, k^2\, k'^8\, K^{12} \tag{17.85}$$

17.7 Computation of $q = \exp(-\pi\, K'/K)$

We use (see [King] p.35/36)

$$q \;=\; \exp\left(-\pi\,\frac{K'}{K}\right) \tag{17.86}$$

Now write

$$\frac{K'}{K} \;=\; \frac{\mathrm{AGM}(1,k')}{\mathrm{AGM}(1,k)} \;=\; \frac{\mathrm{AGM}(1,b_0)}{\mathrm{AGM}(1,b'_0)} \tag{17.87}$$

where $k' = b_0$ and $k = b'_0 = \sqrt{1-b_0^2}$ and use ([King] p.38, note the missing '4' there)

$$\frac{\pi}{2}\,\frac{\mathrm{AGM}(1,b_0)}{\mathrm{AGM}(1,b'_0)} \;=\; \lim_{n\to\infty} \frac{1}{2^n} \log \frac{4\,a_n}{c_n} \tag{17.88}$$

thereby

$$q \;=\; \exp\left(-2 \lim_{n\to\infty} \frac{1}{2^n}\, \log \frac{4\,a_n}{c_n}\right) \;=\; \lim_{n\to\infty} \exp\left(-\frac{1}{2^{n-1}}\, \log \frac{4\,a_n}{c_n}\right) \tag{17.89}$$

$$\;=\; \lim_{n\to\infty} \left(\exp \log \frac{4\,a_n}{c_n}\right)^{-1/2^{n-1}} \;=\; \lim_{n\to\infty} \left(\frac{4\,a_n}{c_n}\right)^{-1/(2^{n-1})} \tag{17.90}$$

This gives

$$q \;=\; \lim_{n\to\infty} \left(\frac{c_n}{4\,a_n}\right)^{1/(2^{n-1})} \tag{17.91}$$

One obtains an algorithm for $\exp(-x)$ by first solving for k, k' so that $x = \pi\, K'/K$ (precomputed π) and applying the last relation that implies the computation of a 2^{n-1}-th root. Note that the quantity c should be computed via $c_{n+1} = \frac{c_n^2}{4\,a_{n+1}}$ throughout the AGM computation in order to preserve its accuracy (see also [PiAGM] p.227).

For $k = 1/\sqrt{2} =: s$ one has $k = k'$ and so $q = \exp(-\pi)$. Thus the calculation of $\exp(-\pi) = 0.0432139182637\ldots$ can directly be done via a single AGM computation as $(c_n/(4a_n))^N$ where $N = 1/2^{(n-1)}$. The quantity $i^i = \exp(-\pi/2) = 0.2078795763507\ldots$ can be obtained using $N = 1/2^n$ (see also [King] p.13).

17.8 Computation of π

In this section we give various iterations for computing π with at least quadratic convergence. The number of full precision multiplications (FPM) are an indication of the efficiency of the algorithm. The approximate number of FPMs that were counted with a computation of π to 4 million decimal digits (using radix $10,000$ and 1 million LIMBs) is indicated like this: #FPM=123.4. The number is computed as one third of the FFT-work in terms of full precision real valued FFTs.

AGM implemented in [hfloat: src/pi/piagm.cc], #FPM=98.4:

$$a_0 = 1 \tag{17.92a}$$

$$b_0 = \frac{1}{\sqrt{2}} \tag{17.92b}$$

$$a_{k+1} = \frac{a_k + b_k}{2} \tag{17.92c}$$

$$b_{k+1} = \sqrt{a_k\, b_k} \tag{17.92d}$$

$$p_n = \frac{2\, a_{n+1}^2}{1 - \sum_{k=0}^{n} 2^k\, c_k^2} \quad \to \pi \tag{17.92e}$$

$$\pi - p_n = \frac{\pi^2\, 2^{n+4}\, e^{-\pi\, 2^{n+1}}}{\mathrm{AGM}^2(a_0, b_0)} \tag{17.92f}$$

Convergence is second order. Computing π based on the fourth order AGM (relations 17.8a... 17.8e on page 430) is possible by setting the second argument of the routine (#FPM=149.3 for the quartic variant). Schönhage's variant of the AGM computation (relations 17.4a... 17.4k on page 430) is implemented in [hfloat: src/pi/piagmsch.cc] (#FPM=78.424).

The AGM method goes back to Gauss, a facsimile of the entry in his 1809 handbook 6 is given on page 101 in [PiUnl]. The entry states that

$$\pi = \frac{\mathrm{AGM}(1, k)\ \mathrm{AGM}(1, k')}{1 - \sum_{k=0}^{\infty} 2^{k-1}\, (c_k^2 + c'^2_k)} \tag{17.93}$$

where $k' = b_0/a_0$ and $k = \sqrt{1 - b_0^2/a_0^2} = c_0/a_0$. For $k = k' = 1/\sqrt{2}$ one gets relation 17.92e. The formula appeared also 1924 in [King], page 39.

AGM variant, implemented in [hfloat: src/pi/piagm3.cc], #FPM=99.5 (#FPM=155.3 for the quartic variant):

$$a_0 = 1 \tag{17.94a}$$

$$b_0 = \frac{\sqrt{6} + \sqrt{2}}{4} \tag{17.94b}$$

$$p_n = \frac{2\, a_{n+1}^2}{\sqrt{3}\,(1 - \sum_{k=0}^{n} 2^k\, c_k^2) - 1} \quad \to \pi \tag{17.94c}$$

$$\pi - p_n < \frac{\sqrt{3}\,\pi^2\, 2^{n+4}\, e^{-\sqrt{3}\,\pi\, 2^{n+1}}}{\mathrm{AGM}^2(a_0, b_0)} \tag{17.94d}$$

AGM variant, implemented in [hfloat: src/pi/piagm3.cc], #FPM=108.2 (#FPM=169.5 for the quartic variant):

$$a_0 = 1 \tag{17.95a}$$

$$b_0 = \frac{\sqrt{6} - \sqrt{2}}{4} \tag{17.95b}$$

$$p_n = \frac{6\,a_{n+1}^2}{\sqrt{3}\,(1-\sum_{k=0}^{n} 2^k\,c_k^2)+1} \quad \to \pi \tag{17.95c}$$

$$\pi - p_n < \frac{\frac{1}{\sqrt{3}}\,\pi^2\,2^{n+4}\,e^{-\frac{1}{\sqrt{3}}\,\pi\,2^{n+1}}}{\mathrm{AGM}(a_0,b_0)^2} \tag{17.95d}$$

Borwein's quartic (fourth order) iteration, variant $r = 4$, implemented in [hfloat: src/pi/pi4th.cc], #FPM=170.5:

$$y_0 = \sqrt{2}-1 \tag{17.96a}$$

$$a_0 = 6-4\sqrt{2} \tag{17.96b}$$

$$y_{k+1} = \frac{1-(1-y_k^4)^{1/4}}{1+(1-y_k^4)^{1/4}} \quad \to 0+ \tag{17.96c}$$

$$= \frac{(1-y_k^4)^{-1/4}-1}{(1-y_k^4)^{-1/4}+1} \tag{17.96d}$$

$$a_{k+1} = a_k\,(1+y_{k+1})^4 - 2^{2k+3}\,y_{k+1}\,(1+y_{k+1}+y_{k+1}^2) \quad \to \frac{1}{\pi} \tag{17.96e}$$

$$= a_k\,((1+y_{k+1})^2)^2 - 2^{2k+3}\,y_{k+1}\,((1+y_{k+1})^2-y_{k+1}) \tag{17.96f}$$

$$0 < a_k - \pi^{-1} \le 16\cdot 4^n\,2\,e^{-4^n\,2\,\pi} \tag{17.96g}$$

Identities 17.96d and 17.96f show how to save operations.

Borwein's quartic (fourth order) iteration, variant $r = 16$, implemented in [hfloat: src/pi/pi4th.cc], #FPM=164.4:

$$y_0 = \frac{1-2^{-1/4}}{1+2^{-1/4}} \tag{17.97a}$$

$$a_0 = \frac{8/\sqrt{2}-2}{(2^{-1/4}+1)^4} \tag{17.97b}$$

$$y_{k+1} = \frac{(1-y_k^4)^{-1/4}-1}{(1-y_k^4)^{-1/4}+1} \quad \to 0+ \tag{17.97c}$$

$$a_{k+1} = a_k\,(1+y_{k+1})^4 - 2^{2k+4}\,y_{k+1}\,(1+y_{k+1}+y_{k+1}^2) \quad \to \frac{1}{\pi} \tag{17.97d}$$

$$0 < a_k - \pi^{-1} \le 16\cdot 4^n\,4\,e^{-4^n\,4\,\pi} \tag{17.97e}$$

Same operation count as before, but this variant gives approximately twice as much precision after the same number of steps.

The general form of the quartic iterations (17.96a... and 17.97a...) is (see [PiAGM], p.170f)

$$y_0 = \sqrt{\lambda^*(r)} \tag{17.98a}$$

$$a_0 = \alpha(r) \tag{17.98b}$$

$$y_{k+1} = \frac{(1-y_k^4)^{-1/4}-1}{(1-y_k^4)^{-1/4}+1} \quad \to 0+ \tag{17.98c}$$

$$a_{k+1} = a_k\,(1+y_{k+1})^4 - 2^{2k+2}\,\sqrt{r}\,y_k\,(1+y_{k+1}+y_{k+1}^2) \quad \to \frac{1}{\pi} \tag{17.98d}$$

$$0 < a_k - \pi^{-1} \le \quad 16\cdot 4^n\,\sqrt{r}\,e^{-4^n\,\sqrt{r}\,\pi} \tag{17.98e}$$

Derived AGM iteration (second order, from [PiAGM], p.46f), implemented in [hfloat: src/pi/pideriv.cc], #FPM=276.2:

$$x_0 = \sqrt{2} \tag{17.99a}$$
$$p_0 = 2+\sqrt{2} \tag{17.99b}$$
$$y_1 = 2^{1/4} \tag{17.99c}$$

$$x_{k+1} = \frac{1}{2}\left(\sqrt{x_k}+\frac{1}{\sqrt{x_k}}\right) \quad (k\geq 0) \quad \to 1+ \tag{17.99d}$$
$$y_{k+1} = \frac{y_k\sqrt{x_k}+\frac{1}{\sqrt{x_k}}}{y_k+1} \quad (k\geq 1) \quad \to 1+ \tag{17.99e}$$
$$p_{k+1} = p_k\,\frac{x_k+1}{y_k+1} \quad (k\geq 1) \quad \to \pi+ \tag{17.99f}$$
$$p_k-\pi < 10^{-2^{k+1}} \tag{17.99g}$$

Cubic AGM iteration (third order) from [26], implemented in [hfloat: src/pi/picubagm.cc], #FPM=182.7:

$$a_0 = 1 \tag{17.100a}$$
$$b_0 = \frac{\sqrt{3}-1}{2} \tag{17.100b}$$

$$a_{n+1} = \frac{a_n+2\,b_n}{3} \tag{17.100c}$$
$$b_{n+1} = \sqrt[3]{\frac{b_n\,(a_n^2+a_n\,b_n+b_n^2)}{3}} \tag{17.100d}$$
$$p_n = \frac{3\,a_n^2}{1-\sum_{k=0}^{n}3^k\,(a_k^2-a_{k+1}^2)} \tag{17.100e}$$

Second order iteration, as in [hfloat: src/pi/pi2nd.cc], #FPM=255.7:

$$y_0 = \frac{1}{\sqrt{2}} \tag{17.101a}$$
$$a_0 = \frac{1}{2} \tag{17.101b}$$

$$y_{k+1} = \frac{1-(1-y_k^2)^{1/2}}{1+(1-y_k^2)^{1/2}} \quad \to 0+ \tag{17.101c}$$
$$= \frac{(1-y_k^2)^{-1/2}-1}{(1-y_k^2)^{-1/2}+1} \tag{17.101d}$$
$$a_{k+1} = a_k\,(1+y_{k+1})^2-2^{k+1}\,y_{k+1} \quad \to \frac{1}{\pi} \tag{17.101e}$$
$$a_k-\pi^{-1} \leq 16\cdot 2^{k+1}\,e^{-2^{k+1}\,\pi} \tag{17.101f}$$

Relation 17.101d shows how to save 1 multiplication per step (see section 15.1).

Quintic (5th order) iteration, implemented in [hfloat: src/pi/pi5th.cc], #FPM=353.2:

$$s_0 = 5(\sqrt{5}-2) \tag{17.102a}$$
$$a_0 = \frac{1}{2} \tag{17.102b}$$

$$s_{n+1} = \frac{25}{s_n(z+x/z+1)^2} \quad \to 1 \tag{17.102c}$$
$$\text{where } x = \frac{5}{s_n} - 1 \quad \to 4 \tag{17.102d}$$
$$\text{and } y = (x-1)^2 + 7 \quad \to 16 \tag{17.102e}$$
$$\text{and } z = \left(\frac{x}{2}\left(y + \sqrt{y^2 - 4x^3}\right)\right)^{1/5} \quad \to 2 \tag{17.102f}$$
$$a_{n+1} = s_n^2 a_n - 5^n \left(\frac{s_n^2 - 5}{2} + \sqrt{s_n\left(s_n^2 - 2s_n + 5\right)}\right) \quad \to \frac{1}{\pi} \tag{17.102g}$$
$$a_n - \frac{1}{\pi} < 16 \cdot 5^n \, e^{-\pi\, 5^n} \tag{17.102h}$$

Cubic (third order) iteration from [14], implemented in [hfloat: src/pi/pi3rd.cc], #FPM=200.3:

$$a_0 = \frac{1}{3} \tag{17.103a}$$
$$s_0 = \frac{\sqrt{3}-1}{2} \tag{17.103b}$$

$$r_{k+1} = \frac{3}{1 + 2\,(1 - s_k^3)^{1/3}} \tag{17.103c}$$
$$s_{k+1} = \frac{r_{k+1} - 1}{2} \tag{17.103d}$$
$$a_{k+1} = r_{k+1}^2\, a_k - 3^k\,(r_{k+1}^2 - 1) \quad \to \frac{1}{\pi} \tag{17.103e}$$

Nonic (9th order) iteration from [14], implemented in [hfloat: src/pi/pi9th.cc], #FPM=273.7:

$$a_0 = \frac{1}{3} \tag{17.104a}$$
$$r_0 = \frac{\sqrt{3}-1}{2} \tag{17.104b}$$
$$s_0 = (1 - r_0^3)^{1/3} \tag{17.104c}$$

$$t = 1 + 2\,r_k \tag{17.104d}$$
$$u = \left(9\,r_k\,(1 + r_k + r_k^2)\right)^{1/3} \tag{17.104e}$$
$$v = t^2 + t\,u + u^2 \tag{17.104f}$$
$$m = \frac{27\,(1 + s_k + s_k^2)}{v} \tag{17.104g}$$
$$a_{k+1} = m\,a_k + 3^{2\,k-1}\,(1 - m) \quad \to \frac{1}{\pi} \tag{17.104h}$$
$$s_{k+1} = \frac{(1 - r_k)^3}{(t + 2\,u)\,v} \tag{17.104i}$$
$$r_{k+1} = (1 - s_k^3)^{1/3} \tag{17.104j}$$

Measured timings and operation counts

#FPM	order	routine in hfloat	time
78.424	2	pi_agm_sch()	76 sec
98.424	2	pi_agm()	93 sec
99.510	2	pi_agm3(fast variant)	94 sec
108.241	2	pi_agm3(slow variant)	103 sec
149.324	4	pi_agm(quartic)	139 sec
155.265	4	pi_agm3(quartic, fast variant)	145 sec
164.359	4	pi_4th_order(r=16 variant)	154 sec
169.544	4	pi_agm3(quartic, slow variant)	159 sec
170.519	4	pi_4th_order(r=4 variant)	160 sec
182.710	3	pi_cubic_agm()	173 sec
200.261	3	pi_3rd_order()	189 sec
255.699	2	pi_2nd_order()	240 sec
273.763	9	pi_9th_order()	256 sec
276.221	2	pi_derived_agm()	259 sec
353.202	5	pi_5th_order()	329 sec

Figure 17.1: Measured operations counts and timings for various iterations for the computation of π.

The operation counts and timings for the algorithms given so far when computing π to 4 million decimal digits (using 1 million `LIMB`s and radix 10,000) are shown in figure 17.1. In view of these figures it seems surprising that the quartic algorithms `pi_4th_order()` and the quartic AGM `pi_agm(quartic)` are usually considered close competitors to the second order AGM schemes.

Apart from the operation count the number of variables used has to be taken into account. The algorithms using more variables (like `pi_5th_order()`) cannot be used to compute as many digits as those using only a few (notably the AGM-schemes) given a fixed amount of RAM. Higher order algorithms tend to require more variables.

A further disadvantage of the algorithms of higher order is the more discontinuous growth of the work: If just a few more digits are to be computed than are available after step k then an additional step is required. Consider an extreme case where an algorithm T of order $1,000$ would compute 1 million digits after the second step, at a slightly lower cost than the most effective competitor. Then algorithm T would likely be the 'best' one only for small ranges in the number of digits around the values 10^3, 10^6, 10^9, ...

Finally, it is much easier to find special arithmetical optimizations for the 'simple' (low order) algorithms, Schönhage's AGM variant being the prime example.

More iterations for π

The following iterations are not implemented in hfloat.

A third order algorithm from [24]:

$$v_0 = 2^{-1/8} \tag{17.105a}$$

$$v_1 = 2^{-7/8}\left((1-3^{1/2})\,2^{-1/2}+3^{1/4}\right) \tag{17.105b}$$

$$w_0 = 1 \tag{17.105c}$$

$$\alpha_0 = 1 \tag{17.105d}$$

$$\beta_0 = 0 \tag{17.105e}$$

$$v_{n+1} = v_n^3 - \left\{ v_n^6 + \left[4v_n^2 (1 - v_n^8) \right]^{1/3} \right\}^{1/2} + v_{n-1} \tag{17.105f}$$

$$w_{n+1} = \frac{2v_n^3 + v_{n+1} \left(3v_{n+1}^2 v_n^2 - 1 \right)}{2v_{n+1}^3 - v_n \left(3v_{n+1}^2 v_n^2 - 1 \right)} w_n \tag{17.105g}$$

$$\alpha_{n+1} = \left(\frac{2v_{n+1}^3}{v_n} + 1 \right) \alpha_n \tag{17.105h}$$

$$\beta_{n+1} = \left(\frac{2v_{n+1}^3}{v_n} + 1 \right) \beta_n + \left(6w_{n+1} v_n - 2v_{n+1} w_n \right) \frac{v_{n+1}^2 \alpha_n}{v_n^2} \tag{17.105i}$$

$$\pi_n = \frac{8 \cdot 2^{1/8}}{\alpha_n \, \beta_n} \quad \to \pi \tag{17.105j}$$

A second order algorithm from [27]:

$$\alpha_0 = 1/3 \tag{17.106a}$$

$$m_0 = 2 \tag{17.106b}$$

$$m_{n+1} = \frac{4}{1 + \sqrt{(4 - m_n)(2 + m_n)}} \tag{17.106c}$$

$$\alpha_{n+1} = m_n \, \alpha_n + \frac{2^n}{3} (1 - m_n) \quad \to \frac{1}{\pi} \tag{17.106d}$$

An implicit second order algorithm from [27]:

$$\alpha_0 = 1/3 \tag{17.107a}$$

$$s_1 = 1/3 \tag{17.107b}$$

$$(s_n)^2 + (s_n^*)^2 = 1 \tag{17.107c}$$

$$(1 + 3\, s_{n+1})(1 + 3\, s_n^*) = 4 \tag{17.107d}$$

$$\alpha_{n+1} = (1 + 3\, s_{n+1}) \, \alpha_n - 2^n \, s_{n+1} \quad \to \frac{1}{\pi} \tag{17.107e}$$

It is trivial turn this algorithm into an explicit form as with the next algorithm. However, there exist iterations that cannot be turned into explicit forms.

An implicit fourth order algorithm from [27]:

$$\alpha_0 = 1/3 \tag{17.108a}$$

$$s_1 = \sqrt{2} - 1 \tag{17.108b}$$

$$(s_n)^4 + (s_n^*)^4 = 1 \tag{17.108c}$$

$$(1 + 3\, s_{n+1})(1 + 3\, s_n^*) = 2 \tag{17.108d}$$

$$\alpha_{n+1} = (1 + s_{n+1})^4 \alpha_n + \frac{4^{n+1}}{3} \left[1 - (1 + s_{n+1})^4 \right] \quad \to \frac{1}{\pi} \tag{17.108e}$$

Combining two steps of the fourth order iteration leads to an algorithm of order 16. The following form

is given by Frank Garvan:

$$\alpha_0 = 1/3 \tag{17.109a}$$
$$s_1 = \sqrt{2} - 1 \tag{17.109b}$$
$$s_n^* = \left(1 - s_n^4\right)^{1/4} \tag{17.109c}$$
$$x_n = 1/\left(1 + s_n^*\right)^4 \tag{17.109d}$$
$$y_n = x_n \left(1 + s_n\right)^4 \tag{17.109e}$$
$$\alpha_n = 16\, y_n\, \alpha_{n-1} + \frac{4^{2n-1}}{3}\left[1 - 12\, x_n - 4\, y_n\right] \quad \to \frac{1}{\pi} \tag{17.109f}$$
$$t_n = 1 + s_n^* \tag{17.109g}$$
$$u_n = \left[8\, s_n^* \left(1 + {s_n^*}^2\right)\right]^{1/4} \tag{17.109h}$$
$$s_{n+1} = \frac{\left(1 - s_n^*\right)^4}{(t+u)^2\left(t^2+u^2\right)} \tag{17.109i}$$

The definitive source for iterations to compute π and the underlying mathematics is [PiAGM].

Chapter 18

Computing the elementary functions with limited resources

This chapter presents two types of algorithms for computations with limited resources, the shift-and-add and the CORDIC algorithms. The algorithms allow the computations of the elementary functions as the logarithm, exponential function, sine, cosine and their inverses with only shifts, adds, comparisons and table lookups. Algorithms of this type are usually used for pocket calculators.

18.1 Shift-and-add algorithms for $\log_b(x)$ and b^x

In this section so-called *shift-and-add algorithms* for the computation of $\log_b(x)$ and b^x are presented. These algorithms use only additions, multiplications by a power of two ('shifts') and comparisons. Precomputed lookup table with as many entries as the desired accuracy in bits is required. The algorithms are especially useful with limited hardware capabilities.

The implementations given in this section use floating point numbers. They can be rewritten to scaled integer arithmetic without difficultly.

18.1.1 Computing the base-b logarithm

We need a table that contains the values $A_n = \log_b\left(1 + \frac{1}{2^n}\right)$ where $n > 0$:

```
double *shiftadd_ltab;
ulong ltab_n;

void
make_shiftadd_ltab(double b)
{
    double l1b = 1.0 / log(b);
    double s = 1.0;
    for (ulong k=0; k<ltab_n; ++k)
    {
        shiftadd_ltab[k] = log(1.0+s) * l1b;  // == log_b(1+1/2^k)
        s *= 0.5;
    }
}
```

The algorithm takes as input the argument $x \geq 1$ and the number of iterations N and computes $\log_b(x)$. It proceeds as follows:

1. Initialize: Set $t_0 = 0$, $e_0 = 1$.
2. Compute $u_n = e_n \cdot (1 + 2^{-n})$. If $u_n \leq x$ the set $d_n = 1$, else set $d_n = 0$.

3. If $d_n \neq 0$ then set $t_{n+1} = t_n + A_n$ and $e_{n+1} = u_n$ and repeat the last step. Else set $t_{n+1} = t_n$ and $e_{n+1} = e_n$.

4. Increment n. If $n = N$ return t_n, else goto step 2.

A C++ implementation is given in [FXT: demo arith/shiftadd-log-demo.cc]:

```
double
shiftadd_log(double x, ulong n)
{
    if ( n>=ltab_n )  n = ltab_n;
    double t = 0.0;
    double e = 1.0;
    double v = 1.0;
    for (ulong k=1; k<n; ++k)
    {
        v *= 0.5;

        double u;
        bool d;
        while ( 1 )
        {
            u = e + e * v;
            d = ( u<=x );
            // [PRINT]
            if ( d==false )  break;
            t += shiftadd_ltab[k];
            e = u;
        }
    }
    return  t;
}
```

The values for first steps of the computation for the argument $x_0 = \sqrt{2}$ are given in figure 18.1.

n	u_n	t_n	e_n	A_n
init	-	0.00000000	+1.00000000	+1.00000000
1	1.50000000	0.00000000	+1.00000000	+0.58496250
2	1.25000000	0.00000000	+1.00000000	+0.32192809
2	1.56250000	0.32192809	+1.25000000	+0.32192809
3	1.40625000	0.32192809	+1.25000000	+0.16992500
3	1.58203125	0.49185309	+1.40625000	+0.16992500
4	1.49414062	0.49185309	+1.40625000	+0.08746284
5	1.45019531	0.49185309	+1.40625000	+0.04439411
6	1.42822265	0.49185309	+1.40625000	+0.02236781
7	1.41723632	0.49185309	+1.40625000	+0.01122725
8	1.41174316	0.49185309	+1.40625000	+0.00562454
8	1.41725778	0.49747764	+1.41174316	+0.00562454
9	1.41450047	0.49747764	+1.41174316	+0.00281501
10	1.41312181	0.49747764	+1.41174316	+0.00140819
10	1.41450182	0.49888583	+1.41312181	+0.00140819
11	1.41381182	0.49888583	+1.41312181	+0.00070426
11	1.41450215	0.49959010	+1.41381182	+0.00070426
12	1.41415698	0.49959010	+1.41381182	+0.00035217
12	1.41450224	0.49994228	+1.41415698	+0.00035217
13	1.41432961	0.49994228	+1.41415698	+0.00017609
14	1.41424330	0.49994228	+1.41415698	+0.00008805
15	1.41420014	0.49994228	+1.41415698	+0.00004402
15	1.41424330	0.49998631	+1.41420014	+0.00004402
∞	1.41421356	0.50000000	+1.41421356	+0.00000000
	$= x$	$= \log_2(\sqrt{2})$	$= x$	$= 0$

Figure 18.1: Numerical values occurring in the shift-and-add computation of $\log_2(\sqrt{2}) = 1/2$. The values are printed after initialization and at the comment `[PRINT]` in the code. The computation of $\log_{1/2}(\sqrt{2}) = -1/2$ corresponds to the same values but opposite signs for all entries A_n and y_n.

The algorithm has been adapted from [Muller] (chapter 5) where the correction is made only once for

n	u_n	t_n	e_n	A_n
init	-	0.00000000	+1.00000000	+1.00000000
1	1.50000000	0.00000000	+1.00000000	+0.58496250
1	2.25000000	0.58496250	+1.50000000	+0.58496250
1	3.37500000	1.16992500	+2.25000000	+0.58496250
1	5.06250000	1.75488750	+3.37500000	+0.58496250
1	7.59375000	2.33985000	+5.06250000	+0.58496250
1	11.3906250	2.92481250	+7.59375000	+0.58496250
2	9.49218750	2.92481250	+7.59375000	+0.32192809
3	8.54296875	2.92481250	+7.59375000	+0.16992500
4	8.06835937	2.92481250	+7.59375000	+0.08746284
5	7.83105468	2.92481250	+7.59375000	+0.04439411
5	8.07577514	2.96920662	+7.83105468	+0.04439411
∞	8.00000000	2.99999999	+8.00000000	+0.00000000
	$= x$	$= \log_2(8)$	$= x$	$= 0$

Figure 18.2: Values occurring in the first few steps of a shift-and-add computation of $\log_2(8) = 3$.

each value A_n limiting the range of convergence to $x < X$ where

$$\begin{aligned} X &= \prod_{n=0}^{\infty} \left(1 + \frac{1}{2^n}\right) \\ &= 4.768462058062743448299798577356794477543\ldots \end{aligned} \tag{18.1}$$

As given, the algorithm converges for any $x > 0$, $x \neq 1$. A numerical example for the argument $x = 8$ is given in figure 18.2. The basis b must satisfy $b > 0$ and $b \neq 1$.

18.1.2 Computing b^x

We can use the same precomputed table as with the computation of $\log_b(x)$.

The algorithm takes as input the argument x and the number of iterations N and computes b^x for $b > 1$, $x \in \mathbb{R}$. It proceeds as follows:

1. Initialize: Set $t_0 = 0$, $e_0 = 1$.
2. Compute $u_n = t_n + A_n$. If $u_n \leq x$ the set $d_n = 1$, else set $d_n = 0$.
3. If $d_n \neq 0$ then set $t_{n+1} = u_n$ and $e_{n+1} = e_n \cdot (1 + 2^{-n})$ and repeat the last step. Else set $t_{n+1} = t_n$ and $e_{n+1} = e_n$.
4. Increment n. If $n = N$ return e_n, else goto step 2.

A C++ implementation is given in [FXT: demo arith/shiftadd-exp-demo.cc]:

```
double
shiftadd_exp(double x, ulong n)
{
    if ( n>=ltab_n )  n = ltab_n;
    double t = 0.0;
    double e = 1.0;
    double v = 1.0;
    for (ulong k=1; k<n; ++k)
    {
        v *= 0.5;

        double u;
        bool d;
        while ( 1 )
        {
            u = t + shiftadd_ltab[k];
            d = ( u<=x );
            // [PRINT]
            if ( d==false )  break;
```

```
            t = u;
            e += e * v;
        }
    }
    return  e;
}
```

n	u_n	t_n	e_n	A_n
init	0.00000000	0.00000000	+1.00000000	+0.00000000
1	0.58496250	0.00000000	+1.00000000	+0.58496250
2	0.32192809	0.00000000	+1.00000000	+0.32192809
2	0.64385618	0.32192809	+1.25000000	+0.32192809
3	0.49185309	0.32192809	+1.25000000	+0.16992500
3	0.66177809	0.49185309	+1.40625000	+0.16992500
4	0.57931593	0.49185309	+1.40625000	+0.08746284
5	0.53624721	0.49185309	+1.40625000	+0.04439411
6	0.51422090	0.49185309	+1.40625000	+0.02236781
7	0.50308035	0.49185309	+1.40625000	+0.01122725
8	0.49747764	0.49185309	+1.40625000	+0.00562454
8	0.50310219	0.49747764	+1.41174316	+0.00562454
9	0.50029266	0.49747764	+1.41174316	+0.00281501
10	0.49888583	0.49747764	+1.41174316	+0.00140819
10	0.50029403	0.49888583	+1.41312181	+0.00140819
11	0.49959010	0.49888583	+1.41312181	+0.00070426
11	0.50029437	0.49959010	+1.41381182	+0.00070426
12	0.49994228	0.49959010	+1.41381182	+0.00035217
12	0.50029446	0.49994228	+1.41415698	+0.00035217
13	0.50011838	0.49994228	+1.41415698	+0.00017609
14	0.50003033	0.49994228	+1.41415698	+0.00008805
15	0.49998631	0.49994228	+1.41415698	+0.00004402
15	0.50003034	0.49998631	+1.41420014	+0.00004402
∞	0.50000000	0.50000000	+1.41421356	+0.00000000
	$= x$	$= x$	$= 2^{1/2}$	$= 0$

Figure 18.3: Numerical values occurring in the shift-and-add computation of $b^x = 2^{1/2} = \sqrt{2}$. The values are printed after initialization and at the comment `[PRINT]` in the code.

18.1.3 An alternative algorithm for the logarithm

A slightly different method for the computation of the base-b logarithm ($b > 0$, $b \neq 1$) is given in [DEK1].

Here the table used has to contain the values $A_n = \log_b\left(\frac{2^n}{2^n-1}\right)$ where $n > 0$:

```
double *briggs_ltab;
ulong  ltab_len;

void
make_briggs_ltab(ulong na, double b)
{
    double l1b = 1.0 / log(b);
    double s = 2.0; // == 2^k
    briggs_ltab[0] = -1.0; // unsused
    for (ulong k=1; k<na; ++k)
    {
        briggs_ltab[k] = log(s/(s-1.0)) * l1b;
        s *= 2.0;
    }
}
```

The algorithm terminates when a given precision (`eps`) is reached:

```
double
briggs_log(double x, double eps)
{
```

```
    double y = 0;
    double z = x * 0.5;
    ulong k = 1;
    double v = 0.5;  // v == 2^(-k)
    while ( fabs(x-1.0)>=eps )
    {
        while ( fabs(x-z)<1.0 )
        {
            z *= 0.5;
            ++k;  v *= 0.5;
            if ( k >= ltab_len )  goto done;  // no more table entries
            // [PRINT1]
        }
        x -= z;
        y += briggs_ltab[k];
        z = x * v;
        // invariant:  y_k + log_b(x_k) == log_b(x_0)
        // [PRINT2]
    }
 done:
    return  y;
}
```

The code is given in [FXT: demo arith/briggs-log-demo.cc]. The values for first steps of the computation for the argument $x_0 = \sqrt{2}$ are given in figure 18.4. The argument x must be greater than or equal to 1. Knuth [DEK1] gives $1 \le x < 2$ but the restriction to values smaller than 2 does not seem to be necessary.

n	x_n	y_n	z_n	A_n
init	1.41421356	0.00000000	+0.70710678	+0.00000000
2	1.41421356	0.00000000	+0.35355339	+0.41503749
2	1.06066017	0.41503749	+0.26516504	+0.41503749
3	1.06066017	0.41503749	+0.13258252	+0.19264507
4	1.06066017	0.41503749	+0.06629126	+0.09310940
5	1.06066017	0.41503749	+0.03314563	+0.04580368
5	1.02751454	0.46084118	+0.03210982	+0.04580368
6	1.02751454	0.46084118	+0.01605491	+0.02272007
6	1.01145962	0.48356126	+0.01580405	+0.02272007
7	1.01145962	0.48356126	+0.00790202	+0.01131531
7	1.00355759	0.49487657	+0.00784029	+0.01131531
8	1.00355759	0.49487657	+0.00392014	+0.00564656
9	1.00355759	0.49487657	+0.00196007	+0.00282051
9	1.00159752	0.49769709	+0.00195624	+0.00282051
10	1.00159752	0.49769709	+0.00097812	+0.00140957
10	1.00061940	0.49910666	+0.00097716	+0.00140957
11	1.00061940	0.49910666	+0.00048858	+0.00070461
11	1.00013081	0.49981128	+0.00048834	+0.00070461
12	1.00013081	0.49981128	+0.00024417	+0.00035226
13	1.00013081	0.49981128	+0.00012208	+0.00017612
13	1.00000873	0.49998740	+0.00012207	+0.00017612
∞	1.00000000	0.50000000	+0.00000000	+0.00000000
	$= 1$	$= \log_2(\sqrt{2})$	$= 0$	$= 0$

Figure 18.4: Numerical values occurring in the computation of $\log_2(\sqrt{2}) = 1/2$. The value of n is incremented in the inner loop (comment `[PRINT1]` in the code, the value of z changes). The values of x and y change just before the location of the comment `[PRINT2]`, corresponding to consecutive rows with same value of n. The computation of $\log_{1/2}(\sqrt{2}) = -1/2$ corresponds to the same values but opposite signs for all entries A_n and y_n.

18.2 CORDIC algorithms

The so-called CORDIC algorithms can be used for the computation of functions like sine, cosine, exp and log. The acronym CORDIC stands for **Co**ordinate **R**otation **Di**gital **C**omputer.

Similar to the shift-and-add algorithms (section 18.1) only multiplications by powers of two (shifts), additions, subtractions and comparisons are used. Again, a precomputed lookup table with as many entries as the desired accuracy in bits is required.

Some early floating point units (FPUs) used CORDIC algorithms and your pocket calculator surely does.

18.2.1 The circular case: sine and cosine

We start with a CORDIC routine for the computation of the sine and cosine. The lookup table has to contain the values $\arctan(2^{-n})$ for $n = 0, 1, 2, 3, \ldots$, these shall be stored in the array `cordic_ctab[]`. An implementation of the function is given in [FXT: demo arith/cordic-circ-demo.cc]:

```
void
cordic_circ(double theta, double &s, double &c, ulong n)
{
    double x = cordic_1K;
    double y = 0;
    double z = theta;
    double v = 1.0;
    for (ulong k=0; k<n; ++k)
    {
        double d = ( z>=0 ? +1 : -1 );
        double tx = x - d * v * y;
        double ty = y + d * v * x;
        double tz = z - d * cordic_ctab[k];
        x = tx;  y = ty;  z = tz;
        v *= 0.5;
    }
    c = x;
    s = y;
}
```

For the sake of clarity floating point types are used. All operations can easily be converted to integer arithmetic. The multiplications by `d` are sign changes and should be replaced by an `if`-construct. The multiplications by `v` are shifts.

n	x_n	y_n	z_n	$-d \cdot A_n$
init	0.60725293	0.00000000	+1.04719755	+0.00000000
0	0.60725293	0.60725293	+0.26179938	-0.78539816
1	0.30362646	0.91087940	-0.20184822	-0.46364760
2	0.53134631	0.83497278	+0.04313044	+0.24497866
3	0.42697471	0.90139107	-0.08122455	-0.12435499
4	0.48331166	0.87470515	-0.01880574	+0.06241880
5	0.51064619	0.85960166	+0.01243409	+0.03123983
6	0.49721492	0.86758051	-0.00318963	-0.01562372
7	0.50399289	0.86369602	+0.00462270	+0.00781234
8	0.50061908	0.86566474	+0.00071647	-0.00390623
9	0.49892833	0.86664251	-0.00123664	-0.00195312
10	0.49977466	0.86615528	-0.00026008	+0.00097656
11	0.50019758	0.86591124	+0.00022819	+0.00048828
12	0.49998618	0.86603336	-0.00001594	-0.00024414
13	0.50009190	0.86597233	+0.00010612	+0.00012207
14	0.50003904	0.86600285	+0.00004508	-0.00006103
15	0.50001261	0.86601811	+0.00001457	-0.00003051
∞	0.50000000	0.86602540	+0.00000000	+0.00000000
	$= \cos(\pi/3)$	$= \sin(\pi/3)$	$= 0$	$= 0$

Figure 18.5: Numerical values occurring in the CORDIC computation of $\cos(\pi/3)$ and $\sin(\pi/3)$.

The values for first 16 steps of the computation for the argument $z_0 = \theta = \pi/3 = 1.04719755\ldots$ are given in figure 18.5 on the facing page. While z gets closer to zero (however, the magnitude of z does not necessarily decrease with every step) the values of x and y approach $\sin(\pi/3) = 1/2$ and $\cos(\pi/3) = \sqrt{3}/2 = 0.86602540\ldots$, respectively.

More formally, one initializes

$$\begin{aligned} x_0 &= 1/K = 0.607252935008881\ldots && (18.2a)\\ y_0 &= 0 && (18.2b)\\ z_0 &= \theta && (18.2c)\end{aligned}$$

and iterates (starting with $n = 0$)

$$\begin{aligned} A_n &= \arctan\left(2^{-n}\right) \quad \text{(precomputed)} && (18.2d)\\ v_n &= 2^{-n} && (18.2e)\\ d_n &= \operatorname{sign}(z_n) && (18.2f)\\ x_{n+1} &= x_n - d_n\, v_n\, y_n \quad \to \cos(\theta) && (18.2g)\\ y_{n+1} &= y_n + d_n\, v_n\, x_n \quad \to \sin(\theta) && (18.2h)\\ z_{n+1} &= z_n - d_n\, A_n \quad \to 0 && (18.2i)\end{aligned}$$

The scaling constant K is

$$\begin{aligned} K &= \prod_{k=0}^{\infty} \sqrt{1 + 2^{-2k}} && (18.3a)\\ K &= 1.64676025812106564836605122228229843565237672570102740 9\ldots && (18.3b)\\ \frac{1}{K} &= 0.60725293500888125616944675250492826311239085215008977 24\ldots && (18.3c)\end{aligned}$$

The algorithm converges if $-r \le z_0 \le r$ where

$$\begin{aligned} r &= \sum_{k=0}^{\infty} \arctan(2^{-k}) && (18.4a)\\ r &= 1.74328662047234000350433765613641628581383118542820652 3\ldots && (18.4b)\\ r &> \frac{\pi}{2} = 1.57079632\ldots && (18.4c)\end{aligned}$$

With arguments x_0, y_0, z_0 one has

$$\begin{aligned} x &\to K\left(x_0\, \cos(z_0) - y_0\, \sin(z_0)\right) && (18.5a)\\ y &\to K\left(y_0\, \cos(z_0) + x_0\, \sin(z_0)\right) && (18.5b)\\ z &\to 0 && (18.5c)\end{aligned}$$

which, for $x_0 = 1/K, y_0 = 0, z_0 = \theta$ specializes to the computation as above.

A nice feature of the algorithm is that it also works backwards: Initialize as above and use the same iteration with the slight modification that $d_n := -\operatorname{sign}(y_n)$, then

$$\begin{aligned} x &\to K\sqrt{x_0^2 + y_0^2} && (18.6a)\\ y &\to 0 && (18.6b)\\ z &\to z_0 - \arctan\left(\frac{y_0}{x_0}\right) && (18.6c)\end{aligned}$$

The algorithm can be derived by writing

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} +\cos(d_n\, A_n) & -\sin(d_n\, A_n) \\ +\sin(d_n\, A_n) & +\cos(d_n\, A_n) \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} \tag{18.7}$$

and noting that (using $d_n = \pm 1$, so $\cos(d_n\, A_n) = \cos(A_n)$ and $\sin(d_n\, A_n) = d_n\, \sin(A_n)$)

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \cos(A_n) \begin{bmatrix} +1 & -d_n\, v_n \\ +d_n\, v_n & +1 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} \tag{18.8}$$

where $v_n = 2^{-n}$. The CORDIC algorithm postpones the multiplications by $\cos(A_n)$. One has

$$\cos(A_n) = \cos\left(\arctan(2^{-n})\right) = \frac{1}{\sqrt{1 + 2^{-2n}}} \tag{18.9}$$

Thereby

$$K = 1/\prod_{k=0}^{\infty} \cos(A_n) = \prod_{k=0}^{\infty} \sqrt{1 + 2^{-2n}} \tag{18.10}$$

18.2.2 The linear case: multiplication and division

A slight variation gives a base-2 multiply-add algorithm:

$$A_n = 2^{-n} \tag{18.11a}$$
$$v_n = 2^{-n} \tag{18.11b}$$
$$d_n = \text{sign}(z_n) \tag{18.11c}$$
$$x_{n+1} = x_n \tag{18.11d}$$
$$y_{n+1} = y_n + d_n\, v_n\, x_n \tag{18.11e}$$
$$z_{n+1} = z_n - d_n\, A_n \tag{18.11f}$$

then

$$x \to x_0 \tag{18.12a}$$
$$y \to y_0 + x_0\, z_0 \tag{18.12b}$$
$$z \to 0 \tag{18.12c}$$

Going backwards ($d_n := -\,\text{sign}\,(y_n)$) gives an algorithm for division:

$$x \to x_0 \tag{18.13a}$$
$$y \to 0 \tag{18.13b}$$
$$z \to z_0 - \frac{y_0}{x_0} \tag{18.13c}$$

18.2.3 The hyperbolic case: *sinh* and *cosh*

The versions presented so far can be unified as

$$v_n = 2^{-n} \tag{18.14a}$$
$$x_{n+1} = x_n - m\, d_n\, v_n\, y_n \tag{18.14b}$$
$$y_{n+1} = y_n + d_n\, v_n\, x_n \tag{18.14c}$$
$$z_{n+1} = z_n - d_n\, A_n \tag{18.14d}$$

where the linear case corresponds to $m = 0$ and $A_n = 2^{-n}$, the circular case to $m = 1$ and $A_n = \arctan(2^{-n})$. The forward direction ('rotation mode') is obtained by setting $d_n = \operatorname{sign}(z_n)$, the backward direction ('vectoring mode') by setting $d_n = -\operatorname{sign}(y_n)$.

Setting $m = -1$ gives a CORDIC algorithm that computes the hyperbolic sine and cosine or their inverses. The lookup table has to contain the values $\operatorname{arctanh}(2^{-n})$ for $n = 1, 2, 3, \ldots$, stored in the array `cordic_htab[]`. The algorithm needs a modification in order to converge: the iteration starts with index one and some steps have to be executed twice. The sequence of the indices that need to be processed twice is $4, 13, 40, 121, \ldots$ ($i_0 = 4$, $i_{n+1} = 3\,i_n + 1$).

A sample implementation is given in [FXT: demo arith/cordic-hyp-demo.cc]:

```
void
cordic_hyp(double theta, double &s, double &c, ulong n)
{
    double x = cordic_1Kp;
    double y = 0;
    double z = theta;
    double v = 1.0;
    ulong i = 4;
    for (ulong k=1; k<n; ++k)
    {
        v *= 0.5;
    again:
        double d = ( z>=0 ? +1 : -1 );
        double tx = x + d * v * y;
        double ty = y + d * v * x;
        double tz = z - d * cordic_htab[k];
        x = tx;  y = ty;  z = tz;
        if ( k==i )  { i=3*i+1; goto again; }
    }
    c = x;
    s = y;
}
```

The values for first steps of the computation for the argument $\theta = z_1 = 1.0$ are given in figure 18.6.

n	x_n	y_n	z_n	A_n
init	1.20749706	0.00000000	+1.00000000	+0.00000000
1	1.20749706	0.60374853	+0.45069385	-0.54930614
2	1.35843420	0.90562280	+0.19528104	-0.25541281
3	1.47163705	1.07542707	+0.06962382	-0.12565721
4	1.53885124	1.16740439	+0.00704225	-0.06258157
+4	1.61181401	1.26358259	-0.05553931	-0.06258157
5	1.57232706	1.21321340	-0.02427913	+0.03126017
6	1.55337060	1.18864579	-0.00865286	+0.01562627
7	1.54408430	1.17651008	-0.00084020	+0.00781265
8	1.53948856	1.17047850	+0.00306606	+0.00390626
9	1.54177465	1.17348532	+0.00111293	-0.00195312
10	1.54292063	1.17499096	+0.00013637	-0.00097656
11	1.54349436	1.17574434	-0.00035190	-0.00048828
12	1.54320731	1.17536751	-0.00010776	+0.00024414
13	1.54306383	1.17517913	+0.00001430	+0.00012207
13	1.54320729	1.17536749	-0.00010776	-0.00012207
14	1.54313555	1.17527330	-0.00004673	+0.00006103
15	1.54309968	1.17522621	-0.00001621	+0.00003051
∞	1.54308063	1.17520119	+0.00000000	+0.00000000
	$= \cosh(1)$	$= \sinh(1)$	$= 0$	$= 0$

Figure 18.6: Numerical values occurring in the CORDIC computation of cosh(1) and sinh(1). Note that step 4 is executed twice.

The scaling constant corresponding to K is K', one has

$$K' = \prod_{k=1}^{\infty} \sqrt{1-2^{-2k}} \cdot \prod_{k=0}^{\infty} \sqrt{1-2^{-2i_k}} \tag{18.15a}$$

$$K' = 0.828159360960215627076198327759175146869453837690842529 1\ldots \tag{18.15b}$$

$$\frac{1}{K'} = 1.207497067763072128877721011310915836812783221769813422\ldots \tag{18.15c}$$

The duplicated indices appear twice in the product.

The algorithm can be used for the computation of the exponential function using $\exp(x) = \sinh(x) + \cosh(x)$.

The algorithm converges if $-r' \leq z_1 \leq r'$ where

$$r' = \sum_{k=1}^{\infty} \operatorname{arctanh}(2^{-k}) + \sum_{k=0}^{\infty} \operatorname{arctanh}(2^{-i_k}) \tag{18.16a}$$

$$r' = 1.118173015526503803610627556783092451806572942929536106\ldots \tag{18.16b}$$

With arguments x_1, y_1, z_1 one has

$$x \rightarrow K' \left(x_1 \cosh(z_1) + y_1 \sinh(z_1)\right) \tag{18.17a}$$

$$y \rightarrow K' \left(y_1 \cosh(z_1) + x_1 \sinh(z_1)\right) \tag{18.17b}$$

$$z \rightarrow 0 \tag{18.17c}$$

which, for $x_1 = 1/K', y_1 = 0, z_1 = \theta$ specializes to the computation as above.

The backward version ($d_n := -\operatorname{sign}(y_n)$) computes

$$x \rightarrow K' \sqrt{x_1^2 - y_1^2} \tag{18.18a}$$

$$y \rightarrow 0 \tag{18.18b}$$

$$z \rightarrow z_1 - \operatorname{arctanh}\left(\frac{y_1}{x_1}\right) \tag{18.18c}$$

For the computation of the natural logarithm use $\log(w) = 2 \operatorname{arctanh} \frac{w-1}{w+1}$. That is, start with $x_1 = w+1$ and $y_1 = w-1$, then $z \rightarrow \frac{1}{2}\log(w)$.

The computation of the square root $\sqrt{w}$ can be obtained by starting with $x_1 = w+1/4$ and $y_1 = w-1/4$ then $z \rightarrow K'\sqrt{w}$.

The reader is referred to [4], [58] and chapter 6 of [Muller] for further studies.

Chapter 19

Recurrences and Chebyshev polynomials

This chapter presents algorithms and material concerning recurrences.

Firstly, several algorithms for recurrences, mostly for the case of constant coefficients, are given. Secondly, the Chebyshev polynomials are described. These are an important special case of a recurrence.

19.1 Recurrences

A sequence $[a_0, a_1, a_2, \dots]$ so that a *recurrence relation*

$$a_n \quad = \quad \sum_{k=1}^{K} m_k \, a_{n-k} \tag{19.1}$$

with given m_k holds for all a_j is called a K-th order recurrence. The sequence is defined by both the recurrence relation and the first $K-1$ elements.

An example, the second order recurrence relation $a_n = 1\,a_{n-1} + 1\,a_{n-2}$ together with $a_0 = 0$ and $a_1 = 1$ gives the Fibonacci numbers F_n, starting with $a_0 = 2$ and $a_1 = 1$ gives the Lucas numbers L_n:

```
  n : 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14
F(n): 0  1  1  2  3  5  8 13 21 34  55  89 144 233 377
L(n): 2  1  3  4  7 11 18 29 47 76 123 199 322 521 843
```

The *characteristic polynomial* of the recurrence relation 19.1 is given by

$$p(x) \quad = \quad x^K - \sum_{k=1}^{K} m_k \, x^{K-k} \tag{19.2}$$

The definition can be motivated by writing down the recurrence relation for the element with index $n = K$:

$$0 \quad = \quad a_K - \left(\sum_{k=1}^{K} m_k \, a_{K-k} \right) \tag{19.3}$$

19.1.1 Fast computation using matrix powers

For the recurrence defined by the recurrence relation

$$a_n \quad := \quad m_1\, a_{n-1} + m_2\, a_{n-2} \tag{19.4}$$

and the start a_0, a_1 use the relation

$$[a_0,\ a_1] \begin{bmatrix} 0 & m_2 \\ 1 & m_1 \end{bmatrix}^k \quad = \quad [a_k,\ a_{k+1}] \tag{19.5}$$

for the fast computation of an individual element a_k. The algorithm is fast when powering algorithms (see section 14.6) are used.

Note that with two consecutive terms of the recurrence in the resulting vector it is easy two compute the following terms a_{k+1}, a_{k+2}, ... using the original recurrence relation.

The generalization is straightforward. For example, a recurrence $a_n = m_1\, a_{n-1} + m_2\, a_{n-2} + m_3\, a_{n-3}$ corresponds to

$$[a_0,\ a_1,\ a_2] \begin{bmatrix} 0 & 0 & m_3 \\ 1 & 0 & m_2 \\ 0 & 1 & m_1 \end{bmatrix}^k \quad = \quad [a_k,\ a_{k+1},\ a_{k+2}] \tag{19.6}$$

The matrix is the companion matrix of the characteristic polynomial $x^3 - (m_1\, x^2 + m_2\, x^1 + m_3\, x^0)$, see relation 23.18 on page 615. Note that the indexing of the m_k is different here.

Performance

The computations are fast. As an example we give the timing of the computation of a few sequence terms with large indices. The following calculations were carried out with exact arithmetic, the post-multiply with the float 1.0 renders the output readable:

```
? M=[0,1;1,1] \\ Fibonacci sequence
? #
   timer = 1 (on)
? ([0,1]*M^10000)[1]*1.0
  time = 1 ms.
  3.364476487643 E2089
? ([0,1]*M^100000)[1]*1.0
  time = 10 ms.
  2.597406934722 E20898
? ([0,1]*M^1000000)[1]*1.0
  time = 458 ms.
  1.953282128707 E208987
```

The powering algorithm can obviously be used also for polynomial recurrences such as for the Chebyshev polynomials $T_n(x)$:

```
? M=[0,-1;1,2*x]
  [0  -1]
  [1 2*x]
? for(n=0,5,print(n,": ",([1,x]*M^n)[1]))
  0: 1
  1: x
  2: 2*x^2 - 1
  3: 4*x^3 - 3*x
  4: 8*x^4 - 8*x^2 + 1
  5: 16*x^5 - 20*x^3 + 5*x
? p=([1,x]*M^1000)[1];
  time = 1,027 ms.
? poldegree(p)
  1000
? log(polcoeff(p,poldegree(p)))/log(10)
  300.728965668317  \\ The coefficient of x^1000 is a 301-digit number
```

With modular arithmetic the quantities remain bounded and the computations can be carried out for extreme large values of n. We use the modulus $m = 2^{1279} - 1$ and compute the $n = (m+1)/4$ element of the sequence $2, 4, 14, 52, \ldots$ where $a_n = 4\, a_{n-1} - a_{n-2}$:

```
? m=2^1279-1;  \\ a 1279-bit number
? log(m)/log(10)
  385.0173 \\ 306 decimal digits
? M=Mod([0,-1;1,4],m);  \\ all entries modulo m
?  component( ([2,4]*M^((m+1)/4))[1], 2)
  time = 118 ms.
  0
```

The result is zero which proves that m is prime, see section 21.10.3. Here is a one-liner that prints all exponents $e < 1000$ of Mersenne primes:

```
? forprime(e=3,1000,m=2^e-1;M=Mod([0,-1;1,4],m);if(0==(([2,4]*M^((m+1)/4))[1]),print1(" ", e)))
  3 5 7 13 17 19 31 61 89 107 127 521 607
```

The computation takes a few seconds only.

19.1.2 Faster computation using polynomial arithmetic

The matrix power algorithm for computing the k-th element of a n-th order recursion involves proportional $\log_k$ multiplications of $n \times n$ matrices. As matrix multiplication (with the straightforward algorithm) is proportional n^3 the algorithm is not optimal for recursions of high order. Note that the matrix entries grow exponentially, so the asymptotics as given is valid only for computations with bounded values such as with modular arithmetic. We will see that the involved work can be brought down from $\log k \cdot n^3$ to $\log k \cdot n^2$ and even to $\log k \cdot n \cdot \log n$.

The characteristic polynomial for the recursion $a_n := 3\,a_{n-1} + 1\,a_{n-2} + 2\,a_{n-3}$ is

$$p(x) \quad = \quad x^3 - 3\,x^2 - 1\,x - 2 \tag{19.7}$$

We list the first few powers of the companion matrix M of $p(x)$:

$$M^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad M^1 = \begin{bmatrix} 0 & 0 & 2 \\ 1 & 0 & 1 \\ 0 & 1 & 3 \end{bmatrix} \qquad M^2 = \begin{bmatrix} 0 & 2 & 6 \\ 0 & 1 & 5 \\ 1 & 3 & 10 \end{bmatrix} \qquad M^3 = \begin{bmatrix} 2 & 6 & 20 \\ 1 & 5 & 16 \\ 3 & 10 & 35 \end{bmatrix} \tag{19.8}$$

Note that each power is a left shifted version of its predecessor, only the rightmost colum is 'new'. Now compare the columns of the matrix powers to the first few values x^k modulo $p(x)$:

$$x^0 \bmod p(x) \quad = \quad 0\,x^2 + 0\,x + 1 \tag{19.9a}$$
$$x^1 \bmod p(x) \quad = \quad 0\,x^2 + 1\,x + 0 \tag{19.9b}$$
$$x^2 \bmod p(x) \quad = \quad 1\,x^2 + 0\,x + 0 \tag{19.9c}$$
$$x^3 \bmod p(x) \quad = \quad 3\,x^2 + 1\,x + 2 \tag{19.9d}$$
$$x^4 \bmod p(x) \quad = \quad 10\,x^2 + 5\,x + 6 \tag{19.9e}$$
$$x^5 \bmod p(x) \quad = \quad 35\,x^2 + 16\,x + 20 \tag{19.9f}$$

Observe that $x^k \bmod p(x)$ corresponds to the leftmost column of M^k.

We now turn the observation into an efficient algorithm. The main routines in this section take as arguments a vector v of initial values, a vector m of recursion coefficients and an index k. The vector $r = [a_k, a_{k+1}, \ldots, a_{k+n}]$ is returned. We compute the leftmost column of M^k as $z := x^k \bmod p(x)$ and compute a_k as the scalar product of z (as a vector) and v. Our main routine is:

```
frec(v, m, k)=
{
    local(n, pc, pv, pp, px, r, t);
    n = length(m);
    if ( k<=n, return( recstep(v, m, k) ) );  \\ small indices by definition
    pc = vec2charpol(m);
    pp = Mod( x, pc );
    px = pp^(k);
    r = vector(n);
    for (i=1, n,
```

```
        t = component(px,2);
        r[i] = sum(j=1,n, v[j]*polcoeff(t,j-1,x));
        px *= pp;
    );
    return( r );
}
```

If only the value a_k is of interest, skip the computations in the final `for` loop for the values `i`> 1.

For small indices k the result is computed directly by definition, using the following auxiliary routine:

```
recstep(v, m, k)=
{ /* update v by k steps according to the recursion coefficients in m */
    local(n,r);
    if ( k<=0, return(v) );  \\ negative k is forbidden
    n = length(m);
    r = vector(n);
    for (i=1, k,
        for (j=1, n-1, r[j]=v[j+1] ); \\ shift left
        r[n] = sum(j=1,n, m[n+1-j]*v[j]);  \\ new element (convolution)
        v = r;
    );
    return( r );
}
```

The auxiliary routine used to compute the characteristic polynomial corresponding to the vector m is:

```
vec2charpol(m)=
{ /* return characteristic polynomial for the recursion coefficients in m */
    local(d,p);
    d = length(m);
    p = x^d - Pol(m,x);
    return( p );
}
```

As proportional $\log k$ modular polynomial multiplications are involved the asymptotics is $\log k \cdot n^2$ when usual polynomial multiplication is used and $\log k \cdot n \cdot \log n$ if an FFT scheme is applied.

The matrix power algorithm, restated for the argument structure defined above, can be implemented as:

```
mrec(v, m, k)=
{
    local(p,M);
    p = vec2charpol(m);
    M = matcompanion(p);
    M = M^k;
    return ( v * M );
}
```

All main routines can be used with symbolic values:

```
? frec([a0,a1],[m1,m2],3)
  [m2*m1*a0 + (m1^2 + m2)*a1, (m2*m1^2 + m2^2)*a0 + (m1^3 + 2*m2*m1)*a1]
? mrec([a0,a1],[m1,m2],3)
  [m2*m1*a0 + (m1^2 + m2)*a1, (m2*m1^2 + m2^2)*a0 + (m1^3 + 2*m2*m1)*a1]
? recstep([a0,a1],[m1,m2],3)
  [m2*m1*a0 + (m1^2 + m2)*a1, (m2*m1^2 + m2^2)*a0 + (m1^3 + 2*m2*m1)*a1]
```

We check the performance (suppressing output):

```
? k=10^5;
? recstep([0,1],[1,1],k);
  time = 2,811 ms.  \\ time linear in k
? mrec([0,1],[1,1],k);
  time = 10 ms. \\ time linear in log(k)
? frec([0,1],[1,1],k);
  time = 4 ms. \\ time linear in log(k)
```

The relative performance of the routine `frec()` and `mrec()` differs more with higher orders n of the recurrence, we use $n = 10$:

```
? n=10; v=vector(n); v[n]=1; m=vector(n,j,1); k=10^5;  \\ tenth order recurrence
? mrec(v,m,k);
  time = 2,813 ms.
? f=frec(v,m,k);
  time =  159 ms.
```

```
? log(f)/log(10.0)
  [30078.67, 30078.97, 30079.27, 30079.58, 30079.88, 30080.18, \
  30080.48, 30080.78, 30081.08, 30081.38] \\ about 30k decimal digits each
```

Surprisingly, we see a performance gain greater than n even though the computations were done integers. Finally, we repeat the computations modulo $p = 2^{521} - 1$ for $k = 10^{30}$:

```
?  n=10; v=vector(n); v[n]=1; m=vector(n,j,1); k=10^30;
?  p=2^521-1; v=Mod(v,p); m=Mod(m,p);
?  mrec(v,m,k);
   time = 312 ms.
?  frec(v,m,k);
   time = 14 ms.
```

That the performance gain with integers is not smaller than with modular arithmetic can be motivated by the fact that the quantities in both algorithms grow with the same rate. Now at each step the performance ratio should approximately equal n. Thereby the algorithms perform with the same ratio.

19.1.3 Inhomogeneous recurrences

The fast algorithms for the computation of recurrences do only work with *homogeneous* recurrences as defined by relation 19.1. A *inhomogeneous* recurrence is defined by a relation

$$a_n \quad = \quad \sum_{k=1}^{K} m_k \, a_{n-k} + P(n) \tag{19.10}$$

where $P(n)$ is a nonzero polynomial in n. We will show how to transform a inhomogeneous recurrence into a homogeneous recurrence of greater order.

Recurrence relations with a constant

In case a constant is to be added in an k-th order relation, one can use a recurrence of order $k + 1$. From the recurrence relation $a_n = m_1 \, a_{n-1} + m_2 \, a_{n-2} + \ldots + m_k \, a_{n-k} + C$ subtract a shifted version $a_{n-1} = m_1 \, a_{n-2} + m_2 \, a_{n-3} + \ldots + m_k \, a_{n-k-1} + C$ to obtain $a_n = (m_1 + 1) \, a_{n-1} + (m_2 - m_1) \, a_{n-2} + \ldots + (m_k - m_{k-1}) \, a_{n-k}$.

An example should make the idea clear: With $a_n = 34 \, a_{n-1} - a_{n-2} + 2$ subtract a shifted version $a_{n-1} = 34 \, a_{n-2} - a_{n-3} + 2$ to obtain $a_n = 35 \, a_{n-1} - 35 \, a_{n-2} + a_{n-3}$. Setting $a_0 = 1$, $a_1 = 36$ we get, using the original relation

```
? n=7;
? ts=vector(n); ts[1]=1; ts[2]=36;
? for(k=3,n,ts[k]=34*ts[k-1]-ts[k-2]+2);
? ts
  [1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
```

and, using the relation without constant,

```
? ts=vector(n); ts[1]=1; ts[2]=36; ts[3]=34*ts[2]-ts[1]+2;
? for(k=4,n,ts[k]=35*ts[k-1]-35*ts[k-2]+ts[k-3]);
? ts
  [1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
```

The general case

If the recurrence is of the form

$$a_n \quad = \quad m_1 \, a_{n-1} + m_2 \, a_{n-2} + \ldots + m_k \, a_{n-k} + P(n) \tag{19.11}$$

where $P(n)$ is a polynomial of degree d in n then a homogeneous recurrence of order $k + d + 1$

$$a_n \quad = \quad M_1 \, a_{n-1} + M_2 \, a_{n-2} + \ldots + M_{k+d+1} \, a_{n-k-d-1} \tag{19.12}$$

can be obtained by repeatedly subtracting a shifted relation.

The following pari/gp routine takes as input a vector of the multipliers m_i ($i = 1, \ldots, k$) and a polynomial of degree d in n. It returns a homogeneous recurrence relation as a vector $[M_1, \ldots, M_{k+d+1}]$:

```
ihom2hom(m, p)=
{
    local(d, M, k);
    if ( p==0, return(m) );
    d = poldegree(p, 'n);
    k = length(m);
    M = vector(k+d+1);
    for (j=1, k, M[j]=m[j]);
    for (s=1, d+1,
        M[1] += 1;  \\ left hand side
        for (j=2, k+s, M[j] -= m[j-1]; );
        m = M;
    );
    return(M);
}
```

In order to verify the output we use a (slow) routine that directly computes the values of an inhomogeneous recurrence:

```
ihom(v, m, k, p)=
{
    local(n, r);
    if ( k<=0, return(v[1]) );
    n = length(m);
    r = vector(n);
    for (i=1, k,
        for (j=1, n-1, r[j]=v[j+1] ); \\ shift left
        r[n] = sum(j=1,n, m[n+1-j]*v[j]);  \\ new element (convolution)
        r[n] += subst(p, 'n, i+n-1);  \\ add inhomogeneous term
        v = r;
    );
    return( r[1] );
}
```

We use the recurrence relation $a_n = 3\,a_{n-1} + 2a_{n-2} + (n^3 - n^2 - 7)$. We compute the homogeneous equivalent (intermediate values of M added):

```
? m=[3,+2];p=n^3-n^2-7;
? M=ihom2hom(m,p)
    [3, 2, 0, 0, 0, 0]
    [4, -1, -2, 0, 0, 0]
    [5, -5, -1, 2, 0, 0]
    [6, -10, 4, 3, -2, 0]
    [7, -16, 14, -1, -5, 2]
  [7, -16, 14, -1, -5, 2]  \\ a_n = 7*a_{n-1} - 16*a_{n-2} + 14*a_{n-3} +- ...
```

We can compute the first few values for the sequence starting with $a_0 = 2$, $a_1 = 5$ by the direct method:

```
? v=[2,5];
? for(k=0,9,print(k,":  ",ihom(v,m,k,p)));
  0:  2
  1:  5
  2:  16
  3:  69
  4:  280
  5:  1071
  6:  3946
  7:  14267
  8:  51134
  9:  182577
```

A vector of start values and the homogeneous equivalent allow the fast computation using the powering algorithms:

```
? V=vector(length(M),j,ihom(v,m,j-1,p))
  [2, 5, 16, 69, 280, 1071]
? for(k=0,9,print(k,":  ",frec(V,M,k)[1]));
 [- same output as with direct computation -]
```

The computation of $a_{10,000}$ now takes less than a second:

```
? z=frec(V,M,10^5)[1];  \\ result computed in 156 ms.
? 1.0*z
  1.72279531330182 E55164
```

```
? z=ihom(v,m,10^5,p);  \\ result computed in 6,768 ms.
```

19.1.4 Recurrence relations for subsequences

Two term recurrences

The recurrence for the subsequence of every k-th element of a two term recurrence $a_n = \alpha\, a_{n-1} + \beta\, a_{n-2}$ can be obtained as follows. Write

$$\begin{aligned}
a_{n+0} &= A_0\, a_n + B_0\, a_{n-0} &&= 2\, a_n \;-\; 1\, a_{n-0} && (19.13a)\\
a_{n+1} &= A_1\, a_n + B_1\, a_{n-1} &&= \alpha\, a_n \;+\; \beta\, a_{n-1} && (19.13b)\\
a_{n+2} &= A_2\, a_n + B_2\, a_{n-2} &&= (\alpha^2 + 2\beta)\, a_n \;-\; \beta^2\, a_{n-2} && (19.13c)\\
a_{n+3} &= A_3\, a_n + B_3\, a_{n-3} &&= (\alpha^3 + 3\alpha\beta)\, a_n \;+\; \beta^3\, a_{n-3} && (19.13d)\\
a_{n+4} &= A_4\, a_n + B_4\, a_{n-4} &&= (\alpha^4 + 4\alpha^2\beta + 2\beta^2)\, a_n \;-\; \beta^4\, a_{n-4} && (19.13e)\\
a_{n+k} &= A_k\, a_n + B_k\, a_{n-k} && && (19.13f)
\end{aligned}$$

We have $a_n = A_k\, a_{n-k} + B_k\, a_{n-2k}$ where $A_0 = 2$, $A_1 = \alpha$ and $A_{k+1} = \alpha\, A_k + \beta\, A_{k-1}$ (and $B_k = -(-\beta)^k$). That is, the first coefficient A_k of the recursion relations for the subsequences can be computed by the original recurrence relation. For efficient computation use

$$[A_k,\ A_{k+1}] \;=\; [2,\ \alpha] \begin{bmatrix} 0 & \beta \\ 1 & \alpha \end{bmatrix}^k \qquad (19.14)$$

A closed form for A_k in terms of Chebyshev polynomials is given in item 14 of [HAKMEM]:

$$A_k \;=\; 2\,(-\beta)^{k/2}\, T_k\left(\alpha/\sqrt{-4\,\beta}\right) \qquad (19.15)$$

A simple example, let F_n and L_n denote the n-th Fibonacci and Lucas number, respectively. Then $\alpha = \beta = 1$ and

$$[A_k,\ A_{k+1}] \;=\; [2,\ 1] \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^k \;=\; [L_k,\ L_{k+1}] \qquad (19.16)$$

That is

$$F_{k\,n+e} \;=\; L_k\, F_{k\,(n-1)+e} - (-1)^k\, F_{k\,(n-2)+e} \qquad (19.17)$$

where $k \in \mathbb{Z}$ and $e \in \mathbb{Z}$. The variable e expresses the shift invariance of the relation.

Recurrences of order n

For the stride-s recurrence relations of order n the following may be the most straightforward algorithm. Let $p(x)$ be the characteristic polynomial of the recurrence and M its companion matrix. Then the characteristic polynomial of M^s corresponds to the recurrence relation of the stride-s subsequence.

```
recsubseq(n, s, m=0)=
{ /* Return vector coefficients of the stride-s subsequence
   * of the n-th order linear recurrence.
   */
    local(p, M, z, r);
    if ( 0==m,
        m = vector(n,j,eval(Str("m" j)));  \\ use symbols m_j
        , /* else */
        n = length(m);  \\ m given
     );
```

```
    p = vec2charpol(m);
    M = matcompanion(p);
    z = x^n-charpoly(M^s);
    r = vector(n,j,polcoeff(z,n-j,x));
    return( r );
}
```

For the second order recurrence we get what we have already seen for $s = 0, \ldots, 4$:

```
? m=[a,b];
? for(s=-2,5,print(s,":  ",recsubseq(0,s,m)););
  -2:  [1/b^2*a^2 + 2/b,   -1/b^2]
  -1:  [-1/b*a,   1/b]
   0:  [2,   -1]
   1:  [a,   b]
   2:  [a^2 + 2*b,   -b^2]
   3:  [a^3 + 3*b*a,   b^3]
   4:  [a^4 + 4*b*a^2 + 2*b^2,   -b^4]
   5:  [a^5 + 5*b*a^3 + 5*b^2*a,   b^5]
```

For the third order recurrence we get:

```
? m=[a,b,c];
? for(s=-2,5,print(s,":  ",recsubseq(0,s,m)););
  -2:  [2/-c*a - 1/-c^2*b^2,   1/-c^2*a^2 + 2/-c^2*b,   1/c^2]
  -1:  [-1/c*b,   1/-c*a,   1/c]
   0:  [3,   -3,   1]
   1:  [a,   b,   c]
   2:  [a^2 + 2*b,   2*c*a - b^2,   c^2]
   3:  [a^3 + 3*b*a + 3*c,   -3*c*b*a + (b^3 - 3*c^2),   c^3]
   4:  [a^4 + 4*b*a^2 + 4*c*a + 2*b^2, \
        -2*c^2*a^2 + 4*c*b^2*a + (-b^4 + 4*c^2*b), \
        c^4]
   5:  [a^5 + 5*b*a^3 + 5*c*a^2 + 5*b^2*a + 5*c*b, \
        5*c^2*b*a^2 + (-5*c*b^3 + 5*c^3)*a + (b^5 - 5*c^2*b^2), \
        c^5]
```

19.1.5 Binet forms for recurrences

A closed form expression for the Fibonacci numbers is

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \tag{19.18}$$

For a two-term recurrence $a_n = m_1\, a_{n-1} + m_2\, a_{n-2}$ a closed form solution is given by

$$a_n = \frac{1}{w} \left[(a_1 - a_0\, M)\, P^n - (a_1 - a_0\, P)\, M^n \right] \tag{19.19a}$$

where $w = \sqrt{m_1^2 + 4\, m_2}$, $P = (m_1 + w)/2$ and $M = (m_1 - w)/2$.

In general such formulas can be obtained as exemplified using a three-term recurrence: Let $a_n = m_1\, a_{n-1} + m_2\, a_{n-2} + m_3\, a_{n-3}$, its characteristic polynomial is $p(x) = x^3 - (m_1\, x^2 + m_2\, x + m_3)$. Let r_0, r_1, r_2 be the roots of $p(x)$, then $a_n = c_0\, r_0^n + c_1\, r_1^n + c_2\, r_2^n$ if c_0, c_1, c_2 satisfy

$$a_0 = c_0 + c_1 + c_2 \tag{19.20a}$$

$$a_1 = r_0\, c_0 + r_1\, c_1 + r_2\, c_2 \tag{19.20b}$$

$$a_2 = r_0^2\, c_0 + r_1^2\, c_1 + r_2^2\, c_2 \tag{19.20c}$$

That is, we have to solve the matrix equation $Z \cdot c = a$ for the vector c where a is the vector of starting values and

$$Z = \begin{bmatrix} 1 & 1 & 1 \\ r_0 & r_1 & r_2 \\ r_0^2 & r_1^2 & r_2^2 \end{bmatrix} \tag{19.21}$$

Verification for the sequence $a_n = 4\, a_{n-1} - a_{n-2}$ that starts with $a_0 = 1$, $a_1 = 3$:

```
? a=[1,3]~;
? m=[4,-1]~;
? K=length(m);
? p=x^K-sum(k=1,K,m[k]*x^(K-k))
  x^2 - 4*x + 1
? r=(polroots(p))
  [0.267949192431122 + 0.E-19*I,  3.73205080756887 + 0.E-19*I]~
? Z=matrix(K,K,ri,ci,r[ci]^(ri-1))
  [1 1]
  [0.267949192431122 + 0.E-19*I   3.73205080756887 + 0.E-19*I]
? c=matsolve(Z,a)
  [0.211324865405187 + 0.E-20*I,  0.788675134594813 + 0.E-19*I]~
? norm(Z*c-a)
  [0.E-38,  4.70197740 E-38]~
? seq(n)=sum(k=0,K-1,c[k+1]*r[k+1]^n)
? for(n=0,7,print(n,": ",seq(n)))
  0: 1.00000000000000 + 0.E-19*I
  1: 3.00000000000000 + 0.E-18*I
  2: 11.0000000000000 + 0.E-18*I
  3: 41.0000000000000 + 0.E-17*I
  4: 153.000000000000 + 0.E-17*I
  5: 571.000000000000 + 0.E-16*I
  6: 2131.00000000000 + 0.E-15*I
  7: 7953.00000000000 + 0.E-15*I
```

With the three term recurrence $a_n = a_{n-1} + a_{n-2} + a_{n-2}$ starting with $a_0 = a_1 = 0$ and $a_2 = 1$ we get:

```
? a=[0,0,1]~;
? m=[1,1,1]~;
? K=length(m);
? p=x^K-sum(k=1,K,m[k]*x^(K-k))
  x^3 - x^2 - x - 1
? r=(polroots(p))
  [1.8392867,   -0.419643 - 0.606290*I,  -0.419643 + 0.6062907*I]~
? Z=matrix(K,K,ri,ci,r[ci]^(ri-1))
  [1 1 1]
  [1.839286   -0.4196433 - 0.6062907*I   -0.4196433 + 0.6062907*I]
  [3.382975   -0.1914878 + 0.5088517*I   -0.1914878 - 0.5088517*I]
? c=matsolve(Z,a)
  [0.1828035 + 1.8947 E-20*I,  -0.09140176 - 0.3405465*I,  -0.0914017 + 0.3405465*I]~
? norm(Z*c-a)
  [1.147 E-39,  6.795 E-39,  3.673 E-39]~
? seq(n)=sum(k=0,K-1,c[k+1]*r[k+1]^n)
? for(n=0,20,print1(" ",round(seq(n))))
  0 0 1 1 2 4 7 13 24 44 81 149 274 504 927 1705 3136 5768 10609 19513 35890
```

The method fails if the characteristic polynomial has multiple roots because then the matrix Z is singular.

Binet form with multiple roots of the characteristic polynomial

When the characteristic polynomial has multiple roots the Binet form has coefficients that are polynomials in n. For example, for the characteristic polynomials $p(x) = (x - r_0)^3 (x - r_1)$ the Binet from would be $a_n = (c_0 + n\,d_0 + n^2\,e_0)\,r_0^n + c_1\,r_1^n$. With $n = 0$, 1, and 2 we obtain the system of equations

$$a_0 = (c_0 + 0\,d_0 + 0^2\,e_0) + c_1 \tag{19.22a}$$
$$a_1 = r_0\,(c_0 + 1\,d_0 + 1^2\,e_0) + r_1\,c_1 \tag{19.22b}$$
$$a_2 = r_0^2\,(c_0 + 2\,d_0 + 2^2\,e_0) + r_1^2\,c_1 \tag{19.22c}$$

In general, the coefficient of the power of the k-th root r_k in the Binet form must be a polynomial of degree $m_k - 1$ where m_k is the multiplicity of r_k.

19.1.6 Generating functions for recurrences

A generating function for a recurrence has a Taylor series where the k-th coefficient equals the k-th term of the recurrence. For example, for the Fibonacci- and Lucas numbers:

$$\frac{x}{1-x-x^2} = 0 + x + x^2 + 2x^3 + 3x^4 + 5x^5 + 8x^6 + 13x^7 + 21x^8 + 34x^9 + \dots \tag{19.23a}$$

$$= \sum_{k=0}^{\infty} F_k\, x^k \tag{19.23b}$$

$$\frac{2-x}{1-x-x^2} = 2 + x + 3x^2 + 4x^3 + 7x^4 + 11x^5 + 18x^6 + 29x^7 + 47x^8 + \dots \tag{19.23c}$$

$$= \sum_{k=0}^{\infty} L_k\, x^k \tag{19.23d}$$

In general, for a recurrence $a_n = \sum_{k=1}^{K} m_k\, a_{n-k}$ with given a_0, a_1, …, a_K one has

$$\frac{\sum_{j=0}^{K-1} b_j\, x^j}{1 - \sum_{j=1}^{K} m_j\, x^j} = \sum_{j=0}^{\infty} a_j\, x^j \tag{19.24a}$$

where the denominator is the reciprocal polynomial of the characteristic polynomial and

$$b_0 = a_0 \tag{19.25a}$$

$$b_1 = a_1 - (a_0\, m_1) \tag{19.25b}$$

$$b_2 = a_2 - (a_0\, m_2 + a_1\, m_1) \tag{19.25c}$$

$$b_3 = a_3 - (a_0\, m_3 + a_1\, m_2 + a_2\, m_1) \tag{19.25d}$$

$$b_k = a_k - \sum_{j=0}^{k-1} a_j\, m_{k-j} \tag{19.25e}$$

As an example we choose the sequence

$$[0,\, 0,\, 1,\, 1,\, 2,\, 4,\, 7,\, 13,\, 24,\, 44,\, 81,\, 149,\, 274,\, \dots] \tag{19.26}$$

with the recurrence relation $a_n = a_{n-1} + a_{n-2} + a_{n-2}$ that was an example for the Binet form:

```
? a=[0,0,1]~;
? m=[1,1,1]~;
? K=length(m);
? b=vector(K, k, a[k]-sum(j=0,k-2, a[j+1]*m[k-j-1]))
  [0, 0, 1]
? pb=sum(j=0,K-1,b[j+1]*x^j)
  x^2
? pr=1-sum(k=1,K,m[k]*x^k) \\ reciprocal of charpoly
  -x^3 - x^2 - x + 1
? gen=pb/pr  \\ the generating function
  x^2/(-x^3 - x^2 - x + 1)
? t=taylor(gen, x)
  x^2 + x^3 + 2*x^4 + 4*x^5 + 7*x^6 + 13*x^7 + 24*x^8 + 44*x^9 + 81*x^10
  + 149*x^11 + 274*x^12 + 504*x^13 + 927*x^14 + 1705*x^15 + 3136*x^16 + O(x^17)
? t=truncate(t);
? for(j=0,poldegree(t),print1("  ",polcoeff(t,j)))
  0  0  1  1  2  4  7  13  24  44  81  149  274  504  927  1705  3136
```

The general form of the expressions for a two term linear recurrence can be obtained using symbols:

```
? a=[a0,a1]~;
? m=[m1,m2]~;
? K=length(m);
? b=vector(K,k,a[k]-sum(j=0,k-2,a[j+1]*m[k-j-1]))
  [a0, -m1*a0 + a1]
? pb=sum(j=0,K-1,b[j+1]*x^j)
```

```
  (-m1*a0 + a1)*x + a0
? pr=1-sum(k=1,K,m[k]*x^k)
  -m2*x^2 - m1*x + 1
? gen=pb/pr  \\ the generating function
  ((-m1*a0 + a1)*x + a0)/(-m2*x^2 - m1*x + 1)
? t=taylor(gen,x);
? t=truncate(t);
? for(j=0,poldegree(t),print(j,":  ",polcoeff(t,j)))
  0:  a0
  1:  a1
  2:  m2*a0 + m1*a1
  3:  m2*m1*a0 + (m1^2 + m2)*a1
  4:  (m2*m1^2 + m2^2)*a0 + (m1^3 + 2*m2*m1)*a1
  5:  (m2*m1^3 + 2*m2^2*m1)*a0 + (m1^4 + 3*m2*m1^2 + m2^2)*a1
  6:  (m2*m1^4 + 3*m2^2*m1^2 + m2^3)*a0 + (m1^5 + 4*m2*m1^3 + 3*m2^2*m1)*a1
  7:  (m2*m1^5 + 4*m2^2*m1^3 + 3*m2^3*m1)*a0 + (m1^6 + 5*m2*m1^4 + 6*m2^2*m1^2 + m2^3)*a1
```

19.1.7 Logarithms of generating functions *

A seemingly mysterious relation for the generating function of the Fibonacci numbers

$$f(x) \;:=\; \frac{1}{1-x-x^2} \;=\; 1+x+2x^2+3x^3+5x^4+8x^5+\ldots \;=\; \sum_{k=0}^{\infty} F_{k+1}\,x^k \tag{19.27}$$

is

$$\log(f(x)) \;=\; x+\frac{1}{2}\,3\,x^2+\frac{1}{3}\,4\,x^3+\frac{1}{4}\,7\,x^4+\frac{1}{5}\,11\,x^5+\frac{1}{6}\,18\,x^6+\frac{1}{7}\,29\,x^7+\ldots \tag{19.28}$$

$$\;=\; \sum_{k=1}^{\infty}\frac{1}{k}\,L_k\,x^k \;=\; \frac{1}{k}\cdot(\text{Lucas numbers}) \tag{19.29}$$

Similarly,

$$g(x) \;:=\; \frac{1}{1-2\,x-x^2} \;=\; 1+2x+5x^2+12x^3+29x^4+70x^5+169x^6+\ldots \tag{19.30a}$$

$$\log(g(x)) \;=\; 2\left[x+\frac{1}{2}\,3\,x^2+\frac{1}{3}\,7\,x^3+\frac{1}{4}\,17\,x^4+\frac{1}{5}\,41\,x^5+\frac{1}{6}\,99\,x^6+\ldots\right] \tag{19.30b}$$

Now set $f(x) =: \frac{1}{h(x)}$, then

$$\frac{d}{d\,x}\log(f(x)) \;=\; \frac{d}{d\,x}\log\left(\frac{1}{h(x)}\right) \;=\; -\frac{h'(x)}{h(x)} \tag{19.31}$$

The expression $\frac{h'(x)}{h(x)}$ is again the generating function of a recurrence and formal integration of the Taylor series terms gives the factors $\frac{1}{k}$. The observation is a special case of the algorithm for the computation of the logarithm for powers series given in section 17.4 on page 437.

19.2 Chebyshev polynomials

The Chebyshev polynomials of the first (T) and second (U) kind can be defined by the functions

$$T_n(x) \;=\; \cos\left[n\,\arccos(x)\right] \tag{19.32a}$$

$$U_n(x) \;=\; \frac{\sin\left[(n+1)\,\arccos(x)\right]}{\sqrt{1-x^2}} \tag{19.32b}$$

For integral n both of them are polynomials. The first few polynomials are given in figure 19.1 on the following page (first kind) and figure 19.2 on the next page (second kind).

$$
\begin{aligned}
T_{-n}(x) &= T_n(x) \\
T_{-1}(x) &= x \\
T_0(x) &= 1 \\
T_1(x) &= x \\
T_2(x) &= 2\,x^2 - 1 \\
T_3(x) &= 4\,x^3 - 3\,x \\
T_4(x) &= 8\,x^4 - 8\,x^2 + 1 \\
T_5(x) &= 16\,x^5 - 20\,x^3 + 5\,x \\
T_6(x) &= 32\,x^6 - 48\,x^4 + 18\,x^2 - 1 \\
T_7(x) &= 64\,x^7 - 112\,x^5 + 56\,x^3 - 7\,x \\
T_8(x) &= 128\,x^8 - 256\,x^6 + 160\,x^4 - 32\,x^2 + 1 \\
T_9(x) &= 256\,x^9 - 576\,x^7 + 432\,x^5 - 120\,x^3 + 9\,x \\
T_{10}(x) &= 512\,x^{10} - 1280\,x^8 + 1120\,x^6 - 400\,x^4 + 50\,x^2 - 1 \\
T_{11}(x) &= 1024\,x^{11} - 2816\,x^9 + 2816\,x^7 - 1232\,x^5 + 220\,x^3 - 11\,x
\end{aligned}
$$

Figure 19.1: The first few Chebyshev polynomials of the first kind.

$$
\begin{aligned}
U_{-n}(x) &= -U_{n-2}(x) \\
U_{-2}(x) &= -1 \\
U_{-1}(x) &= 0 \\
U_0(x) &= 1 \\
U_1(x) &= 2\,x \\
U_2(x) &= 4\,x^2 - 1 \\
U_3(x) &= 8\,x^3 - 4\,x \\
U_4(x) &= 16\,x^4 - 12\,x^2 + 1 \\
U_5(x) &= 32\,x^5 - 32\,x^3 + 6\,x \\
U_6(x) &= 64\,x^6 - 80\,x^4 + 24\,x^2 - 1 \\
U_7(x) &= 128\,x^7 - 192\,x^5 + 80\,x^3 - 8\,x \\
U_8(x) &= 256\,x^8 - 448\,x^6 + 240\,x^4 - 40\,x^2 + 1 \\
U_9(x) &= 512\,x^9 - 1024\,x^7 + 672\,x^5 - 160\,x^3 + 10\,x \\
U_{10}(x) &= 1024\,x^{10} - 2304\,x^8 + 1792\,x^6 - 560\,x^4 + 60\,x^2 - 1 \\
U_{11}(x) &= 2048\,x^{11} - 5120\,x^9 + 4608\,x^7 - 1792\,x^5 + 280\,x^3 - 12\,x
\end{aligned}
$$

Figure 19.2: The first few Chebyshev polynomials of the second kind.

One has

$$T_n(x) = \frac{n}{2} \sum_{k=0}^{\lfloor n/2 \rfloor} \frac{(-1)^k \, (n-k-1)! \, (2x)^{n-2k}}{k! \, (n-2k)!} \tag{19.35a}$$

$$= \frac{n}{2} \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \, \frac{1}{n-k} \binom{n-k}{k} (2x)^{n-2k} \tag{19.35b}$$

$$= \sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{2\,k} x^{n-2k} \, (x^2-1)^k \tag{19.35c}$$

and

$$U_n(x) = \sum_{k=0}^{\lfloor n/2 \rfloor} \frac{(-1)^k \, (n-k)! \, (2x)^{n-2k}}{k! \, (n-2k)!} \tag{19.36a}$$

$$= \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \binom{n-k}{k} (2x)^{n-2k} \tag{19.36b}$$

$$= \sum_{k=0}^{\lfloor n/2+1 \rfloor} \binom{n+1}{2\,k+1} x^{n-2k} \, (x^2-1)^k \tag{19.36c}$$

The indexing of U seems to be slightly unfortunate, having $U_0 = 0$ would render many of the relations for the Chebyshev polynomials more symmetric.

The $n+1$ extrema of $T_n(x)$ are located at the points $x_k = \cos \frac{k\,\pi}{n}$ where $k = 0, 1, 2, \ldots, n$ and $-1 \leq x_k \leq +1$, which can be seen from the definition. The values at those points are ± 1. The n zeros lie at $x_k = \cos \frac{(k-1/2)\,\pi}{n}$ where $k = 1, 2, 3, \ldots, n$.

The expansion of x^n in terms of Chebyshev polynomials of the first kind is

$$x^n = \frac{1}{2^n} \binom{n}{n/2} + \frac{1}{2^{n-1}} \sum_{k=0}^{n/2-1} \binom{n}{k} T_{n-2k}(x) \tag{19.37a}$$

for n even, and

$$x^n = \frac{1}{2^{n-1}} \sum_{k=0}^{(n-1)/2} \binom{n}{k} T_{n-2k}(x) \tag{19.37b}$$

for n odd.

For the Chebyshev polynomials of the first kind one has

$$T_n\left(\frac{x+1/x}{2}\right) = \frac{x^n + 1/x^n}{2} \tag{19.38}$$

This relation allows to find a solution of $T_n(x) = z$ directly. Indeed

$$x = \frac{R_n + 1/R_n}{2} \quad \text{where} \quad R_n := \left(z + \sqrt{z^2-1}\right)^{1/n} \tag{19.39}$$

is a solution which can be chosen to be real if $z \in \mathbb{R}$ and $z > 1$.

19.2.1 Recurrence relation, generating functions and the composition law

Both types of Chebyshev polynomials obey the same recurrence (omitting argument x)

$$N_n = 2\,x\,N_{n-1} - N_{n-2} \tag{19.40}$$

where N can be either symbol, T or U.

Recurrence relations for subsequences are:

$$N_{n+1} = [2\,x]\cdot N_n - N_{n-1} \tag{19.41a}$$
$$N_{n+2} = \left[2\,(2x^2-1)\right]\cdot N_n - N_{n-2} \tag{19.41b}$$
$$N_{n+3} = \left[2\,(4x^3-3x)\right]\cdot N_n - N_{n-3} \tag{19.41c}$$
$$N_{n+4} = \left[2\,(8x^4-8x^2+1\right]\cdot N_n - N_{n-4} \tag{19.41d}$$
$$N_{n+5} = \left[2\,(16x^5-20x^3+5x)\right]\cdot N_n - N_{n-5} \tag{19.41e}$$
$$N_{n+s} = [2\,T_s(x)]\cdot N_n - N_{n-s} \tag{19.41f}$$

The generating functions are

$$\frac{1-x\,t}{1-2\,x\,t+t^2} = \sum_{n=0}^{\infty} t^n\,T_n(x) \tag{19.42a}$$
$$\frac{1}{1-2\,x\,t+t^2} = \sum_{n=0}^{\infty} t^n\,U_n(x) \tag{19.42b}$$

Quick check of relation 19.42a using pari/gp:

```
? gen=truncate(taylor((1-t*x)/(1-2*x*t+t^2),t))
 [...]
? for(k=0,5,print(k,":  ",polcoeff(gen,k,t)));
 0:  1
 1:  x
 2:  2*x^2 - 1
 3:  4*x^3 - 3*x
 4:  8*x^4 - 8*x^2 + 1
 5:  16*x^5 - 20*x^3 + 5*x
```

Binet forms for T and U are:

$$T_n(z) = \frac{1}{2}\left[\left(z+\sqrt{z^2-1}\right)^n + \left(z-\sqrt{z^2-1}\right)^n\right] \tag{19.43a}$$
$$U_n(z) = \frac{1}{2\sqrt{z^2-1}}\left[\left(z+\sqrt{z^2-1}\right)^{n+1} - \left(z-\sqrt{z^2-1}\right)^{n+1}\right] \tag{19.43b}$$

Composition is multiplication of indices which can be seen by the definition:

$$T_n(T_m(x)) = T_{n\,m}(x) \tag{19.44}$$

For example,

$$T_{2n}(x) = T_2(T_n(x)) = 2\,T_n^2(x) - 1 \tag{19.45a}$$
$$= T_n(T_2(x)) = T_n(2\,x^2-1) \tag{19.45b}$$

19.2.2 Index-doubling and relations between T and U

Index-doubling relations for the polynomials of the first kind are

$$T_{2n} = 2\,T_n^2 - 1 \tag{19.46a}$$
$$T_{2n+1} = 2\,T_{n+1}\,T_n - x \tag{19.46b}$$
$$T_{2n-1} = 2\,T_n\,T_{n-1} - x \tag{19.46c}$$

Similar relation for the polynomials of the second kind are

$$\begin{aligned}
U_{2n} &= U_n^2 - U_{n-1}^2 \;=\; (U_n + U_{n-1})\,(U_n - U_{n-1}) && (19.47a)\\
&= U_n\,(U_n - U_{n-2}) - 1 \;=\; U_{n-1}\,(U_{n+1} - U_{n-1}) + 1 && (19.47b)\\
U_{2n+1} &= U_n\,(U_{n+1} - U_{n-1}) && (19.47c)\\
&= 2\,U_n\,(U_{n+1} - x\,U_n) \;=\; 2\,U_n\,(x\,U_n - U_{n-1}) && (19.47d)\\
U_{2n-1} &= U_{n-1}\,(U_n - U_{n-2}) && (19.47e)\\
&= 2\,U_{n-1}\,(U_n - x\,U_{n-1}) \;=\; 2\,U_{n-1}\,(x\,U_{n-1} - U_{n-2}) && (19.47f)
\end{aligned}$$

Some relations between T and U are

$$\begin{aligned}
T_n &= U_n - x\,U_{n-1} \;=\; x\,U_{n-1} - U_{n-2} \;=\; \frac{1}{2}\,(U_n - U_{n-2}) && (19.48a)\\
T_{n+1} &= x\,T_n - (1-x^2)\,U_{n-1} && (19.48b)\\
U_{2n} &= 2\,T_n\,U_n - 1 && (19.48c)\\
U_{2n-1} &= 2\,T_n\,U_{n-1} \;=\; 2\,(T_{n+1}\,U_n + x) && (19.48d)\\
U_{2n+1} &= 2\,T_{n+1}\,U_n \;=\; 2\,(T_{n+2}\,U_{n-1} + x) && (19.48e)\\
U_{2^n-1} &= 2^n \prod_{k=0}^{n-1} T_{2^k} && (19.48f)
\end{aligned}$$

Relation 19.48b, written as

$$U_n \;=\; \frac{x\,T_{n+1} - T_{n+2}}{1-x^2} \;=\; \frac{T_n - x\,T_{n+1}}{1-x^2} \qquad (19.49)$$

allows to compute the polynomials of the second kind from those of the first kind.

One further has:

$$\begin{aligned}
T_{n+m} + T_{n-m} &= 2\,T_n\,T_m && (19.50a)\\
T_{n+m} - T_{n-m} &= 2\,(x^2+1)\,U_{n-1}\,U_{m-1} && (19.50b)\\
U_{n+m-1} + U_{n-m-1} &= 2\,U_{n-1}\,T_m && (19.50c)\\
U_{n+m-1} - U_{n-m-1} &= 2\,T_n\,U_{m-1} && (19.50d)
\end{aligned}$$

Summation:

$$\begin{aligned}
\sum_{k=0}^{n} T_{2k} &= \frac{1}{2}\,(1 + U_{2n}) && (19.51a)\\
\sum_{k=0}^{n-1} T_{2k+1} &= \frac{1}{2}\,U_{2n-1} && (19.51b)\\
\sum_{k=0}^{n} U_{2k} &= \frac{1 - T_{2n+2}}{2\,(1-x^2)} && (19.51c)\\
\sum_{k=0}^{n-1} U_{2k+1} &= \frac{x - T_{2n+1}}{2\,(1-x^2)} && (19.51d)
\end{aligned}$$

Using $\partial_x \cos(n\,\arccos(x)) = n\,\sin(n\,\arccos(x))/\sqrt{1-x^2}$ we obtain

$$\partial_x T_n(x) \;=\; n\,U_{n-1}(x) \qquad (19.52)$$

19.2.3 Fast computation of the Chebyshev polynomials

We give algorithms that improve on both the matrix power, and the polynomial based algorithms.

Chebyshev polynomials of the first kind

For even index use relation 19.46a ($T_{2n} = 2\,T_n^2 - 1$). For odd index we use relations 19.46c and 19.46b. We compute the pair $[T_{n-1},\, T_n]$ recursively via

$$[T_{n-1},\, T_n] \;=\; \left[2\,T_{q-1}\,T_q - x,\; 2\,T_q^2 - 1\right] \quad \text{where} \quad q = n/2, \quad \text{if } n \text{ even} \tag{19.53a}$$

$$[T_{n-1},\, T_n] \;=\; \left[2\,T_{q-1}^2 - 1,\; 2\,T_{q-1}\,T_q - x\right] \quad \text{where} \quad q = (n+1)/2, \quad \text{if } n \text{ odd} \tag{19.53b}$$

Note that no multiplication with x occurs thereby the computation is efficient also for floating point arguments. With integer x the cost of the computation of $T_n(x)$ is $\sim M(n)$ where $M(n)$ is the cost of a multiplication of numbers with the precision of the result. When x is a floating point number the cost is $\sim \log_2(n)\, M(n)$ where $M(n)$ is the cost of a multiplication with the precision used.

The code for the pair computations is

```
fvT(n, x)=
{ /* return  [ T(n-1,x), T(n,x) ]  */
    local(nr, t, t1, t2);
    if ( n<=1,
        if ( 1==n, return( [1, x] ) );
        if ( 0==n, return( [x, 1] ) );
        if ( -1==n, return( [2*x^2-1, x] ) );
        return( 0 );  \\ disallow negative index < -1
    );
    nr = (n+1) >> 1; \\   if ( "n even",  nr = n/2 , nr = (n+1)/2; );
    vr = fvT(nr, x);  \\ recursion
    t1 = vr[1];  t2 = vr[2];
    if ( !bitand(n,1), \\ n is even
        t = [2*t1*t2-x, 2*t2^2-1];
        ,
        t = [2*t1^2-1, 2*t1*t2-x];
     );
    return( t );
}
```

The function to be called by the user is

```
fT(n, x)=
{
    local(q, t, v, T);
    n = abs(n);
    if ( n<=1,
        if (n>=0,  return(if(0==n,1,x)));
        return( fT(-n, x) );
    );
    t = 0;  q = n;
    while ( 0==bitand(q, 1), q>>=1; t+=1; );
    \\ here: n==q*2^t
    T = fvT(q, x)[2];
    while ( t,  T=2*T*T-1; t-=1; );
    return( T );
}
```

We check the speedup by comparing with the matrix-power computation that gives identical results. We compute $T_{4{,}545{,}967}(2)$, a number with more than 2,600,000 decimal digits:

```
vT(n,x)= return( ([1, x]*[0,-1; 1,2*x]^n) );
x=2;  \\ want integer calculations
n=4545967;
vT(n,x);   \\ computed in 9,800 ms.
fvT(n,x);  \\ computed in 2,241 ms.
```

C++ implementations for the computation of $T_n(2)$ and $T_n(x)$ modulo m are given in [FXT: file mod/chebyshev1.cc].

Chebyshev polynomials of the second kind

One can use the fast algorithm for the polynomials of the first kind and relation 19.49 ($U_n = (T_n - x\,T_{n+1})/(1-x^2)$, involving a division):

```
fU(n, x)=
{
    local(v);
    if ( 1==x, return(n+1) );  \\ avoid division by zero
    v = fvT(n+1, x);
    return( (v[1]-x*v[2])/(1-x^2) );
}
```

We give an additional algorithm that uses 3 multiplication for each reduction of the index n. One multiplication is by the variable x. We compute the pair $[U_{n-1},\, U_n]$ recursively via

$$M_q := (U_q + U_{q-1})\,(U_q - U_{q-1}) \tag{19.54a}$$

$$[U_{n-1},\, U_n] = [2\,U_{q-1}\,(U_q - x\,U_{q-1}),\; M_q] \quad \text{where} \quad q = n/2, \quad \text{if } n \text{ even} \tag{19.54b}$$

$$[U_{n-1},\, U_n] = [M_q,\; 2\,U_q\,(x\,U_q - U_{q-1})] \quad \text{where} \quad q = (n-1)/2, \quad \text{if } n \text{ odd} \tag{19.54c}$$

The code for the pair computations is

```
fvU(n, x)=
{ /* return  [ U(n-1,x), U(n,x) ] */
    local(nr, u1, u0, ue, t, u);
    if ( n<=1,
        if ( 1==n, return( [1, (2*x)] ) );
        if ( 0==n, return( [0, 1] ) );
        if ( -1==n, return( [-1, 0] ) );
        if ( -2==n, return( [-(2*x), -1] ) );
        return( 0 );  \\ disallow negative index < -2
    );
    nr = n >> 1;  \\ if ( "n even",  nr = n/2 , nr = (n-1)/2; );
    vr = fvU(nr, x);  \\ recursion
    u1 = vr[1];  u0 = vr[2];
    ue = (u0+u1) * (u0-u1);
    if ( !bitand(n,1),  \\ n is even
        t = u1*(u0-x*u1);  t+=t;
        u = [t,  ue];
        ,
        t = u0*(x*u0-u1);  t+=t;
        u = [ue, t];
    );
    return( u );
}
```

The function to be called by the user is

```
fU(n, x)=  return( fvU(n,x)[2] );
```

The comparison with the matrix-power computation gives almost the same speedup as for the polynomials of the first kind:

```
vU(n,x)= return( [0, 1]*[0,-1; 1,2*x]^n );
x=2;  \\ want integer calculations
n=4545967;
vU(n,x);   \\ computed in 9,783 ms.
fvU(n,x);  \\ computed in 2,704 ms.
```

The routine using the polynomials of the first kind has about the same speed for integer arguments.

C++ implementations for the computation of $U_n(2)$ and $U_n(x)$ modulo m are given in [FXT: file mod/chebyshev2.cc].

Symbolic computations

For symbolic computations the explicit power series as in 19.35a or 19.36a on page 473 should be preferred. The following routine computes T_n as a polynomial in x:

```
chebyTrec(n)=
{
    local(b, s);
    b = 2^(n-1);
    if ( 0==n%2, if ( 0==n%4, s=+1, s=-1 ), s=0 );
    forstep (k=n, 1, -2,
        s += b*'x^(k);
        b *= -(k*(k-1))/((n+k-2)*(n-k+2));
    );
    return( Pol(s) );
}
```

For U_n one can use

```
chebyUrec(n)=
{
    local(b, s);
    n += 1;
    b = 2^(n-1)/n;
    s = 0;
    forstep (k=n, 1, -2,
        s += (k)*b*'x^(k-1);
        b *= -(k*(k-1))/((n+k-2)*(n-k+2));
    );
    return( Pol(s) );
}
```

19.2.4 Relations to approximations of the square root *

Padé approximants for $\sqrt{x^2 \pm 1}$

We start with the relation

$$T_n^2 - (x^2-1)\,U_{n-1}^2 \;=\; 1 \tag{19.55}$$

which we write as

$$\sqrt{x^2-1} \;=\; \sqrt{\frac{T_n^2-1}{U_{n-1}^2}} \tag{19.56}$$

If we define $R_n = T_n/U_{n-1}$, then

$$R_n(x) \;=\; \frac{T_n}{U_{n-1}} \;\approx\; \sqrt{x^2-1} \tag{19.57}$$

A composition law holds for $R(.)$:

$$R_{m\,n}(x) \;=\; R_m(R_n(x)) \tag{19.58}$$

We list the first few values of $R_k(2)$ and $R_k(x)$:

k:	$R_k(2)$	$R_k(x)$
1:	$2/1$	$\dfrac{x}{1}$
2:	$7/4$	$\dfrac{2x^2-1}{2x}$
3:	$26/15$	$\dfrac{4x^3-3x}{4x^2-1}$
4:	$97/56$	$\dfrac{8x^4-8x^2+1}{8x^3-4x}$
5:	$362/209$	$\dfrac{16x^5-20x^3+5x}{16x^4-12x^2+1}$
∞:	$\sqrt{3}$	$\sqrt{x^2-1}$

(19.59)

If we define $T_n^+(x) := T(i\,x)/i^n$ and $U_n^+(x) := U(i\,x)/i^n$ then

$$T_n^{+\,2} - (x^2+1)\,U_{n-1}^{+\,2} \quad = \quad 1 \tag{19.60}$$

Defining $R_n^+ := T_n^+/U_{n-1}^+$ we have

$$R_{m\,n}^+(x) \quad = \quad R_m^+(R_n^+(x)) \tag{19.61}$$

and

$$\sqrt{x^2+1} \quad = \quad \sqrt{\frac{T_n^{+\,2}-1}{U_{n-1}^{+\,2}}} \quad \approx \quad \frac{T_n^+}{U_{n-1}^+} \;=\; R_n^+(x) \tag{19.62}$$

The first few values of $R_k^+(1)$ and $R_k^+(x)$ are

$$\begin{array}{ccc} k\text{:} & R_k^+(1) & R_k^+(x) \\ 1\text{:} & 1/1 & \dfrac{x}{1} \\ 2\text{:} & 3/2 & \dfrac{2x^2+1}{2x} \\ 3\text{:} & 7/5 & \dfrac{4x^3+3x}{4x^2+1} \\ 4\text{:} & 17/12 & \dfrac{8x^4+8x^2+1}{8x^3+4x} \\ 5\text{:} & 41/29 & \dfrac{16x^5+20x^3+5x}{16x^4+12x^2+1} \\ \infty\text{:} & \sqrt{2} & \sqrt{x^2+1} \end{array} \tag{19.63}$$

Two products for the square root

For those fond of products: for $d > 0$, $d \neq 1$

$$\sqrt{d} \quad = \quad \prod_{k=0}^{\infty}\left(1+\frac{1}{q_k}\right) \quad \text{where} \quad q_0 = \frac{d+1}{d-1}, \quad q_{k+1} = 2q_k^2 - 1 \tag{19.64}$$

(convergence is quadratic) and

$$\sqrt{d} \quad = \quad \prod_{k=0}^{\infty}\left(1+\frac{2}{h_k}\right) \quad \text{where} \quad h_0 = \frac{d+3}{d-1}, \quad h_{k+1} = (h_k+2)^2\,(h_k-1)+1 \tag{19.65}$$

(convergence is cubic). These are given in [16], the first is ascribed to Friedrich Engel. The paper gives $h_{k+1} = \frac{4\,d}{d-1}\prod_{i=0}^{k}\left(h_i^2\right) - 3$. Note that for relation 19.64

$$q_k \quad = \quad T_{2^k}(q_0) \tag{19.66}$$

$$\frac{1}{q_k} \quad = \quad \frac{(d-1)^N}{\sum_{i=0}^{N}\binom{2\,N}{2\,i}\,d^i} \;=\; \frac{2\,(1-d)^N}{(1+\sqrt{d})^{2N}+(1-\sqrt{d})^{2N}} \quad \text{where} \quad N = 2^k \tag{19.67}$$

where T_n is the n-th Chebyshev polynomial of the first kind. One finds

$$q_k \quad = \quad T_{2^k}(1/c) \quad \text{where} \quad c = \frac{1-d}{1+d}, \quad c < 1 \tag{19.68}$$

and

$$\sqrt{\frac{1-c}{1+c}} \quad \approx \quad \frac{1-c}{c}\,\frac{U_{2^k-1}(1/c)}{T_{2^k}(1/c)} \tag{19.69}$$

which can be expressed in $d = \frac{1-c}{1+c}$ as

$$\sqrt{d} \quad \approx \quad \frac{2\,d}{1-d}\,\frac{U_{2^k-1}(\frac{1+d}{1-d})}{T_{2^k}(\frac{1+d}{1-d})} \quad \text{where} \quad d > 1 \tag{19.70}$$

where U_n is the n-th Chebyshev polynomial of the second kind. We have $U_{2^k-1}(x) = 2^k \prod_{i=0}^{k-1} T_{2^i}(x)$. A computation might successively compute $T_{2^i} = 2\,T_{2^{i-1}}^2 - 1$, accumulate the product $U_{2^i-1} = 2\,U_{2^{i-1}-1}\,T_{2^{i-1}}$ until U_{2^k-1} and T_{2^k} are obtained. Alternatively one can use the relation $U_k(x) = \frac{1}{k+1}\,\partial_x T_{k+1}(x)$ and use the recursion for the coefficients of T as shown on page 477.

Chapter 20

Miscellaneous arithmetical algorithms

This chapter contains material that does not fit in elsewhere...

The binary splitting method and the sumalt algorithm are described, followed by material about continued fractions and hypergeometric functions.

20.1 The binary splitting algorithm for rational series

The straightforward computation of a series for which each term adds a constant amount of precision[1] to a precision of N digits involves the summation of proportional N terms. To get N bits of precision one has to add proportional N terms of the sum, each term involves one (length-N) short division (and one addition). Therefore the total work is proportional N^2, which makes it impossible to compute billions of digits from linearly convergent series even if they are as 'good' as Chudnovsky's famous series for π:

$$\frac{1}{\pi} = \frac{6541681608}{\sqrt{640320}^3} \sum_{k=0}^{\infty} \left(\frac{13591409}{545140134} + k \right) \left(\frac{(6k)!}{(k!)^3\,(3k)!} \frac{(-1)^k}{640320^{3k}} \right) \tag{20.1a}$$

$$= \frac{12}{\sqrt{640320}^3} \sum_{k=0}^{\infty} (-1)^k \frac{(6k)!}{(k!)^3\,(3k)!} \frac{13591409 + k\,545140134}{(640320)^{3k}} \tag{20.1b}$$

There is an alternative way to evaluate a sum $\sum_{k=0}^{N-1} a_k$ of rational summands. One looks at the ratios r_k of consecutive terms:

$$r_k := \frac{a_k}{a_{k-1}} \tag{20.2}$$

Set $a_{-1} := 1$ to avoid a special case for $k = 0$. One has

$$\sum_{k=0}^{N-1} a_k =: r_0\,(1 + r_1\,(1 + r_2\,(1 + r_3\,(1 + \ldots (1 + r_{N-1}) \ldots)))) \tag{20.3}$$

Now define

$$r_{m,n} := r_m\,(1 + r_{m+1}\,(\ldots (1 + r_n) \ldots)) \quad \text{where} \quad m < n \tag{20.4a}$$

$$r_{m,m} := r_m \tag{20.4b}$$

[1] e.g. arc-cotangent series with arguments > 1

then

$$r_{m,n} = \frac{1}{a_{m-1}} \sum_{k=m}^{n} a_k \tag{20.5}$$

and especially

$$r_{0,n} = \sum_{k=0}^{n} a_k \tag{20.6}$$

We have

$$\begin{aligned} r_{m,n} &= r_m + r_m \cdot r_{m+1} + r_m \cdot r_{m+1} \cdot r_{m+2} + \dots \\ &\quad \dots + r_m \cdot \dots \cdot r_x + r_m \cdot \dots \cdot r_x \cdot [r_{x+1} + \dots + r_{x+1} \cdot \dots \cdot r_n] \end{aligned} \tag{20.7a}$$

$$= r_{m,x} + \prod_{k=m}^{x} r_k \cdot r_{x+1,n} \tag{20.7b}$$

The product telescopes, one gets (for $m \le x < n$)

$$r_{m,n} = r_{m,x} + \frac{a_x}{a_{m-1}} \cdot r_{x+1,n} \tag{20.8}$$

Note that the algorithm can trivially be extended (or rather simplified) to infinite products.

Implementation using rationals

Now we can formulate the binary splitting algorithm by giving a binsplit function using pari/gp:

```
R(m, n)=
{ /* Rational binsplit */
    local(x, ret);
    if ( m==n, /* then: */
        ret = A(m)/A(m-1);
    , /* else: */
        x = floor( (m+n)/2 );
        ret = R(m, x) + A(x) / A(m-1) * R(x+1, n);
    );
    return( ret );
}
```

Here `A(k)` must be a function that returns the `k`-th term of the series we wish to compute, in addition one must have `a(-1)`=1. For example, to compute arctan(1/10) one would use

```
A(k)=if(k<0, 1, (-1)^(k)/((2*k+1)*10^(2*k+1)));
```

It is instructive to modify the binsplit function to make it self-documenting. Simple enhancements will print the calling tree and the intermediate results in the process:

```
indent(i)=for(k=1,8*i,print1(" "));  \\ print 8*i spaces
Ri(m, n, i=0)=
{ /* Rational binsplit, self documenting */
    local(x, ret);
    indent(i);  print( "R(", m, ", ", n, ")");
    if ( m==n, /* then: */
        ret = A(m)/A(m-1);
    , /* else: */
        x = floor( (m+n)/2 );
        ret = Ri(m, x, i+1) + A(x) / A(m-1) * Ri(x+1, n, i+1);
    );
    indent(i);  print( "^== ", ret);
    return( ret );
}
```

The additional parameter i is the level of depth in the calling tree and the function `indent(i)` indents (by $8 \cdot i$ spaces) the following print output. We use the function $A(k) = 2^{-(k+1)}$ and obtain, upon calling `Ri(0, 6)`, the following output:

```
R(0, 6)
        R(0, 3)
                R(0, 1)
                        R(0, 0)
                        ^== 1/2
                        R(1, 1)
                        ^== 1/2
                ^== 3/4
                R(2, 3)
                        R(2, 2)
                        ^== 1/2
                        R(3, 3)
                        ^== 1/2
                ^== 3/4
        ^== 15/16
        R(4, 6)
                R(4, 5)
                        R(4, 4)
                        ^== 1/2
                        R(5, 5)
                        ^== 1/2
                ^== 3/4
                R(6, 6)
                ^== 1/2
        ^== 7/8
^== 127/128
```

A fragment like

```
                R(2, 3)
                        R(2, 2)
                        ^== 1/2
                        R(3, 3)
                        ^== 1/2
                ^== 3/4
```

says: "`R(2,3)` called `R(2,2)` [which returned 1/2], then called `R(3,3)` [which returned 1/2]. Then `R(2,3)` returned 3/4."

Implementation using integers

In case the programming language used does not provide rational numbers one needs to rewrite formula 20.8 in separate parts for denominator and numerator. With $a_i = p_i/q_i$, $p_{-1} = q_{-1} = 1$ and $r_{m,n} =: U_{m,n}/V_{m,n}$ one gets

$$U_{m,n} = p_{m-1}\, q_x\, U_{m,x}\, V_{x+1,n} + p_x\, q_{m-1}\, U_{x+1,n}\, V_{m,x} \tag{20.9a}$$

$$V_{m,n} = p_{m-1}\, q_x\, V_{m,x}\, V_{x+1,n} \tag{20.9b}$$

The following implementation also contains code for reduction to lowest terms:

```
Q(m, n)=
{ /* Integer binsplit */
    local(x, ret, bm, bx, tm, tx);
    if ( m==n, /* then: */
        bm = B(m);  bx = B(m-1);
        ret = [ bm[1]*bx[2] , bx[1]*bm[2] ]; \\ == B(m)/B(m-1);
        x = gcd(ret[1], ret[2]);  /* Reduction */
        ret = [ret[1]/x, ret[2]/x];  /* Reduction */
    , /* else: */
        x = floor( (m+n)/2 );
        tm = Q(m, x);    \\ [U_{m,x}, V_{m,x}]
        tx = Q(x+1, n);  \\ [U_{x+1,n}, V_{m,n}]
        bm = B(m-1);     \\ [p_{m-1}, q_{m-1}]
        bx = B(x);       \\ [p_{x}, q_{x}]
        \\ ret == Q(m, x) + B(x) / B(m-1) * Q(x+1, n);
        ret = [ (bm[1]*bx[2]*tm[1]*tx[2] + bx[1]*bm[2]*tx[1]*tm[2])/10,
                (bm[1]*bx[2]*tm[2]*tx[2])/10 ];
```

```
        x = gcd(ret[1], ret[2]);  /* Reduction */
        ret = [ret[1]/x, ret[2]/x];  /* Reduction */
    );
    return( ret );
}
```

The reduction step can do good or bad, depending on the terms of the sum. When computing arctan(1/10) *without* the reduction, the intermediate quantities grow exponentially:

```
Q(0, 6)
        Q(0, 3)
                Q(0, 1)
                        Q(0, 0)
                        ^== [1, 10]
                        Q(1, 1)
                        ^== [-10, 3000]
                ^== [29900, 300000]
                Q(2, 3)
                        Q(2, 2)
                        ^== [3000, -500000]
                        Q(3, 3)
                        ^== [-500000, 70000000]
                ^== [-104250000000000000, 17500000000000000000]
        ^== [1569781275000000000000000000, 157500000000000000000000000000]
 [-snip-]
```

The square brackets are the quantities $[U_{m,n}, V_{m,n}]$. Such explosive growth will occur with all Taylor series unless the function argument equals one.

Performance

We compute the sum for arctan(1/10) up to the 5,000th term with the direct method, the rational binsplit and the integer binsplit with and without reduction. The timings for the computation are:

```
  A(k)=if(k<0,1, (-1)^(k)/((2*k+1)*10^(2*k+1)));  \\ for rational binsplit
  B(k)=if(k<0, [1,1], [(-1)^(k), ((2*k+1)*10^(2*k+1))] ); \\ for integer binsplit
  N=5000;
  sum(k=0,N,A(k)); \\ direct method:  69,385 ms.
  R(0,N); \\ rational binsplit: 2,532 ms.
  Q(0,N); \\ integer binsplit with gcd reduction: 4,152 ms.
  Q(0,N); \\ integer binsplit without gcd reduction: >8min, "forever"
```

Things look quite different when computing the sum $\sum_{k=0}^{50,000} (-1)^k/(2k+1)^2$. The intermediate quantities U and V have only small common factors, so it is better to omit the reduction step:

```
B(k)=if(k<0, [1,1], [(-1)^k, (2*k+1)^2] );
A(k)=if(k<0,1, (-1)^(k)/(2*k+1)^2);
  N=50000;
  sum(k=0,N,A(k)); \\ direct method:  32,396 ms.
  R(0,N); \\ rational binsplit:  6,826 ms.
  Q(0,N); \\ integer binsplit with gcd reduction:  27,485 ms.
  Q(0,N); \\ integer binsplit without gcd reduction:  6,251 ms.
```

With builtin routines for binsplit summation the advantage will be much more in favor than these figures suggest.

The reason why summation via binary splitting is better than the straightforward way is that its complexity is only $O(\log N\, M(N))$, where $M(N)$ is the complexity of one N-bit multiplication (see [56]). If an FFT based multiplication algorithm is used ($M(N) = N \log N$) the work is $O((\log N)^2\, N)$ This means that sums of linear but sufficient convergence are again candidates for high precision computations. The algorithm should be implemented in the 'depth first' manner presented, and *not* via the naive pairs, pairs of pairs, etc. (breadth first) way. The reasons are better locality and less memory consumption. The naive way needs most memory after the first pass, when pairs have been multiplied.

Extending prior computations

The ratio $r_{0,N-1}$ (that is, the sum of the first N terms) can be reused if one wants to evaluate the sum to a higher precision than before. To get twice the precision use

$$r_{0,2\,N-1} \quad = \quad r_{0,N-1} + a_{N-1} \cdot r_{N,2\,N-1} \tag{20.10}$$

(this is formula 20.8 with $m = 0, x = N-1, n = 2N-1$). With explicit rational arithmetic:

$$U_{0,2N-1} \quad = \quad q_{N-1}\, U_{0,N-1}\, V_{N,2N-1} + p_{N-1}\, U_{N,2N-1}\, V_{0,N-1} \tag{20.11a}$$

$$V_{0,2N-1} \quad = \quad q_{N-1}\, V_{0,N-1}\, V_{N,2N-1} \tag{20.11b}$$

Thereby with the appearance of some new computer that can multiply two length $2 \cdot N$ numbers[2] one only needs to combine the two ratios $r_{0,N-1}$ and $r_{N,2N-1}$ that had been precomputed by the last generation of computers. This costs only a few full-size multiplications on your new and expensive supercomputer (instead of several *hundreds* for the iterative schemes), which means that one can improve on prior computations at low cost.

If one wants to stare at zillions of decimal digits of the floating point expansion then one division is also needed which costs not more than 4 multiplications (see section 15.1 on page 399).

Computation of π: binary splitting versus AGM-type iterations

Using formula 20.1a and the binary splitting scheme for the computation of π can outperform the AGM-style iterations given in section 17.8 on page 444. The reason is that the memory access pattern is more favorable than with the iterations. When computing N digits of π the iterations compute proportional $\log_2(N)$ roots (and or inverses) to full precision. At the last phase of each root computation full-length multiplications have to be computed. These access all memory storing a few full-precision words. In contrast, the binary splitting involves full-precision multiplications only at the very last phase.

The drawback of the binary splitting scheme is that it may need significantly more memory than two full words. This may happen if the numerator and denominator grow fast which is more likely if no series so favorable as 20.1a can be used for the quantity to be computed.

20.2 The magic sumalt algorithm

The following convergence acceleration algorithm for alternating series is due to Cohen, Villegas and Zagier, see [37]. As remarked in the cited paper, the algorithm often gives meaningful results also for non-alternating and even divergent series.

Pseudo code to compute an estimate of $\sum_{k=0}^{\infty} x_k$ using the first n summands. The summands x_k have to be supplied in the array `x[0,1,...,n-1]`:

```
function sumalt(x[], n)
{
    d := (3+sqrt(8))^n
    d := (d+1/d)/2
    b := 1
    c := d
    s := 0
    for k:=0 to n-1
    {
        c := c - b
        s := s + c * x[k]
        b := b * (2*(n+k)*(n-k)) / ((2*k+1)*(k+1))
    }
    return s/d
}
```

[2] assuming the old model could multiply length-N numbers.

With alternating sums the accuracy of the estimate will be $(3+\sqrt{8})^{-n} \approx 5.82^{-n}$.

As an example let us write down the estimate for $4 \cdot \arctan(1)$ using the first 8 terms

$$\pi \approx 4 \cdot \left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} \right) = 3.017\ldots \tag{20.12}$$

The sumalt-massaged estimate is

$$\begin{aligned} \pi &\approx 4 \cdot \left(\frac{665856}{1} - \frac{665728}{3} + \frac{663040}{5} - \frac{641536}{7} + \right. \\ &\qquad \left. + \frac{557056}{9} - \frac{376832}{11} + \frac{163840}{13} - \frac{32768}{15} \right) / 665857 \\ &= 4 \cdot 3365266048/4284789795 = 3.141592665\ldots \end{aligned} \tag{20.13}$$

and already gives 7 correct digits of π.

n	sumalt(n)	sumalt(n)$-\pi$
1	2.666666666666666666	0.474925986923126571795
2	3.137254901960784313725	0.004337751629008924737
3	3.140740740740740740740	0.000851912849052497721
4	3.141635718412148221507	−0.000043064822354983044
5	3.141586546403673968348	0.000006107186119270114
6	3.141593344215659403660	−0.000006906258661651197
7	3.141592564937540122015	0.000000088652253116447
8	3.141592665224315864017	−0.000000011634522625555
9	3.141592652008811951619	0.000000001580981286843
10	3.141592653809731569318	−0.000000000219938330856
11	3.141592653558578755513	0.000000000031214482948
12	3.141592653594296338470	−0.000000000004503100007
13	3.141592653589134580517	0.000000000000658657944
14	3.141592653589890718625	−0.000000000000097480163
15	3.141592653589778664375	0.000000000000014574087
16	3.141592653589795436775	−0.000000000000002198312
17	3.141592653589792904285	0.000000000000000334177
18	3.141592653589793289614	−0.000000000000000051151
19	3.141592653589793230584	0.000000000000000007877
20	3.141592653589793239682	−0.000000000000000001220

Figure 20.1: Sumalt-estimates of $\pi = 4 \cdot \arctan(1)$ using $n = 1, 2, \ldots, 20$ terms.

The linear but impressive growth of the accuracy of successive sumalt estimates with n, the number of terms used, is illustrated in figure 20.1.

Therefore even slowly converging series like

$$\pi = 4 \cdot \sum_{k=0}^{\infty} \frac{(-1)^k}{2\,k+1} = 4 \cdot \arctan(1) \tag{20.14a}$$

$$C = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2\,k+1)^2} = 0.9159655941772190\ldots \tag{20.14b}$$

$$\log(2) = \sum_{k=0}^{\infty} \frac{(-1)^k}{k+1} = 0.6931471805599453\ldots \tag{20.14c}$$

$$\zeta(s) = \frac{1}{1-2^{1-s}} \sum_{k=1}^{\infty} \frac{(-1)^k}{k^s} \tag{20.14d}$$

can be used to compute estimates that are correct up to thousands of digits. The algorithm scales like n^2 if the series terms in the array `x[]` are small rational values and like $n^3 \cdot \log(n)$ if they are full precision (rational or float) values.

In fact, pari/gp has a built-in sumalt routine, we use it to compute the *Catalan constant*:

```
? default(realprecision,1000);
? sumalt(k=0,(-1)^k/(2*k+1)^2);  \\ takes    60 ms.
? default(realprecision,2000);
? sumalt(k=0,(-1)^k/(2*k+1)^2);  \\ takes   376 ms.
? default(realprecision,4000);
? sumalt(k=0,(-1)^k/(2*k+1)^2);  \\ takes 2,730 ms.
```

The time scales roughly with the third power of the precision used.

All values c_k and b_k occurring in the computation are integers. In fact, the b_k in the computation with n terms are the coefficients of the expanded n-th Chebyshev polynomial (of the first kind, see section 19.2 on page 471) with argument $1 + 2x$:

k	b_k	c_k
0	1	665857
1	128	665856
2	2688	665728
3	21504	663040
4	84480	641536
5	180224	557056
6	212992	376832
7	131072	163840
8	32768	32768

$$\begin{aligned} T_8(1+2x) &= 1 + 128x + 2688x^2 + 21504x^3 + 84480x^4 + \qquad (20.15a)\\ &\quad +180224x^5 + 212992x^6 + 131072x^7 + 32768x^8 = T_{16}(\sqrt{1+x}) \\ T_{16}(x) &= 1 - 128x^2 + 2688x^4 - 21504x^6 + 84480x^8 - \qquad (20.15b)\\ &\quad -180224x^{10} + 212992x^{12} - 131072x^{14} + 32768x^{16} \end{aligned}$$

Now observe that one has always $c_n = b_n = 2^{2n-1}$ in a length-n sumalt computation. Obviously, 'going backwards' avoids the computation of $(3 + \sqrt{8})^n$:

```
function sumalt(x[], n)
{
    b := 2**(2*n-1)
    c := b
    s := 0
    for k:=n-1 to 0 step -1
    {
        s := s + c * x[k]
        b := b * ((2*k+1)*(k+1)) / (2*(n+k)*(n-k))
        c := c + b
    }
    return s/c
}
```

The b_k and c_k occurring in a length-n sumalt computation can be given explicitly as

$$b_k = \frac{n}{n+k}\binom{n+k}{2k}2^{2k} \qquad (20.16a)$$

$$c_k = \sum_{i=k}^{n} \frac{n}{n+i}\binom{n+i}{2i}2^{2i} \qquad (20.16b)$$

To compute an estimate of $\sum_{k=0}^{\infty} x_k$ using the first n partial sums use the following pseudo code (the partial sums $p_k = \sum_{j=0}^{k} x_j$ are expected in `p[0,1,...,n-1]`):

```
function sumalt_partial(p[], n)
{
    d := (3+sqrt(8))^n
    d := (d+1/d)/2
    b := 1
    c := d
    s := 0
    for k:=0 to n-1
    {
        s := s + b * p[k]
        b := b * (2*(n+k)*(n-k)) / ((2*k+1)*(k+1))
    }
    return s/d
}
```

The backward variant is:

```
function sumalt_partial(p[], n)
{
    b := 2**(2*n-1)
    c := b
    s := 0
    for k:=n-1 to 0 step -1
    {
        s := s + b * p[k]
        b := b * ((2*k+1)*(k+1)) / (2*(n+k)*(n-k))
        c := c + b
    }
    return s/c
}
```

For series of already geometrical rate of convergence (where $|a_k/a_{k+1}| \approx e$) it is better to use

```
function sumalt_partial(p[], n, e)
{
    d := ( 2*e + 1 + 2*sqrt(e*(e+1)) )^n
    d := (d+1/d)/2
    b := 1
    c := d
    s := 0
    for k:=0 to n-1
    {
        s := s + b * p[k]
        b := b * (2*(n+k)*(n-k)) / ((2*k+1)*(k+1)) * e
    }
    return s/d
}
```

Convergence is improved from $\sim e^{-n}$ to $\sim (2e + 1 + 2\sqrt{e(e+1)})^{-n} \approx (4e + 2)^{-n}$. This algorithm specializes to the original one for $e = 1$.

C++ routines implementing the sumalt algorithm and the variant for partial sums are given in [hfloat: src/hf/sumalt.cc].

20.3 Continued fractions

A *continued fraction* is an expression of the form:

$$K(a,b) = a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cfrac{b_4}{a_4 + \cdots}}}} \tag{20.17}$$

Continued fractions are sometimes expressed in the following form:

$$K(a,b) = a_0 + \frac{b_1}{a_1+}\ \frac{b_2}{a_2+}\ \frac{b_3}{a_3+}\ \frac{b_4}{a_4+}\ \cdots \tag{20.18}$$

The a_k and b_k are called the k-th *partial numerators* and *denominators*.

For $k > 0$ let P_k/Q_k be the value of the above fraction if b_{k+1} is set to zero (that is, the continued fraction terminates at index k). The ratio is called the k-th *convergent* of the continued fraction:

$$\frac{P_k}{Q_k} \quad = \quad a_0 + \frac{b_1}{a_1+}\,\frac{b_2}{a_2+}\,\frac{b_3}{a_3+}\,\frac{b_4}{a_4+}\,\ldots\,\frac{b_{k-1}}{a_{k-1}+}\,\frac{b_k}{a_k} \tag{20.19}$$

We note that multiplication of a_i, b_i, b_{i+1} by some nonzero value does not change the value of the continued fraction. Thereby

$$a_0 + \frac{b_1}{a_1+}\,\frac{b_2}{a_2+}\,\frac{b_3}{a_3+}\,\frac{b_4}{a_4+}\,\ldots \quad = \quad a_0 + \frac{c_1\,b_1}{c_1\,a_1+}\,\frac{c_2\,c_1\,b_2}{c_2\,a_2+}\,\frac{c_3\,c_2\,b_3}{c_3\,a_3+}\,\frac{c_4\,c_3\,b_4}{c_4\,a_4+}\,\ldots \tag{20.20}$$

where all c_i are arbitrary nonzero constants.

Simple continued fractions

Continued fractions where $b_k = 1\ \forall k$ are called *simple continued fractions*. Rational numbers have terminating continued fractions $[a_0, \ldots, a_n]$. Note that

$$[a_0, \ldots, a_{n-1}, a_n] \quad = \quad [a_0, \ldots, a_{n-1}, a_n - 1, 1] \quad \text{if } a_n > 1 \tag{20.21a}$$

$$[a_0, \ldots, a_{n-1}, 1] \quad = \quad [a_0, \ldots, a_{n-1} + 1] \qquad \text{if} \quad a_n = 1 \tag{20.21b}$$

That is, the expression of a rational number as simple continued fraction is not unique.

Solutions of the quadratic equation $\alpha\,x^2 + \beta\,x + \gamma = 0$ that are not rational ($\Delta := \beta^2 - 4\,\alpha\,\gamma$ not a square) have simple continued fractions that are eventually periodic. For example:

```
? contfrac(sqrt(5))
  [2, 4, 4, 4, 4, 4, ...]
? contfrac(2+sqrt(3))
  [3, 1,2, 1,2, 1,2, ...]
? contfrac(sqrt(19))
  [4, 2,1,3,1,2,8, 2,1,3,1,2,8, 2,1,3,1,2,8, ...]
```

For the k-th convergent P_k/Q_k (in lowest terms) of the simple continued fraction expansion of some number x the following 'best approximation' statement is true: If p/q is any better rational approximation to x (that is, $\left|\frac{p}{q} - x\right| < \left|\frac{P_k}{Q_k} - x\right|$), then one must have $q > Q_k$.

Determinantal expressions

The so-called *determinant formula* for the numerators and denominators of successive convergents is

$$\det \begin{bmatrix} P_k & P_{k-1} \\ Q_k & Q_{k-1} \end{bmatrix} \quad = \quad P_k\,Q_{k-1} - P_{k-1}\,Q_k \;=\; (-1)^{k-1} \prod_{j=1}^{k} b_j \tag{20.22}$$

For simple continued fractions we have $b_j = 1$ so the product in the numerator equals one. The relation can also be written as

$$\frac{P_k}{Q_k} - \frac{P_{k-1}}{Q_{k-1}} \quad = \quad \frac{(-1)^{k-1} \prod_{j=1}^{k} b_j}{Q_{k-1}\,Q_k} \tag{20.23}$$

Further, by inserting $P_{k-1} = (P_k - b_k\,P_{k-2})/a_k$ (relation 20.27a) and the equivalent expression for Q_{k-1} into relation 20.22, one obtains

$$\det \begin{bmatrix} P_k & P_{k-2} \\ Q_k & Q_{k-2} \end{bmatrix} \quad = \quad P_k\,Q_{k-2} - P_{k-2}\,Q_k \;=\; (-1)^{k}\,a_k \prod_{j=1}^{k-1} b_j \tag{20.24}$$

Equivalently,

$$\frac{P_k}{Q_k} - \frac{P_{k-2}}{Q_{k-2}} \quad = \quad \frac{(-1)^k\, a_k \prod_{j=1}^{k-1} b_j}{Q_{k-2}\, Q_k} \tag{20.25}$$

This relation tells us (provided all a_j and b_j are positive) that the sequence of even convergents is increasing and the sequence of odd convergents is decreasing. As both converge to a common limit we have $P_o/Q_o \geq P_e/Q_e$ for all even e and odd o. Equality can occur only for terminating continued fractions.

For the simple continued fraction of x one has

$$\left| x - \frac{P_n}{Q_n} \right| \quad \leq \quad \frac{1}{Q_n\, Q_{n-1}} \quad < \quad \frac{1}{Q_n^2} \tag{20.26}$$

and equality can only occur with terminating continued fractions.

20.3.1 Computation of the convergents

Computation of the sequence of convergents is easily achieved via the following recursion relations:

$$P_k \quad = \quad a_k\, P_{k-1} + b_k\, P_{k-2} \tag{20.27a}$$
$$Q_k \quad = \quad a_k\, Q_{k-1} + b_k\, Q_{k-2} \tag{20.27b}$$

Set $\frac{P_{-1}}{Q_{-1}} := \frac{1}{0}$ and $\frac{P_0}{Q_0} := \frac{a_0}{1}$ to initialize. The following is a procedure that computes the sequences $P_k, Q_k \quad k = -1 \ldots n$ of a continued fraction:

```
procedure ratios_from_cf(a[0..n], b[0..n], n, P[-1..n], Q[-1..n])
{
    P[-1] := 1
    Q[-1] := 0

    P[0] := a[0]
    Q[0] := 1

    for k:=1 to n
    {
        P[k] := a[k] * P[k-1] + b[k] * P[k-2]
        Q[k] := a[k] * Q[k-1] + b[k] * Q[k-2]
    }
}
```

If only the last ratio is of interest, the 'backward' variant of the algorithm can be used. The version that computes the numerical value `x` from the first `n` terms of a simple continued fraction is:

```
function ratio_from_cf(a[0..n-1], n)
{
    x := a[n-1]
    for k:=n-2 to 0 step -1
    {
        x := 1/x + a[k]
    }

    return x
}
```

Using rational arithmetic and a general (non-simple) continued fraction, the algorithm becomes:

```
function ratio_from_cf(a[0..n-1], b[0..n-1]], n)
{
    P := a[n-1]
    Q := b[n-1]
    for k:=n-2 to 0 step -1
    {
        {P, Q} := {a[k]*P+b[k]*Q, P}  // x := b[k] / x + a[k]
    }

    return P/Q
}
```

Given a numerical quantity one can compute the sequence a_k of its simple continued fraction by the following algorithm:

```
procedure number_to_scf(x, n, a[0..n-1])
{
    for k:=0 to n-1
    {
        xi := floor(x)
        a[k] := xi
        x := 1 / (x-xi)
    }
}
```

Here `n` is the number of requested terms a_k. Obviously some check has to be inserted in order to avoid possible division be zero (and indicate a terminating continued fraction as will occur for rational x). If in the process one keeps track of the exact rational values of the simple continued fraction convergents (using the recursion relations) then the algorithm can be used to produce a 'best possible' rational approximation where the denominator does not exceed a certain specified size. Alternatively one can stop as soon as the convergent is within some specified bound.

A comparison of methods for the computation of continued fractions see [31].

20.3.2 Implementation in pari/gp

Converting a number to a simple continued fraction can be done with pari/gp's builtin function `contfrac()`. The final convergent can be computed with `contfracpnqn()`:

```
? default(realprecision,23)
    realprecision = 28 significant digits (23 digits displayed)
? Pi
    3.1415926535897932384626
? cf=contfrac(Pi)
    [3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2, 1, 84, 2, 1, 1, 15, 3]
? ?contfracpnqn
    contfracpnqn(x): [p_n,p_{n-1}; q_n,q_{n-1}] corresponding to the continued fraction x.
? m=contfracpnqn(cf)
    [428224593349304 139755218526789]
    [136308121570117 44485467702853]
? 1.0*m[1,1]/m[2,1]
    3.1415926535897932384626
```

The number of terms of the continued fraction depends on the precision used.

The computation of the m-th convergent of a continued fraction given as two vectors `a[]` and `b[]` can be implemented as (backward variant):

```
cfab2r(a,b, m=-2)=
{
    local(n, r);
    n = length(a);
    if ( m>-2, m = min(n, m) ); \\ default: m=n
    if ( m>=n,  m=n-1 );
    if ( m<0, return( 0 ) ); \\ infinity
    r = 0;
    m += 1;
    forstep (k=m, 2, -1,  r = b[k]/(a[k]+r); );
    r += a[1];  \\ b[1] unused
    return( r );
}
```

Alternatively, one can use the recursions (relations 20.27a, and 20.27b):

```
cfab2pq(a,b,m=-2)=
{
    local(n, p, p1, p2, q, q1, q2, i);
    n = length(a)-1;
    if ( m>-2, m = min(n, m) ); \\ default: m=n
    if ( m<0, return( [1, 0] ) ); \\ infinity
    p1 = 1;
    q1 = 0;
    p = a[1];
```

```
    q = 1;  \\ b[1] unused
    for (k=1, m,
        i = k+1;
        p2 = p1;  p1 = p;
        q2 = q1;  q1 = q;
        p = a[i]*p1 + b[i]*p2;
        q = a[i]*q1 + b[i]*q2;
    );
    return( [p,q] );
}
```

Here we do not store all pairs P_n, Q_n but only return the final pair P_m, Q_m.

```
    default(realprecision, 55);  \\ use enough precision
    default(format, "g.11"); \\ print with moderate precision
    default(echo, 0);
    \r contfrac.inc.gp  \\ functions cfab2pq and cfab2r

    x=4.0/Pi
    n=15
    /* set up the continued fraction: */
    a=vector(n, j, 2);  a[1]=1;
    b=vector(n, j, (2*j-3)^2);

    /* print convergents and their error: */
    { for(k=0, n-1,
        t=cfab2pq(a,b, k);
        p=t[1]; q=t[2];
        print1(k, ":  ",p, " / ", q);
        print1("\n  d=", x-p/q);
        print();
      ); }

    quit; /* ------ end of script -------- */

    /* ------ start output: ------ */
    15  /* =n */
    [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]  /* =a */
    [1, 1, 9, 25, 49, 81, 121, 169, 225, 289, 361, 441, 529, 625, 729]  /* =b */
    1.2732395447 /* = 4/Pi */

    0:  1 / 1
      d=0.27323954473
    1:  3 / 2
      d=-0.22676045526
    2:  15 / 13
      d=0.11939339088
    3:  105 / 76          /* =p3/q3 */
      d=-0.10833940263    /* =p3/q3-4/Pi */
    4:  945 / 789
      d=0.075520913556
    5:  10395 / 7734
      d=-0.070825622061
    [--snip--]
    13:  213458046676875 / 163842638377950
      d=-0.029583998575
    14:  6190283353629375 / 4964894559637425
      d=0.026428906710
```

Figure 20.2: A pari/gp script demonstrating the function `cfab2pq()` that computes the convergents of a continued fraction (top) and its output (bottom, comments added). Convergence is rather slow.

We use our routines to compute the convergents of the continued fraction for $4/\pi$ given 1658 by Brouncker:

$$\frac{4}{\pi} \;=\; 1+\cfrac{1^2}{2+\cfrac{3^2}{2+\cfrac{5^2}{2+\cfrac{7^2}{2+\cdots}}}} \;=\; a_0+\cfrac{b_1}{a_1+\cfrac{b_2}{a_2+\cfrac{b_3}{a_3+\cfrac{b_4}{a_4+\cdots}}}} \tag{20.28}$$

Setting up the vectors containing the a_k and b_k and checking the convergents, using a script, is shown in figure 20.2.

Dividing relation 20.27a by relation 20.27b

$$\frac{P_k}{Q_k} = \frac{a_k\,P_{k-1} + b_k\,P_{k-2}}{a_k\,Q_{k-1} + b_k\,Q_{k-2}} \tag{20.29a}$$

$$= \frac{\frac{a_k}{b_k}\,P_{k-1} + P_{k-2}}{\frac{a_k}{b_k}\,Q_{k-1} + Q_{k-2}} \tag{20.29b}$$

we see that one can use certain routines designed for simple continued fractions by using the ratios $\frac{a_k}{b_k}$.

20.3.3 Subsequences of convergents

Sometimes the terms a_k, b_k of the continued fraction are given in the form "$a_k = u(k)$ *if* k *even,* $a_k = v(k)$ *else*" (and b_k equivalently). Then one might want to compute the $x = K(a, b)$ in a 'stride two' manner as

$$P_k = A_k\,P_{k-2} + B_k\,P_{k-4} \tag{20.30a}$$

$$Q_k = A_k\,Q_{k-2} + B_k\,Q_{k-4} \tag{20.30b}$$

in order to regularize the involved expressions. We write the recurrence relation three times

$$P_k = a_k\,P_{k-1} + b_k\,P_{k-2} \tag{20.31a}$$

$$P_{k-1} = a_{k-1}\,P_{k-2} + b_{k-1}\,P_{k-3} \tag{20.31b}$$

$$P_{k-2} = a_{k-2}\,P_{k-3} + b_{k-2}\,P_{k-4} \tag{20.31c}$$

and eliminate the terms P_{k-1} and P_{k-3}. The resulting relation has the form of 20.30a and we can read off A_k and B_k. One obtains

$$A_k = \frac{a_k\,b_{k-1} + b_k\,a_{k-2} + a_k\,a_{k-1}\,a_{k-2}}{a_{k-2}} = \frac{a_k\,b_{k-1}}{a_{k-2}} + b_k + a_k\,a_{k-1} \tag{20.32a}$$

$$B_k = \frac{-a_k\,b_{k-1}\,b_{k-2}}{a_{k-2}} \tag{20.32b}$$

The stride three version

$$P_k = A_k\,P_{k-3} + B_k\,P_{k-6} \tag{20.33a}$$

$$Q_k = A_k\,Q_{k-3} + B_k\,Q_{k-6} \tag{20.33b}$$

leads to the expressions (writing a_n for a_{k-n} to reduce line width):

$$A_k = \frac{a_0b_1b_3 + b_0a_2b_3 + b_0b_2a_4 + a_0a_1a_2b_3 + a_0a_1b_2a_4 + a_0b_1a_3a_4 + b_0a_2a_3a_4 + a_0a_1a_2a_3a_4}{b_3 + a_3a_4} \tag{20.34a}$$

$$B_k = \frac{b_0b_2b_3b_4 + a_0a_1b_2b_3b_4}{b_3 + a_3a_4} \tag{20.34b}$$

When setting $a_k = \alpha$, $b_k = \beta$ the expressions for A_k and B_k simplify to the coefficients in relations 19.13c on page 467 (stride two) and 19.13d (stride three) for recurrences.

20.3.4 Fast computation as infinite product

For the evaluation of a large number of terms rewrite relation 20.27a on page 490 as

$$[P_{k+1},\ P_k]^T = \begin{bmatrix} 0 & a_k \\ 1 & b_k \end{bmatrix} [P_k,\ P_{k-1}]^T \tag{20.35}$$

The very same is obviously true for Q_k. Express the recursion as an infinite product:

$$[P_{k+1},\ P_k]^T \quad = \quad M(k)\ [P_0,\ P_{-1}]^T \tag{20.36a}$$

where

$$M(k) \quad = \quad \prod_{j=1}^{k} \begin{bmatrix} 0 & a_k \\ 1 & b_k \end{bmatrix} \tag{20.36b}$$

and $P_{-1} = 0$, $P_0 = a_0$. Then use the binary splitting algorithm (section 20.1 on page 481) for the computation of the matrix product. Use $Q_{-1} = 0$ and $Q_0 = 1$ as starting point for the denominators.

20.3.5 Miscellaneous relations for continued fractions

Relation to alternating series

Using relation 20.23 on page 489 it is possible to rewrite a continued fraction $x = K(a, b)$ with positive a_k, b_k as an alternating series

$$\begin{aligned} x \quad &= \quad a_0 + \sum_{k=1}^{\infty} (-1)^{k+1}\, s_k \\ &= \quad a_0 + \frac{b_1}{Q_0\, Q_1} - \frac{b_1\, b_2}{Q_1\, Q_2} + \frac{b_1\, b_2\, b_3}{Q_2\, Q_3} \pm \ldots + (-1)^{k+1} \frac{\prod_{i=1}^{k} b_i}{Q_k\, Q_{k+1}} \pm \ldots \end{aligned} \tag{20.37}$$

Thereby the algorithm for the accelerated summation of alternating series from section 20.2 can be applied to compute x.

Continued fractions from infinite products

A continued fraction for the product

$$P \quad := \quad \prod_{k=0}^{\infty} (1 + Y_k) \tag{20.38a}$$

in terms of $a = [a_0,\ a_1,\ \ldots]$ and $b = [b_0,\ b_1,\ \ldots]$ is

$$a \quad = \quad [1, \quad 1, \quad +Y_1 + (1 + Y_1) \cdot Y_0, \quad +Y_2 + (1 + Y_2) \cdot Y_1, \quad +Y_3 + (1 + Y_3) \cdot Y_2,\ \ldots\,] \tag{20.38b}$$

$$b \quad = \quad [1, \quad +Y_0, \quad -1 \cdot Y_1 \cdot (1 + Y_0), \quad -Y_0 \cdot Y_2 \cdot (1 + Y_1), \quad -Y_1 \cdot Y_3 \cdot (1 + Y_2),\ \ldots\,] \tag{20.38c}$$

Computation of individual a_k and b_k given a vector $y = [\ldots, Y_k, \ldots]$ can be implemented as:

```
cfproda(yv, n)=
{
    local( y2, y3, d );
    if ( n<=1, return(1) );
    y3 = yv[n];
    y2 = yv[n-1];
    return( (y2+y3*(1+y2)) );
}

cfprodb(yv, n)=
{
    local( y1, y2, y3 );
    if (0==n, return(1) );  \\ unused
    if (1==n, return(+yv[1+0]) );
    y3 = yv[n];
```

```
Y(k) = eval(Str("Y" k))  \\ return symbol Yk
yprod(n)= if (n<=0, 1, prod(j=0, n-1, (1+Y(j))))
n=3
pr = yprod(n)
    ((Y2 + 1)*Y1 + (Y2 + 1))*Y0 + ((Y2 + 1)*Y1 + (Y2 + 1))
yv = vector(n, j, Y(j-1) )
    [Y0, Y1, Y2]
t = cfprod(yv);
a=t[1]
    [1, 1, (Y1 + 1)*Y0 + Y1, (Y2 + 1)*Y1 + Y2]
b=t[2]
    [1, Y0, -Y1*Y0 - Y1, (-Y2*Y1 - Y2)*Y0]
{ for(k=0, n,
    t=cfab2pq(a,b, k);
    p=t[1]; q=t[2];
    print1(k, ":  (",p, ") / (", q,")");
    yp = yprod(k);
    print1("\n   == ", simplify(p/q));
    print();
  ); }
    0:  (1) / (1)
       == 1
    1:  (Y0 + 1) / (1)  \\  (p1) / (q1)
       == Y0 + 1        \\  == yprod(1)
    2:  ((Y1 + 1)*Y0^2 + (Y1 + 1)*Y0) / (Y0)  \\  (p2) / (q2)
       == (Y1 + 1)*Y0 + (Y1 + 1)              \\ == yprod(2) == (1+Y0)*(1+Y1)
    3:  (((Y2 + 1)*Y1^2 + (Y2 + 1)*Y1)*Y0^2 + ((Y2 + 1)*Y1^2 + (Y2 + 1)*Y1)*Y0) / (Y1*Y0)
       == ((Y2 + 1)*Y1 + (Y2 + 1))*Y0 + ((Y2 + 1)*Y1 + (Y2 + 1))
```

Figure 20.3: Verification of relations 20.38b and 20.38c on the facing page using pari/gp.

```
    y2 = yv[n-1];
    y1 = if ( n==2, 1, yv[n-2] );
    return( -y1*y3*(1+y2) );
}
```

The routine `cfprod()` generates the vectors a and b with $n+1$ terms where n is the length of y:

```
cfprod(yv)=
{
    local(n, a, b);
    n = length(yv);
    n += 1;  \\ n+1 terms in continued fraction
    a = vector(n);
    b = vector(n);
    for (k=0, n-1,
        a[k+1] = cfproda(yv, k);
        b[k+1] = cfprodb(yv, k);
    );
    return( [a, b] );
}
```

Relations 20.38b and 20.38c can be verified using pari/gp as shown in figure 20.3.

An expression for a sum of products

Define Z_n as

$$Z_n \;:=\; z_1 + z_1\,z_2 + z_1\,z_2\,z_3 + \ldots \;=\; \sum_{k=1}^{n}\prod_{i=1}^{k} z_i \tag{20.39}$$

Then Z_∞ has the continued fraction

$$Z_\infty = \cfrac{z_1}{1 - \cfrac{z_2}{1 + z_2 - \cfrac{z_3}{1 + z_3 - \cfrac{z_4}{1 + z_4 - \cfrac{z_5}{1 + z_5 - \cdots}}}}} \tag{20.40}$$

That is, $Z = K(a, b)$ where

$$a = [0,\ 1,\ z_2 + 1,\ z_3 + 1,\ z_4 + 1,\ z_5 + 1,\ z_6 + 1,\ \ldots] \tag{20.41a}$$
$$b = [1,\ z_1,\ -z_2,\quad -z_3,\quad -z_4,\quad -z_5,\quad -z_6,\quad \ldots] \tag{20.41b}$$

For the n-th convergent P_n/Q_n one has $Q_n = 1$ and $P_n = Z_n$.

For further information on continued fractions see [Olds] and [ConcMath] for an easy introduction. In depth information can be found in [LoWaa].

20.4 Hypergeometric functions

The *hypergeometric function* $F\left(\begin{matrix}a,\, b\\ c\end{matrix}\middle|\, z\right)$ can be defined as

$$F\left(\begin{matrix}a,\, b\\ c\end{matrix}\middle|\, z\right) := \sum_{k=0}^{\infty} \frac{a^{\overline{k}}\, b^{\overline{k}}}{c^{\overline{k}}}\, \frac{z^k}{k!} \tag{20.42}$$

where $z^{\overline{k}} := z\,(z+1)\,(z+2)\,\ldots\,(z+k-1)$ is the *rising factorial power* ($z^{\overline{0}} := 1$). Some sources use the so-called *Pochhammer symbol* $(x)_k$ which is the same: $(x)_k = x^{\overline{k}}$. We'll stick to the factorial notation.

z is the *argument* of the function, a, b and c are the *parameters*. Parameters in the upper and lower row are called upper and lower parameters, respectively.

Note the $k! = 1^{\overline{k}}$ in the denominator of relation 20.42. You might want to have an additional 1 in the lower row in mind:

$$F\left(\begin{matrix}2,\, 2\\ 1\end{matrix}\middle|\, z\right) = \text{“}F\left(\begin{matrix}2,\, 2\\ 1,\, 1\end{matrix}\middle|\, z\right)\text{”} \tag{20.43}$$

The expression is a sum of perfect squares if z is a square.

Obviously,

$$F\left(\begin{matrix}a,\, b\\ c\end{matrix}\middle|\, z\right) = 1 + \frac{a}{1}\,\frac{b}{c}\,z\left(1 + \frac{a+1}{2}\,\frac{b+1}{c+1}\,z\left(1 + \frac{a+2}{3}\,\frac{b+2}{c+2}\,z\,(1+\ldots)\right)\right) \tag{20.44}$$

so by formula 20.44 hypergeometric functions with rational arguments can be computed with the binary splitting method described in section 20.1.

Hypergeometric functions can have any number of parameters:

$$F\left(\begin{matrix}a_1,\, \ldots,\, a_m\\ b_1,\, \ldots,\, b_n\end{matrix}\middle|\, z\right) = \sum_{k=0}^{\infty} \frac{a_1^{\overline{k}} \ldots a_m^{\overline{k}}}{b_1^{\overline{k}} \ldots b_n^{\overline{k}}}\, \frac{z^k}{k!} \tag{20.45}$$

These are sometimes called *generalized hypergeometric functions*. The number of upper and lower parameters are often emphasized as subscripts left and right to the symbol F. For example, ${}_mF_n$ for the hypergeometric function in the last relation.

The functions $F\left(\begin{matrix}a\\b\end{matrix}\middle|\, z\right)$ (of type ${}_1F_1$) are sometimes written as $M(a,b,z)$ or $\Phi(a;b;z)$. *Kummer's function* $U(a,b,z)$ (or $\Psi(a;b;z)$) is related to hypergeometric functions of type ${}_2F_0$:

$$U(a,b,z) \quad = \quad z^{-a}\, F\left(\begin{matrix}a,\, 1+a-b\\ \end{matrix}\middle|\, -1/z\right) \tag{20.46}$$

Note that series ${}_2F_0$ are not convergent. Still, they can be used as asymptotic series for large values of z.

The so-called *Whittaker functions* are related to hypergeometric functions as follows:

$$M_{a,b}(z) \quad = \quad e^{-z/2}\, z^{b+1/2}\, F\left(\begin{matrix}\frac{1}{2}+b-a\\ 1+2b\end{matrix}\middle|\, z\right) \tag{20.47a}$$

$$W_{a,b}(z) \quad = \quad e^{-z/2}\, z^{b+1/2}\, U\left(\frac{1}{2}+b-a,\, 1+2b,\, z\right) \tag{20.47b}$$

$$\quad = \quad e^{-z/2}\, z^{a}\, F\left(\begin{matrix}\frac{1}{2}+b-a,\, \frac{1}{2}-b-a\\ \end{matrix}\middle|\, -1/z\right) \tag{20.47c}$$

Negative integer parameters in the upper row lead to polynomials:

$$F\left(\begin{matrix}-3,\, 3\\ 1\end{matrix}\middle|\, z\right) \quad = \quad 1-9\,z+18\,z^2-10\,z^3 \tag{20.48}$$

The lower parameter must not be zero or a negative integer unless there is a negative upper parameter with smaller absolute value.

Sometimes one finds the notational convention to omit an argument $z=1$:

$$F\left(\begin{matrix}a_1,\, \ldots,\, a_m\\ b_1,\, \ldots,\, b_n\end{matrix}\right) \quad := \quad F\left(\begin{matrix}a_1,\, \ldots,\, a_m\\ b_1,\, \ldots,\, b_n\end{matrix}\middle|\, 1\right) \tag{20.49}$$

In what follows the argument is never omitted.

Derivative and differential equation

Using

$$\frac{d}{dz^n}\, F\left(\begin{matrix}a,\, b\,\ldots\\ c,\,\ldots\end{matrix}\middle|\, z\right) \quad = \quad \frac{a^{\overline{n}}\, b^{\overline{n}}\,\ldots}{c^{\overline{n}}\,\ldots}\, F\left(\begin{matrix}a+n,\, b+n\,\ldots\\ c+n,\,\ldots\end{matrix}\middle|\, z\right) \tag{20.50}$$

one can verify that $f(z)=F\left(\begin{matrix}a,b\\c\end{matrix}\middle|\, z\right)$ is a solution of the differential equation

$$z\,(1-z)\,\frac{d^2\,f}{dz^2}+[c-(1+a+b)\,z]\,\frac{d\,f}{dz}-a\,b\,f \quad = \quad 0 \tag{20.51}$$

A general form of the differential equation satisfied by $F\left(\begin{matrix}a,b,c,\ldots\\u,v,w,\ldots\end{matrix}\middle|\, z\right)$ is

$$\vartheta\,(\vartheta+u-1)\,(\vartheta+v-1)\,(\vartheta+w-1)\,\ldots\,f(z) \quad = \quad z\,(\vartheta+a)\,(\vartheta+b)\,(\vartheta+c)\,\ldots\,f(z) \tag{20.52}$$

where ϑ is the operator $z\,\frac{d}{dz}$. The leftmost ϑ in the equation takes care of the hidden lower parameter 1: $\vartheta=(\vartheta+1-1)$. See [ConcMath] for a beautiful derivation.

Evaluations for fixed z

A closed form (in terms of the gamma function) evaluation at $z = 1$ can be given for ${}_2F_1$:

$$F\left(\begin{matrix}a,\, b\\ c\end{matrix}\middle|\, 1\right) \;=\; \frac{\Gamma(c)\,\Gamma(c-a-b)}{\Gamma(c-a)\,\Gamma(c-b)} \qquad \text{if} \quad \mathrm{Re}(c-a-b) > 0 \quad \text{or } b \in \mathbb{N}, b < 0 \tag{20.53}$$

For $z = -1$ there is an evaluation due to Kummer:

$$F\left(\begin{matrix}a,\, b\\ 1+a-b\end{matrix}\middle|\, -1\right) \;=\; \frac{\Gamma(1-a+b)\,\Gamma(1+a/2)}{\Gamma(1+a)\,\Gamma(1+a/2-b)} \tag{20.54a}$$

$$\;=\; 2^{-a}\,\pi\,\frac{\Gamma(1-a+b)}{\Gamma(1/2+a/2)\,\Gamma(1+a/2-b)} \tag{20.54b}$$

Several evaluations at $z = \frac{1}{2}$ are given in [HaMaFu], we just give one:

$$F\left(\begin{matrix}a,\, b\\ \frac{1}{2}+\frac{1}{2}a+\frac{1}{2}b\end{matrix}\middle|\, \frac{1}{2}\right) \;=\; \sqrt{\pi}\,\frac{\Gamma(\frac{1}{2}+\frac{1}{2}a+\frac{1}{2}b)}{\Gamma(\frac{1}{2}+\frac{1}{2}a)\,\Gamma(\frac{1}{2}+\frac{1}{2}b)} \tag{20.55}$$

For further information see (chapter 15 of) [HaMaFu], [MathWorld] and [83]. Various evaluations of $F\left(\begin{smallmatrix}-an,\, bn+b_1\\ cn+c_1\end{smallmatrix}\middle|\, z\right)$ for integer a, b, c and n, $1 \le a \le 2$, $-4 \le b \le 4$ and $-4 \le c \le 4$ can be found in [43].

Extraction of even and odd part

Let $E[f(z)] = (f(z)+f(z))/2$ (the even powers of the series of $f(z)$), and $O[f(z)] = (f(z)-f(-z))/2$ (the odd powers). We express the even and odd parts of a hypergeometric series as hypergeometric functions:

$$E\left[F\left(\begin{matrix}a,\, b\\ c\end{matrix}\middle|\, z\right)\right] \;=\; F\left(\begin{matrix}\frac{a}{2},\, \frac{a+1}{2},\, \frac{b}{2},\, \frac{b+1}{2}\\ \frac{c}{2},\, \frac{c+1}{2},\, \frac{1}{2}\end{matrix}\middle|\, z^2\right) \tag{20.56a}$$

$$O\left[F\left(\begin{matrix}a,\, b\\ c\end{matrix}\middle|\, z\right)\right] \;=\; \frac{a\,b}{c}\,z\,F\left(\begin{matrix}\frac{a+1}{2},\, \frac{a+2}{2},\, \frac{b+1}{2},\, \frac{b+2}{2}\\ \frac{c+1}{2},\, \frac{c+2}{2},\, \frac{3}{2}\end{matrix}\middle|\, z^2\right) \tag{20.56b}$$

The lowers parameters $1/2$ and $3/2 = (1+2)/2$ are due to the hidden lower parameter 1. The general case for

$$H(z) \;:=\; F\left(\begin{matrix}a_1,\, \ldots,\, a_m\\ b_1,\, \ldots,\, b_n\end{matrix}\middle|\, z\right) \tag{20.57a}$$

is

$$E\left[H(z)\right] \;=\; F\left(\begin{matrix}\frac{a_1}{2},\, \frac{a_1+1}{2},\, \ldots,\, \frac{a_m}{2},\, \frac{a_m+1}{2}\\ \frac{b_1}{2},\, \frac{b_1+1}{2},\, \ldots,\, \frac{b_n}{2},\, \frac{b_n+1}{2},\, \frac{1}{2}\end{matrix}\middle|\, X\,z^2\right) \tag{20.57b}$$

$$O\left[H(z)\right] \;=\; \frac{a_1\,\cdots\,a_m}{b_1\,\cdots\,b_n}\,z\,F\left(\begin{matrix}\frac{a_1+1}{2},\, \frac{a_1+2}{2},\, \ldots,\, \frac{a_m+1}{2},\, \frac{a_m+2}{2}\\ \frac{b_1+1}{2},\, \frac{b_1+2}{2},\, \ldots,\, \frac{b_n+1}{2},\, \frac{b_n+2}{2},\, \frac{3}{2}\end{matrix}\middle|\, X\,z^2\right) \tag{20.57c}$$

where $X = 4^{m-n-1}$. For example,

$$E\left[F\left(\begin{matrix}1\\ 2\end{matrix}\middle|\, z\right)\right] \;=\; F\left(\begin{matrix}\frac{1}{2},\, 1\\ 1,\, \frac{3}{2},\, \frac{1}{2}\end{matrix}\middle|\, \frac{z^2}{4}\right) \;=\; F\left(\begin{matrix}\\ \frac{3}{2}\end{matrix}\middle|\, \frac{z^2}{4}\right) \;=\; \frac{\sinh z}{z} \tag{20.58}$$

We indicate a further generalization by the extraction of all terms of $H(z)$ where the exponent of z is divisible by 3:

$$\frac{H(z)+H(\omega\,z)+H(\omega^2\,z)}{3} \;=\; F\left(\begin{matrix}\frac{a_1}{3},\,\frac{a_1+1}{3},\,\frac{a_1+2}{3}\,\ldots,\,\frac{a_m}{3},\,\frac{a_m+1}{3},\,\frac{a_m+2}{3}\\ \frac{b_1}{3},\,\frac{b_1+1}{3},\,\frac{b_1+2}{3},\,\ldots,\,\frac{b_n}{3},\,\frac{b_n+1}{3},\,\frac{b_n+2}{3},\,\frac{1}{3},\,\frac{2}{3}\end{matrix}\,\middle|\,X\,z^3\right) \tag{20.59}$$

where $\omega=\exp(2\,i\,\pi/3)$ and $X=27^{m-n-1}$. For example, with $H(z)=\exp(z)=F\left(\,\middle|\,z\right)$ we obtain

$$F\left(\begin{matrix}\\ \frac{1}{3},\,\frac{2}{3}\end{matrix}\,\middle|\,\frac{z^3}{27}\right) \;=\; \sum_{k=0}^{\infty}\frac{z^{3k}}{(3k)!} \tag{20.60}$$

Define the power series $C_0(z)$, $C_1(z)$, and $C_2(z)$ by

$$C_s \;=\; \sum_{k=0}^{\infty}\frac{z^{3k+s}}{(3k+s)!} \tag{20.61a}$$

then (omitting arguments)

$$\det\begin{bmatrix}C_0 & C_1 & C_2\\ C_2 & C_0 & C_1\\ C_1 & C_2 & C_0\end{bmatrix} \;=\; C_0^3+C_1^3+C_2^3-3\,C_0\,C_1\,C_2 \;=\; 1 \tag{20.61b}$$

which is a three power series analogy to the relation $\cosh^2-\sinh^2=1$.

20.4.1 Transformations

As obvious from the definition, parameters in the upper row can be swapped (capitalized symbols for readability):

$$F\left(\begin{matrix}A,\,B,\,c\\ e,\,f,\,g\end{matrix}\,\middle|\,z\right) \;=\; F\left(\begin{matrix}B,\,A,\,c\\ e,\,f,\,g\end{matrix}\,\middle|\,z\right) \tag{20.62}$$

The same is true for the lower row. Usually one writes the parameters in ascending order. Identical elements in the lower and upper row can be canceled:

$$F\left(\begin{matrix}a,\,b,\,C\\ e,\,f,\,C\end{matrix}\,\middle|\,z\right) \;=\; F\left(\begin{matrix}a,\,b\\ e,\,f\end{matrix}\,\middle|\,z\right) \tag{20.63}$$

These trivial transformations are true for any number of elements. The following transformations are only valid for the given structure, unless the list of parameters contain a ellipsis '…'.

Elementary relations

$$F\left(\begin{matrix}a,\,b,\,\ldots\\ c,\,\ldots\end{matrix}\,\middle|\,z\right) \;=\; 1+z\,\frac{a\,b\,\ldots}{c\,\ldots}\,F\left(\begin{matrix}a+1,\,b+1,\,\ldots,\,1\\ c+1,\,\ldots,\,2\end{matrix}\,\middle|\,z\right) \tag{20.64}$$

$$(a-b)\,F\left(\begin{matrix}a,\,b,\,\ldots\\ c,\,\ldots\end{matrix}\,\middle|\,z\right) \;=\; a\,F\left(\begin{matrix}a+1,\,b,\,\ldots\\ c,\,\ldots\end{matrix}\,\middle|\,z\right)-b\,F\left(\begin{matrix}a,\,b+1,\,\ldots\\ c,\,\ldots\end{matrix}\,\middle|\,z\right) \tag{20.65}$$

$$(a-c)\,F\left(\begin{matrix}a,\,b,\,\ldots\\ c+1,\,\ldots\end{matrix}\,\middle|\,z\right) \;=\; a\,F\left(\begin{matrix}a+1,\,b,\,\ldots\\ c+1,\,\ldots\end{matrix}\,\middle|\,z\right)-c\,F\left(\begin{matrix}a,\,b,\,\ldots\\ c,\,\ldots\end{matrix}\,\middle|\,z\right) \tag{20.66}$$

These are given in [ConcMath], the following is taken from [ModAnl].

$$F\left(\begin{matrix}a,\,b\\ c\end{matrix}\,\middle|\,z\right) \;=\; F\left(\begin{matrix}a,\,b+1\\ c\end{matrix}\,\middle|\,z\right)-\frac{a\,z}{c}\,F\left(\begin{matrix}a+1,\,b+1\\ c+1\end{matrix}\,\middle|\,z\right) \tag{20.67}$$

More relations of this type are given in [HaMaFu].

Pfaff's reflection law and Euler's identity

Pfaff's reflection law can be given as either of:

$$\frac{1}{(1-z)^a}\,F\left(\begin{matrix}a,\,b\\c\end{matrix}\Big|\,\frac{-z}{1-z}\right) = F\left(\begin{matrix}a,\,c-b\\c\end{matrix}\Big|\,z\right) \tag{20.68a}$$

$$F\left(\begin{matrix}a,\,b\\c\end{matrix}\Big|\,z\right) = \frac{1}{(1-z)^a}\,F\left(\begin{matrix}a,\,c-b\\c\end{matrix}\Big|\,\frac{-z}{1-z}\right) \tag{20.68b}$$

$$= \frac{1}{(1-z)^b}\,F\left(\begin{matrix}c-a,\,b\\c\end{matrix}\Big|\,\frac{-z}{1-z}\right) \tag{20.68c}$$

Euler's identity is obtained by applying the reflection on both upper parameters:

$$F\left(\begin{matrix}a,\,b\\c\end{matrix}\Big|\,z\right) = (1-z)^{(c-a-b)}\,F\left(\begin{matrix}c-a,\,c-b\\c\end{matrix}\Big|\,z\right) \tag{20.69}$$

Euler's transformation can generalized for hypergeometric functions ${}_{r+1}F_r$, see [77]. While Pfaff's transformation cannot be generalized, the relation

$$(1-z)^d\,F\left(\begin{matrix}f-h+2,\,-(f-h+1),\,d\\f-h+1,\,h\end{matrix}\Big|\,z\right) = F\left(\begin{matrix}f,\,1+\frac{1}{2}f,\,d\\\frac{1}{2}f,\,h\end{matrix}\Big|\,\frac{-z}{1-z}\right) \tag{20.70}$$

which is reminiscent to the reflection law, is given in [77].

A transformation by Gauss

$$F\left(\begin{matrix}2a,\,2b\\a+b+\frac{1}{2}\end{matrix}\Big|\,z\right) = F\left(\begin{matrix}a,\,b\\a+b+\frac{1}{2}\end{matrix}\Big|\,4z(1-z)\right) \quad \text{where} \quad |z| < \frac{1}{2} \tag{20.71a}$$

$$F\left(\begin{matrix}a,\,b\\a+b+\frac{1}{2}\end{matrix}\Big|\,z\right) = F\left(\begin{matrix}2a,\,2b\\a+b+\frac{1}{2}\end{matrix}\Big|\,\frac{1-\sqrt{1-z}}{2}\right) \tag{20.71b}$$

Note that the right hand side of relation 20.71a does not change if z is replaced by $1-z$, so it seems that

$$F\left(\begin{matrix}2a,\,2b\\a+b+\frac{1}{2}\end{matrix}\Big|\,z\right) = F\left(\begin{matrix}2a,\,2b\\a+b+\frac{1}{2}\end{matrix}\Big|\,1-z\right) \tag{20.72}$$

However, the relation is true only for terminating series, that is, for polynomials. Rewriting relation 20.71a for the argument $\frac{1-z}{2}$ we get

$$F\left(\begin{matrix}2a,\,2b\\a+b+\frac{1}{2}\end{matrix}\Big|\,\frac{1-z}{2}\right) = F\left(\begin{matrix}a,\,b\\a+b+\frac{1}{2}\end{matrix}\Big|\,1-z^2\right) \tag{20.73}$$

$$F\left(\begin{matrix}2a,\,2b\\a+b+\frac{1}{2}\end{matrix}\Big|\,\frac{1-\sqrt{1-z^2}}{2}\right) = F\left(\begin{matrix}a,\,b\\a+b+\frac{1}{2}\end{matrix}\Big|\,z^2\right) \tag{20.74}$$

Whipple's identity and quadratic transformations

$$F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}a+\frac{1}{2},\,1-a-b-c\\1+a-b,\,1+a-c\end{matrix}\Big|\,\frac{-4z}{(1-z)^2}\right) = (1-z)^a\,F\left(\begin{matrix}a,\,b,\,c\\1+a-b,\,1+a-c\end{matrix}\Big|\,z\right) \tag{20.75}$$

Specializing 20.75 for $c = (a+1)/2$ (note the symmetry between b and c so specializing for $c = (b+1)/2$ produces the identical relation) gives

$$F\left(\begin{matrix}a,\, b\\ 1+a-b\end{matrix}\Big|\, z\right) = \frac{1}{(1-z)^a}\, F\left(\begin{matrix}\frac{1}{2}a,\, \frac{1}{2}a+\frac{1}{2}-b\\ 1+a-b\end{matrix}\Big|\, \frac{-4z}{(1-z)^2}\right) \tag{20.76}$$

$$F\left(\begin{matrix}a,\, b\\ a+b+\frac{1}{2}\end{matrix}\Big|\, z\right) = \left(\frac{2\,(1-\sqrt{1-z})}{z}\right)^{2a} F\left(\begin{matrix}2a,\, a-b+\frac{1}{2}\\ a+b+\frac{1}{2}\end{matrix}\Big|\, -\frac{\left(1-\sqrt{1-z}\right)^2}{z}\right) \tag{20.77}$$

With $c := a-b$ in 20.76 one gets:

$$F\left(\begin{matrix}a,\, a-c\\ 1+c\end{matrix}\Big|\, z\right) = \frac{1}{(1-z)^a}\, F\left(\begin{matrix}\frac{1}{2}a,\, \frac{1}{2}-\frac{1}{2}a+c\\ 1+c\end{matrix}\Big|\, \frac{-4z}{(1-z)^2}\right) \tag{20.78}$$

If $b = a$ in relation 20.76, then

$$F\left(\begin{matrix}a,\, a\\ 1\end{matrix}\Big|\, z\right) = \frac{1}{(1-z)^a}\, F\left(\begin{matrix}\frac{1}{2}a,\, \frac{1}{2}-\frac{1}{2}a\\ 1\end{matrix}\Big|\, \frac{-4z}{(1-z)^2}\right) \tag{20.79}$$

Observe that the hypergeometric function on the right side does not change when replacing a by $1-a$.

Similarly as for the relations by Gauss, from relations 20.76 and 20.77:

$$F\left(\begin{matrix}a,\, b\\ 1+a-b\end{matrix}\Big|\, -\frac{1-z}{1+z}\right) = \left(\frac{1+z}{2}\right)^a F\left(\begin{matrix}\frac{1}{2}a,\, \frac{1}{2}a+\frac{1}{2}-b\\ 1+a-b\end{matrix}\Big|\, 1-z^2\right) \tag{20.80}$$

$$F\left(\begin{matrix}a,\, b\\ 1+a-b\end{matrix}\Big|\, -\frac{1-\sqrt{1-z^2}}{1+\sqrt{1-z^2}}\right) = \left(\frac{1+\sqrt{1-z^2}}{2}\right)^a F\left(\begin{matrix}\frac{1}{2}a,\, \frac{1}{2}a+\frac{1}{2}-b\\ 1+a-b\end{matrix}\Big|\, z^2\right) \tag{20.81}$$

$$F\left(\begin{matrix}a,\, b\\ a+b+\frac{1}{2}\end{matrix}\Big|\, 1-z^2\right) = \left(\frac{2}{1+z}\right)^{2a} F\left(\begin{matrix}2a,\, a-b+\frac{1}{2}\\ a+b+\frac{1}{2}\end{matrix}\Big|\, -\frac{1-z}{1+z}\right) \tag{20.82}$$

$$F\left(\begin{matrix}a,\, b\\ a+b+\frac{1}{2}\end{matrix}\Big|\, z^2\right) = \left(\frac{2}{1+\sqrt{1-z^2}}\right)^{2a} F\left(\begin{matrix}2a,\, a-b+\frac{1}{2}\\ a+b+\frac{1}{2}\end{matrix}\Big|\, -\frac{1-\sqrt{1-z^2}}{1+\sqrt{1-z^2}}\right) \tag{20.83}$$

The transformations

$$F\left(\begin{matrix}a,\, b\\ a-b+1\end{matrix}\Big|\, z\right) = (1+z)^{-a}\, F\left(\begin{matrix}\frac{1}{2}a,\, \frac{1}{2}a+\frac{1}{2}\\ a-b+1\end{matrix}\Big|\, \frac{4z}{(1+z)^2}\right) \tag{20.84a}$$

$$= (1-z)^{-a}\, F\left(\begin{matrix}\frac{1}{2}a,\, \frac{1}{2}a-b+\frac{1}{2}\\ a-b+1\end{matrix}\Big|\, \frac{-4z}{(1-z)^2}\right) \tag{20.84b}$$

$$= (1\pm\sqrt{z})^{-2a}\, F\left(\begin{matrix}a,\, a-b+\frac{1}{2}\\ 2a-2b+1\end{matrix}\Big|\, \frac{\pm 4\sqrt{z}}{(1\pm\sqrt{z})^2}\right) \tag{20.84c}$$

are given in [HaMaFu]. Specializing for $a = b$ gives

$$F\left(\begin{matrix}a,\, a\\ 1\end{matrix}\Big|\, z\right) = (1+z)^{-a}\, F\left(\begin{matrix}\frac{1}{2}a,\, \frac{1}{2}a+\frac{1}{2}\\ 1\end{matrix}\Big|\, \frac{4z}{(1+z)^2}\right) \tag{20.84d}$$

$$= (1-z)^{-a}\, F\left(\begin{matrix}\frac{1}{2}a,\, \frac{1}{2}-\frac{1}{2}a\\ 1\end{matrix}\Big|\, \frac{-4z}{(1-z)^2}\right) \tag{20.84e}$$

$$= (1\pm\sqrt{z})^{-2a}\, F\left(\begin{matrix}a,\, \frac{1}{2}\\ 1\end{matrix}\Big|\, \frac{\pm 4\sqrt{z}}{(1\pm\sqrt{z})^2}\right) \tag{20.84f}$$

The following transformations are given in [77]:

$$(1-z)^{-1}\, F\left(\begin{matrix}a,\, b,\, 1\\ \frac{1}{2}+\frac{1}{2}a+\frac{1}{2}b,\, 2\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}\frac{1}{2}+\frac{1}{2}a,\, \frac{1}{2}+\frac{1}{2}b,\, 1\\ \frac{1}{2}+\frac{1}{2}a+\frac{1}{2}b,\, 2\end{matrix}\Big|\, 4z(1-z)\right) \tag{20.85}$$

$$(1-z)^a\, F\left(\begin{matrix}a,\, b,\, c\\ 1+a-b,\, 1+a-c\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}\frac{1}{2}a,\, \frac{1}{2}+\frac{1}{2}a,\, 1+a-b-c\\ 1+a-b,\, 1+a-c\end{matrix}\Big|\, \frac{-4z}{(1-z)^2}\right) \tag{20.86}$$

The nonlinear transformation (given in [83])

$$F\left(\begin{matrix}a,\,b\\ c\end{matrix}\Big|\,z\right) \;=\; \sum_{n=0}^{\infty} d_n\,\omega^n \tag{20.87a}$$

where

$$\omega \;=\; \frac{-4z}{(1-z)^2}, \qquad z \;=\; \frac{\sqrt{1-\omega}-1}{\sqrt{1-\omega}+1} \tag{20.87b}$$

and

$$d_0 \;=\; 1 \tag{20.87c}$$

$$d_1 \;=\; \frac{2a\,(c-2b)}{c} \tag{20.87d}$$

$$d_{n+2} \;=\; \frac{2\,(c-2b)\,(n+1+a)\,d_{n+1}+(n+2a)\,(n+2a+1-c)\,d_n}{(n+2)\,(n+1+c)} \tag{20.87e}$$

maps the complex (z-)plane into the unit circle. Thereby the ω-form of the series converges for all $z \neq 1$.

Clausen's product formulas

$$\left[F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,z\right)\right]^2 \;=\; F\left(\begin{matrix}2a,\,a+b,\,2b\\ a+b+\frac{1}{2},\,2a+2b\end{matrix}\Big|\,z\right) \tag{20.88a}$$

$$F\left(\begin{matrix}\frac{1}{4}+a,\,\frac{1}{4}+b\\ 1+a+b\end{matrix}\Big|\,z\right)\,F\left(\begin{matrix}\frac{1}{4}-a,\,\frac{1}{4}-b\\ 1-a-b\end{matrix}\Big|\,z\right) \;=\; F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}+a-b,\,\frac{1}{2}-a+b\\ 1+a+b,\,1-a-b\end{matrix}\Big|\,z\right) \tag{20.88b}$$

If $a = b + \frac{1}{2}$ in 20.88a then (two parameters on the right hand side cancel)

$$\left[F\left(\begin{matrix}b+\frac{1}{2},\,b\\ 2b+1\end{matrix}\Big|\,z\right)\right]^2 \;=\; F\left(\begin{matrix}2b+\frac{1}{2},\,2b\\ 4b+1\end{matrix}\Big|\,z\right) \tag{20.89}$$

and the right hand side again matches the structure on the left. The corresponding function can be identified (see [PiAGM]) as $G_b(z) = \left(\frac{1+\sqrt{1-z}}{2}\right)^{-2b}$. One has $G_{n\,m}(z) = (G_n(z))^m$.

Specializing relation 20.88b for $b = -a$ we get

$$\left[F\left(\begin{matrix}\frac{1}{4}+a,\,\frac{1}{4}-a\\ 1\end{matrix}\Big|\,z\right)\right]^2 \;=\; F\left(\begin{matrix}\frac{1}{2}+2a,\,\frac{1}{2},\,\frac{1}{2}-2a\\ 1,\,1\end{matrix}\Big|\,z\right) \tag{20.90}$$

For $a = 0$, $z = 1$ (or z a sixth power) this relation is an identity between the square of a sum of squares and a sum of cubes:

$$\left[\sum_{n=0}^{\infty}\left[\prod_{j=1}^{n}\frac{1/4+j-1}{j}\right]^2\right]^2 \;=\; \sum_{n=0}^{\infty}\left[\prod_{j=1}^{n}\frac{1/2+j-1}{j}\right]^3 \;=\; 1.39320392968\ldots \tag{20.91a}$$

For the square root of the expressions ($\sqrt{1.39320\ldots} = 1.180340\ldots$) we have

$$F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\Big|\,\frac{1}{2}\right) \;=\; \left[\sum_{n=-\infty}^{\infty} e^{-n^2\,\pi}\right]^2 \;=\; 1.180340599016\ldots \tag{20.92}$$

This relation is given in [Finch].

The Kummer transformation

The Kummer transformation connects two hypergeometric function of type ${}_1F_1$:

$$\exp(z)\, F\left(\begin{matrix} a \\ a+b \end{matrix}\middle|\, -z\right) \;=\; F\left(\begin{matrix} b \\ a+b \end{matrix}\middle|\, z\right) \tag{20.93}$$

The relation is not valid if both a and b are negative integers. In that case one obtains the Padé approximants of $\exp(z)$, see relation 17.41 on page 436.

A transformation from ${}_1F_1$ to ${}_2F_3$ is given by

$$F\left(\begin{matrix} a \\ b \end{matrix}\middle|\, z\right)\, F\left(\begin{matrix} a \\ b \end{matrix}\middle|\, -z\right) \;=\; F\left(\begin{matrix} a,\, b-a \\ b,\, \frac{1}{2}b,\, \frac{1}{2}(b+1) \end{matrix}\middle|\, \frac{z^2}{4}\right) \tag{20.94}$$

Setting $b = 2a$ and using 20.93 gives

$$\left[F\left(\begin{matrix} a \\ 2a \end{matrix}\middle|\, z\right)\right]^2 \;=\; \exp(z)\, F\left(\begin{matrix} a \\ a+\frac{1}{2},\, 2a \end{matrix}\middle|\, \frac{z^2}{4}\right) \tag{20.95}$$

The following transformation connects functions ${}_0F_1$ and ${}_2F_3$:

$$F\left(\begin{matrix} - \\ a \end{matrix}\middle|\, z\right)\, F\left(\begin{matrix} - \\ b \end{matrix}\middle|\, z\right) \;=\; F\left(\begin{matrix} \frac{1}{2}(a+b),\, \frac{1}{2}(a+b-1) \\ a,\, b,\, a+b-1 \end{matrix}\middle|\, 4z\right) \tag{20.96}$$

Setting $b = a$ gives

$$\left[F\left(\begin{matrix} - \\ a \end{matrix}\middle|\, z\right)\right]^2 \;=\; F\left(\begin{matrix} a-\frac{1}{2} \\ a,\, 2a-1 \end{matrix}\middle|\, 4z\right) \tag{20.97}$$

From relations 20.95 and 20.97 one can obtain

$$\exp(z)\, F\left(\begin{matrix} - \\ a \end{matrix}\middle|\, \frac{z^2}{4}\right) \;=\; F\left(\begin{matrix} a-\frac{1}{2} \\ 2a-1 \end{matrix}\middle|\, 2z\right) \tag{20.98}$$

A transformation by Ramanujan

For comparison, we give the quadratic transform

$$F\left(\begin{matrix} \frac{1}{2},\, \frac{1}{2} \\ 1 \end{matrix}\middle|\, z'\right) \;=\; (1+z)\, F\left(\begin{matrix} \frac{1}{2},\, \frac{1}{2} \\ 1 \end{matrix}\middle|\, z^2\right) \tag{20.99a}$$

where

$$z \;=\; \frac{1-(1-z')^{1/2}}{1+(1-z')^{1/2}} \tag{20.99b}$$

$$z' \;=\; 1-\left(\frac{1-z}{1+z}\right)^2 \tag{20.99c}$$

It is (see [46]) the special case $a = 1/2$ and $b = 1/2$ of

$$F\left(\begin{matrix} a,\, b \\ 2b \end{matrix}\middle|\, \frac{4z}{(1+z)^2}\right) \;=\; F\left(\begin{matrix} a,\, b \\ 2b \end{matrix}\middle|\, 1-\left(\frac{1-z}{1+z}\right)^2\right) \;=\; (1+z)^{2a}\, F\left(\begin{matrix} a,\, a-b+\frac{1}{2} \\ b+\frac{1}{2} \end{matrix}\middle|\, z^2\right) \tag{20.100}$$

The following transformation is due to Ramanujan:

$$F\left(\begin{matrix} \frac{1}{3},\, \frac{2}{3} \\ 1 \end{matrix}\middle|\, z'\right) \;=\; (1+2\,z)\, F\left(\begin{matrix} \frac{1}{3},\, \frac{2}{3} \\ 1 \end{matrix}\middle|\, z^3\right) \tag{20.101a}$$

where

$$z = \frac{1-(1-z')^{1/3}}{1+2\,(1-z')^{1/3}} \tag{20.101b}$$

$$z' = 1-\left(\frac{1-z}{1+2z}\right)^3 \tag{20.101c}$$

The general form is given in [17]:

$$F\left(\begin{matrix}c,\ c+\frac{1}{3}\\ \frac{3c+1}{2}\end{matrix}\middle|\ 1-\left(\frac{1-z}{1+2z}\right)^3\right) = (1+2z)^{3c}\,F\left(\begin{matrix}c,\ c+\frac{1}{3}\\ \frac{3c+5}{6}\end{matrix}\middle|\ z^3\right) \tag{20.102}$$

For $c = 1/3$ one gets relation 20.101a. A computer algebra proof that relation 20.102 is is the only possible generalization of relation 20.101c is given in [70]. In [46] various such transformations and their generalizations are given. For example, the transformation

$$F\left(\begin{matrix}\frac{1}{4},\ \frac{3}{4}\\ 1\end{matrix}\middle|\ z'\right) = \sqrt{1+3\,z}\,F\left(\begin{matrix}\frac{1}{4},\ \frac{3}{4}\\ 1\end{matrix}\middle|\ z^2\right) \tag{20.103a}$$

where

$$z' = 1-\left(\frac{1-z}{1+3\,z}\right)^2 \tag{20.103b}$$

$$z = \frac{1-(1-z')^{1/2}}{1+3\,(1-z')^{1/2}} \tag{20.103c}$$

is the special case $d = 1/4$ of

$$F\left(\begin{matrix}d,\ d+\frac{1}{2}\\ \frac{4d+2}{3}\end{matrix}\middle|\ z'\right) = (1+3\,z)^{2d}\,F\left(\begin{matrix}d,\ d+\frac{1}{2}\\ \frac{4d+5}{6}\end{matrix}\middle|\ z^2\right) \tag{20.104}$$

20.4.2 Examples: elementary functions

The 'well-known' functions like exp, log and sin are expressed as hypergeometric functions. In some cases a transform is applied to give an alternative series.

Binomial series and powers of square roots

$$\frac{1}{(1-z)^a} = F\left(\begin{matrix}a\\ \end{matrix}\middle|\ z\right) = \sum_{k=0}^{\infty}\binom{a+k-1}{k}z^k = F\left(\begin{matrix}-a\\ \end{matrix}\middle|\ \frac{-z}{1-z}\right) \tag{20.105a}$$

$$(1+z)^a = F\left(\begin{matrix}-a\\ \end{matrix}\middle|\ -z\right) = \sum_{k=0}^{\infty}\binom{a}{k}z^k = F\left(\begin{matrix}a\\ \end{matrix}\middle|\ \frac{z}{1+z}\right) \tag{20.105b}$$

$$F\left(\begin{matrix}\frac{n}{2},\ \frac{n+1}{2}\\ \frac{1}{2}\end{matrix}\middle|\ z\right) = \frac{(1-\sqrt{z})^{-n}+(1+\sqrt{z})^{-n}}{2} \tag{20.106a}$$

$$F\left(\begin{matrix}\frac{n+1}{2},\ \frac{n+2}{2}\\ \frac{3}{2}\end{matrix}\middle|\ z\right) = \frac{(1-\sqrt{z})^{-n}-(1+\sqrt{z})^{-n}}{2\,n\,\sqrt{z}} \tag{20.106b}$$

$$F\left(\begin{matrix}\frac{n}{2},\ \frac{n+1}{2}\\ n+1\end{matrix}\middle|\ z\right) = \left(\frac{1+\sqrt{1-z}}{2}\right)^{-n} \tag{20.106c}$$

Chebyshev polynomials

$$F\left(\begin{matrix}n,\,-n\\ \frac{1}{2}\end{matrix}\middle|\, z\right) = T_n(1-2z) \tag{20.107a}$$

$$T_n(z) = F\left(\begin{matrix}n,\,-n\\ \frac{1}{2}\end{matrix}\middle|\, \frac{1-z}{2}\right) = T_{n/2}(2\,z^2-1) = T_{n/2}(T_2(z)) \quad \text{by 20.73} \tag{20.107b}$$

$$U_n(z) = (n+1)\,F\left(\begin{matrix}-n,\,n+2\\ \frac{3}{2}\end{matrix}\middle|\, \frac{1-z}{2}\right) \tag{20.107c}$$

Using relation 19.44 (as $T_n(T_{1/n}(z)) = z = \mathrm{id}(z)$) we find that

$$\left[F\left(\begin{matrix}n,\,-n\\ \frac{1}{2}\end{matrix}\middle|\, \frac{1-z}{2}\right)\right]^{-1} = F\left(\begin{matrix}\frac{1}{n},\,-\frac{1}{n}\\ \frac{1}{2}\end{matrix}\middle|\, \frac{1-z}{2}\right) \tag{20.108}$$

near $z=1$ (here F^{-1} denotes the inverse function).

Hermite polynomials

$$H_n(z) = (2z)^n\,F\left(\begin{matrix}-\frac{1}{2}\,n,\,-\frac{1}{2}\,(n-1)\\ \end{matrix}\middle|\, -\frac{1}{z^2}\right) \tag{20.109}$$

We have $H_0(z)=1$, $H_1(z)=2z$ and $H_{n+1}(z) = 2z\,H_n(z) - 2n\,H_{n-1}(x)$.

Logarithm and exponential function

$$\log(1+z) = z\,F\left(\begin{matrix}1,\,1\\ 2\end{matrix}\middle|\, -z\right) = \sum_{k=0}^{\infty} \frac{(-1)^k\,z^{k+1}}{k+1} \tag{20.110}$$

$$\exp(z) = F\left(\begin{matrix}\\ \end{matrix}\middle|\, z\right) = \sum_{k=0}^{\infty} \frac{z^k}{k!} \tag{20.111}$$

Bessel functions, error function and incomplete beta function

The following expressions for the modified Bessel functions of order $c-1$ are given in [ConcMath]:

$$F\left(\begin{matrix}\\ c\end{matrix}\middle|\, z\right) = \frac{(c-1)!}{z^{(c-1)/2}}\,I_{c-1}(2\sqrt{z}) \tag{20.112}$$

$$F\left(\begin{matrix}\\ 1\end{matrix}\middle|\, z\right) = I_0(2\sqrt{z}) = \sum_{k=0}^{\infty} \frac{z^k}{k!^2} \tag{20.113}$$

Error function:

$$\mathrm{erf}(z) := \frac{2}{\sqrt{\pi}} \int_{t=0}^{z} \exp(-t^2)\,dt = \frac{2\,z}{\sqrt{\pi}}\,F\left(\begin{matrix}\frac{1}{2}\\ \frac{3}{2}\end{matrix}\middle|\, -z^2\right) \tag{20.114a}$$

$$= \frac{2\,z}{\sqrt{\pi}}\,\exp(-z^2)\,F\left(\begin{matrix}1\\ \frac{3}{2}\end{matrix}\middle|\, z^2\right) \tag{20.114b}$$

where we used the Kummer transformation (relation 20.93).

Incomplete beta function:

$$B_z(p,q) := \int_{t=0}^{z} t^{p-1}\,(1-t)^{q-1}\,dt \tag{20.115a}$$

$$= \frac{z^p}{p}\,F\left(\begin{matrix}p,\,1-q\\ p+1\end{matrix}\middle|\,z\right) \tag{20.115b}$$

Trigonometric and hyperbolic functions

Series for sine and cosine:

$$\sin(z) = z\,F\left(\begin{matrix}\\ \frac{3}{2}\end{matrix}\middle|\,\frac{-z^2}{4}\right) = \sum_{k=0}^{\infty}\frac{(-1)^k\,z^{2k+1}}{(2k+1)!} \tag{20.116a}$$

$$\sinh(z) = z\,F\left(\begin{matrix}\\ \frac{3}{2}\end{matrix}\middle|\,\frac{z^2}{4}\right) = \sum_{k=0}^{\infty}\frac{z^{2k+1}}{(2k+1)!} \tag{20.116b}$$

Applying 20.97 on page 503 on 20.116a gives

$$(\sin(z))^2 = z^2\,F\left(\begin{matrix}1\\ \frac{3}{2},\,2\end{matrix}\middle|\,-z^2\right) \tag{20.117}$$

$$\cos(z) = F\left(\begin{matrix}\\ \frac{1}{2}\end{matrix}\middle|\,\frac{-z^2}{4}\right) = \sum_{k=0}^{\infty}\frac{(-1)^k\,z^{2k}}{(2k)!} \tag{20.118a}$$

$$\cosh(z) = F\left(\begin{matrix}\\ \frac{1}{2}\end{matrix}\middle|\,\frac{z^2}{4}\right) = \sum_{k=0}^{\infty}\frac{z^{2k}}{(2k)!} \tag{20.118b}$$

Inverse trigonometric and hyperbolic functions

Series for the inverse sine and cosine:

$$\arcsin(z) = z\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ \frac{3}{2}\end{matrix}\middle|\,z^2\right) \tag{20.119a}$$

$$= z\,F\left(\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\middle|\,\frac{1-\sqrt{1-z^2}}{2}\right) \quad \text{by 20.71b} \tag{20.119b}$$

$$(\arcsin(z))^2 = z^2\,F\left(\begin{matrix}1,\,1,\,1\\ \frac{3}{2},\,2\end{matrix}\middle|\,z^2\right) \quad \text{by 20.88a} \tag{20.120}$$

$$\operatorname{arcsinh}(z) = \log(z+\sqrt{1+z^2}) = z\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ \frac{3}{2}\end{matrix}\middle|\,-z^2\right) \tag{20.121a}$$

$$= \frac{z}{\sqrt{1+z^2}}\,F\left(\begin{matrix}\frac{1}{2},\,1\\ \frac{3}{2}\end{matrix}\middle|\,\frac{z^2}{1+z^2}\right) \quad \text{by 20.68b} \tag{20.121b}$$

$$= z\,F\left(\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\middle|\,\frac{1-\sqrt{1+z^2}}{2}\right) \quad \text{by 20.71b} \tag{20.121c}$$

Series for the inverse tangent and cotangent:

$$\arctan(z) = z\,F\left(\begin{matrix}\frac{1}{2},\,1\\ \frac{3}{2}\end{matrix}\Big|\,-z^2\right) = \sum_{k=0}^{\infty}\frac{(-1)^k\,z^{2k+1}}{2k+1} \tag{20.122a}$$

$$= \frac{z}{\sqrt{1+z^2}}\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ \frac{3}{2}\end{matrix}\Big|\,\frac{z^2}{1+z^2}\right) \quad \text{by 20.68b} \tag{20.122b}$$

$$= \frac{z}{1+z^2}\,F\left(\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\Big|\,\frac{z^2}{1+z^2}\right) \quad \text{by 20.68b} \tag{20.122c}$$

Applying relation 20.88a on 20.122b one obtains

$$(\arctan(z))^2 = \frac{z^2}{1+z^2}\,F\left(\begin{matrix}1,\,1,\,1\\ \frac{3}{2},\,2\end{matrix}\Big|\,\frac{z^2}{1+z^2}\right) \tag{20.123}$$

$$\operatorname{arctanh}(z) = z\,F\left(\begin{matrix}\frac{1}{2},\,1\\ \frac{3}{2}\end{matrix}\Big|\,z^2\right) = \sum_{k=0}^{\infty}\frac{z^{2k+1}}{2k+1} \tag{20.124}$$

$$\operatorname{arccot}(z) = \frac{1}{z}\,F\left(\begin{matrix}\frac{1}{2},\,1\\ \frac{3}{2}\end{matrix}\Big|\,-\frac{1}{z^2}\right) = \arctan\left(\frac{1}{z}\right) \tag{20.125a}$$

$$= \frac{z}{\sqrt{1+z^2}}\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ \frac{3}{2}\end{matrix}\Big|\,\frac{1}{1+z^2}\right) = \arcsin\left(\frac{1}{\sqrt{1+z^2}}\right) \tag{20.125b}$$

$$= \frac{z}{1+z^2}\,F\left(\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\Big|\,\frac{1}{1+z^2}\right) \tag{20.125c}$$

20.4.3 The function x^x

Boldly setting $a = 1+z$ in $(1+z)^a = F\left(\begin{matrix}-a\\ \end{matrix}\Big|\,-z\right)$ (relation 20.105b on page 504) gives

$$(1+z)^{(1+z)} = F\left(\begin{matrix}-1-z\\ \end{matrix}\Big|\,-z\right) = \exp\left[(1+z)\,\log(1+z)\right] \tag{20.126a}$$

$$= 1+\frac{(z+1)}{1}\,z\left[1+\frac{(z+0)}{2}\,z\left[1+\frac{(z-1)}{3}\,z\left[1+\frac{(z-2)}{4}\,z\left[1+\ldots\ \right]\right]\right]\right] \tag{20.126b}$$

$$= 1+z+z^2+\frac{1}{2}\,z^3+\frac{1}{3}\,z^4+\frac{1}{12}\,z^5+\frac{3}{40}\,z^6-\frac{1}{120}\,z^7+\frac{59}{2520}\,z^8-\frac{71}{5040}\,z^9\pm\ldots \tag{20.126c}$$

This somewhat surprising expression allows the computation of z^z without computing exp() or log(). The series converges for real $z>0$ so we can compute x^x (where $x = 1+z$) for real $x > +1$ as

$$x^x = F\left(\begin{matrix}-x\\ \end{matrix}\Big|\,-x+1\right) \tag{20.127a}$$

$$= 1+\frac{x-0}{1}\,(x-1)\left[1+\frac{x-1}{2}\,(x-1)\left[1+\frac{x-2}{3}\,(x-1)\left[1+\ldots\ \right]\right]\right] \tag{20.127b}$$

We denote the series obtained by truncating after the n-th term of the hypergeometric function by $g_n(x)$. For example, with $n=2$ and $n=4$ we obtain:

$$g_2(x) = \frac{1}{2}\,x^4-\frac{3}{2}\,x^3+\frac{5}{2}\,x^2-\frac{3}{2}\,x+1 \tag{20.128a}$$

$$= \frac{1}{2}\,z^4+\frac{1}{2}\,z^3+z^2+z+1 \tag{20.128b}$$

$$g_4(x) = \frac{1}{24}\,x^8-\frac{5}{12}\,x^7+\frac{15}{8}\,x^6-\frac{19}{4}\,x^5+\frac{61}{8}\,x^4-\frac{31}{4}\,x^3+\frac{131}{24}\,x^2-\frac{25}{12}\,x+1 \tag{20.128c}$$

$$= \frac{1}{24}\,z^8-\frac{1}{12}\,z^7+\frac{1}{8}\,z^6+\frac{1}{12}\,z^5+\frac{1}{3}\,z^4+\frac{1}{2}\,z^3+z^2+z+1 \tag{20.128d}$$

We have $g_n(n) = n^n$ and further $g_n(k) = k^k$ for all integer $k \le n$. Thus we have just invented a curious way to find polynomials of degree $2n$ that interpolate k^k for $0 \le k \le n$ (setting $0^0 := 1$ for our purposes).

The polynomials actually give acceptable estimates for x^x also for non-integer x, especially for x near 1. The (unique) degree-n polynomials $i_n(x)$ that are obtained by interpolating the values k^k have much bigger coefficients and give values far away from x^x for non-integer arguments x.

For $0 < x < n$ the interpolating polynomials $i_n(x)$ give an estimate that is consistently worse than $g_n(x)$ for non-integer values of x. The same is true even for the polynomials $i_{2n}(x)$ that interpolate k^k for $0 \le k \le 2n$ (so that $\deg(i_{2n}) = \deg(g_n) = 2n$). In fact, the $i_{2n}(x)$ approximate consistently worse than $i_n(x)$ for non-integer x.

Finally, the Padé approximants $p_{[n,n]}(x)$ for $g_n(x)$ give estimates that are worse than with both $i_n(x)$ or $g_n(x)$. Further, $g_n(x) \neq x^x$ even for integer x and the $p_{[n,n]}(x)$ have a pole on the real axis near $x = 1$.

That is, we found a surprisingly good and compact polynomial approximation for the function x^x.

In a similar manner (using relation 20.106c):

$$(1-z)^{(1-z)} = F\left(\begin{matrix}\frac{z-1}{2},\ \frac{z}{2}\\ z\end{matrix}\,\middle|\, 4\,z\,(1-z)\right) \tag{20.129a}$$

$$= 1 + R_1\left[1 + R_2\left[1 + R_3\left[1 + \ldots\left[1 + R_k\left[\quad\ldots\quad\right]\right]\right]\right]\right] \quad \text{where} \tag{20.129b}$$

$$R_k := \frac{z\,(z-1)}{k}\,\frac{(z+2k-2)\,(z+2k-3)}{(z+k-1)} \tag{20.129c}$$

The Taylor series is that of relation 20.126c with z replaced by $-z$. Further, purely symbolically:

$$x^x = F\left(\begin{matrix}\frac{-x}{2}\ \frac{1-x}{2}\\ 1-x\end{matrix}\,\middle|\, 4\,x\,(1-x)\right) \tag{20.129d}$$

$$= 1 + S_1\left[1 + S_2\left[1 + S_3\left[1 + \ldots\left[1 + S_k\left[\quad\ldots\quad\right]\right]\right]\right]\right] \quad \text{where} \tag{20.129e}$$

$$S_k := \frac{x\,(x-1)}{k}\,\frac{(x-2k+1)\,(x-2k+2)}{(x-k)} \tag{20.129f}$$

The sequence of the n-th derivatives of $(1+z)^{1+z}$ at $z = 0$,

```
? Vec(serlaplace(exp((1+z)*log(1+z))))
  [1, 1, 2, 3, 8, 10, 54, -42, 944, -5112, 47160, -419760, 4297512, ...
```

is entry A005727 of [OEIS].

20.4.4 Elliptic K and E

In order to avoid the factor $\frac{\pi}{2}$ we let $\tilde{K} := \frac{2\,K}{\pi}$, $\tilde{E} := \frac{2\,E}{\pi}$.

$$\tilde{K}(k) = F\left(\begin{matrix}\frac{1}{2},\ \frac{1}{2}\\ 1\end{matrix}\,\middle|\, k^2\right) \tag{20.130a}$$

$$\tilde{E}(k) = F\left(\begin{matrix}-\frac{1}{2},\ \frac{1}{2}\\ 1\end{matrix}\,\middle|\, k^2\right) \tag{20.130b}$$

See section 17.5 on page 439 for explicit expansions. We further set

$$\tilde{N}(k) = F\left(\begin{matrix}-\frac{1}{2},\ -\frac{1}{2}\\ 1\end{matrix}\,\middle|\, k^2\right) \tag{20.130c}$$

$$= 1 + \frac{1}{4}\,k^2 + \frac{1}{64}\,k^4 + \frac{1}{256}\,k^6 + \frac{25}{16384}\,k^8 + \frac{49}{65536}\,k^{10} + \frac{441}{1048576}\,k^{10} + \ldots \tag{20.130d}$$

A special value is $\tilde{N}(1) = 4/\pi$. Most of the following relations can be written in several ways by one of the identities

$$k' = \sqrt{1-k^2} \tag{20.131a}$$

$$\frac{-4k^2}{(1-k^2)^2} = -\left(\frac{2k}{k'^2}\right)^2 = 1-\left(\frac{1+k^2}{1-k^2}\right)^2 \tag{20.131b}$$

$$-\frac{k^2}{1-k^2} = -\left(\frac{k}{k'}\right)^2 = -\frac{1-k'^2}{k'^2} \tag{20.131c}$$

$$-\left(\frac{k}{1-\sqrt{1-k^2}}\right)^2 = -\frac{1+k'}{1-k'} = -\frac{1-k'^2}{(1-k')^2} = -\frac{(1+k')^2}{1-k'^2} \tag{20.131d}$$

$$\frac{2}{1+k'} = 1+\frac{1-k'}{1+k'} \tag{20.131e}$$

$$\frac{4k}{(1+k)^2} = 1-\left(\frac{1-k}{1+k}\right)^2 \tag{20.131f}$$

Relations for K

$$\tilde{K}(k) = F\left(\begin{matrix}\frac{1}{2},\frac{1}{2}\\1\end{matrix}\middle| k^2\right) \tag{20.132}$$

$$= \frac{1}{k'}F\left(\begin{matrix}\frac{1}{2},\frac{1}{2}\\1\end{matrix}\middle| -\left(\frac{k}{k'}\right)^2\right) \qquad \text{by 20.68b} \tag{20.133}$$

$$\tilde{K}(k) = \frac{1}{1-k'}F\left(\begin{matrix}\frac{1}{2},\frac{1}{2}\\1\end{matrix}\middle| -\frac{1+k'}{1-k'}\right) \tag{20.134}$$

From the product form (relation 17.59 on page 440) we get:

$$\tilde{K}(k) = \frac{2}{1+k'}F\left(\begin{matrix}\frac{1}{2},\frac{1}{2}\\1\end{matrix}\middle| \left(\frac{1-k'}{1+k'}\right)^2\right) \tag{20.135a}$$

$$= \left(1+\frac{1-k'}{1+k'}\right)F\left(\begin{matrix}\frac{1}{2},\frac{1}{2}\\1\end{matrix}\middle| \left(\frac{1-k'}{1+k'}\right)^2\right) \tag{20.135b}$$

With $z(k) := \frac{1-k'}{1+k'}$ the last relation can be written as:

$$\tilde{K}(k) = (1+z(k))\ \tilde{K}\left(z(k)\right) \tag{20.135c}$$

Relation 20.84f on page 501 with $a = \frac{1}{2}$ gives

$$\tilde{K}(k) = \frac{1}{1+k}\tilde{K}\left(\frac{2\sqrt{k}}{1+k}\right) \tag{20.136}$$

$$\tilde{K}(k) = \frac{1}{\sqrt{1-k^2}}F\left(\begin{matrix}\frac{1}{4},\frac{1}{4}\\1\end{matrix}\middle| \frac{-4k^2}{(1-k^2)^2}\right) \tag{20.137a}$$

$$= \frac{1}{k'}F\left(\begin{matrix}\frac{1}{4},\frac{1}{4}\\1\end{matrix}\middle| 1-\left(\frac{1+k^2}{1-k^2}\right)^2\right) \tag{20.137b}$$

Euler's transform on 20.137a gives:

$$\tilde{K}(k) = \frac{1}{k'}\,\frac{1+k^2}{1-k^2}\,F\left(\begin{matrix}\frac{3}{4},\,\frac{3}{4}\\ 1\end{matrix}\middle|\,-\left(\frac{2k}{k'^2}\right)^2\right) \tag{20.138}$$

$$\tilde{K}(k) = F\left(\begin{matrix}\frac{1}{4},\,\frac{1}{4}\\ 1\end{matrix}\middle|\,(2\,k\,k')^2\right) \qquad \text{by 20.71a} \tag{20.139}$$

Relations for E

$$\tilde{E}(k) = F\left(\begin{matrix}-\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\middle|\,k^2\right) \tag{20.140}$$

$$\tilde{E}(k) = k'\,F\left(\begin{matrix}-\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\middle|\,-\left(\frac{k}{k'}\right)^2\right) \qquad \text{by 20.68b} \tag{20.141}$$

The following relation resembles relation 20.135a:

$$\tilde{E}(k) = \frac{\left(1+\sqrt{1-k^2}\right)}{2}\,F\left(\begin{matrix}-\frac{1}{2},\,-\frac{1}{2}\\ 1\end{matrix}\middle|\,\left(\frac{1-\sqrt{1-k^2}}{1+\sqrt{1-k^2}}\right)^2\right) \tag{20.142a}$$

$$= \frac{1+k'}{2}\,F\left(\begin{matrix}-\frac{1}{2},\,-\frac{1}{2}\\ 1\end{matrix}\middle|\,\left(\frac{1-k'}{1+k'}\right)^2\right) \tag{20.142b}$$

$$= (1+z(k))^{-1}\;\tilde{N}\left(z(k)\right) \tag{20.142c}$$

Relations for N

$$\tilde{N}(k) = \sqrt{1-k^2}\,F\left(\begin{matrix}-\frac{1}{4},\,\frac{3}{4}\\ 1\end{matrix}\middle|\,\frac{-4\,k^2}{\left(1-k^2\right)^2}\right) \qquad \text{by 20.79} \tag{20.143a}$$

$$= k'\,F\left(\begin{matrix}-\frac{1}{4},\,\frac{3}{4}\\ 1\end{matrix}\middle|\,-\left(\frac{2k}{k'^2}\right)^2\right) \tag{20.143b}$$

$$= \sqrt{1+k^2}\,F\left(\begin{matrix}-\frac{1}{4},\,\frac{1}{4}\\ 1\end{matrix}\middle|\,\left(\frac{2k}{k^2+1}\right)^2\right) \qquad \text{by 20.68b} \tag{20.143c}$$

Relation 20.65 on page 499 with $a = b = -1/2$ and $c = 1$ gives

$$F\left(\begin{matrix}\frac{1}{2},\,-\frac{1}{2}\\ 1\end{matrix}\middle|\,z\right) = \frac{1}{2}\,F\left(\begin{matrix}\frac{3}{2},\,-\frac{1}{2}\\ 1\end{matrix}\middle|\,z\right) + \frac{1}{2}\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\middle|\,z\right) \tag{20.144}$$

Applying 20.68c on page 500 to the second function ($F\left(\begin{matrix}3/2,\,-1/2\\ 1\end{matrix}\middle|\,z\right) = (1-z)\,F\left(\begin{matrix}-1/2,\,-1/2\\ 1\end{matrix}\middle|\,\frac{-z}{1-z}\right)$) and rearranging gives

$$2\,\tilde{E}(k) - \tilde{K}(k) = k'\,\tilde{N}\left(i\,\frac{k}{k'}\right) \tag{20.145}$$

Fast computation of the function $\tilde{N}(k)$ is possible via

$$\tilde{N}(k) = (1+k)\;\tilde{E}\left(\frac{2\sqrt{k}}{1+k}\right) \tag{20.146}$$

The relation

$$2\,\tilde{E}(k) - k'^2\,\tilde{K}(k) \quad = \quad \tilde{N}(k) \tag{20.147}$$

allows to rewrite Legendre's relation 17.68b on page 441 as either

$$\frac{4}{\pi} \quad = \quad \tilde{N}\,\tilde{K}' + \tilde{K}\,\tilde{N}' - \tilde{K}\,\tilde{K}' \tag{20.148}$$

or (setting $N := \pi/2\,\tilde{N}$)

$$\pi \quad = \quad N\,K' + K\,N' - K\,K' \tag{20.149}$$

Product form revisited *

We start by restating relation 17.59 on page 440 as:

$$F\left(\left.\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\right|x^2\right) \quad = \quad \prod_{j=1}^{\infty}(1+x_j) \quad \text{where} \tag{20.150}$$

$$x_0 := x, \quad x_{j+1} := \frac{1-\sqrt{1-x_j^2}}{1+\sqrt{1-x_j^2}}, \quad 0 < x \leq 1$$

Now set

$$F_n(x) \quad = \quad \prod_{j=1}^{\infty}(1+(n-1)\,x_j) \quad \text{where} \tag{20.151}$$

$$x_0 := x, \quad x' := \sqrt[n]{1-x^n}, \quad x_{j+1} := \frac{1-x'_j}{1+(n-1)\,x'_j}, \quad 0 < x \leq 1$$

Then

$$F_2(x) \quad = \quad F\left(\left.\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\right|x^2\right) \tag{20.152}$$

The explicit series of F_2 was given (with an additional factor $\frac{\pi}{2}$) as formula 17.53d. For F_3 we find

$$\begin{aligned} F_3(x) \quad &= \quad F\left(\left.\begin{matrix}\frac{1}{3},\,\frac{2}{3}\\ 1\end{matrix}\right|x^3\right) \\ &= \quad 1 + \frac{2\,x^3}{9} + \frac{10\,x^6}{81} + \frac{560\,x^9}{6561} + \frac{3850\,x^{12}}{59049} + \frac{28028\,x^{15}}{531441} + \ldots \end{aligned} \tag{20.153}$$

The underlying generalization of the AGM can be found in [28]: let $\mathrm{AGM}_n(a,b)$ defined as the common limit of the iteration

$$a_{j+1} \quad = \quad \frac{a_j - b_j}{n} + b_j \tag{20.154a}$$

$$b_{j+1} \quad = \quad \left[a_{j+1}^n - \left(\frac{a_j - b_j}{n}\right)^n\right]^{1/n} \tag{20.154b}$$

starting with $a_0 = a$, $b_0 = b$. Note the index $j+1$ appearing on the right hand side of relation 20.154b. This is one possible generalization that resembles the alternative relations (17.3a ... 17.3c on page 429) for the AGM. By construction one has

$$c_{j+1}^n \quad := \quad a_{j+1}^n - b_{j+1}^n \;=\; \left(\frac{a_j - b_j}{n}\right)^n \tag{20.155}$$

The one-sided version is

$$\begin{aligned} \mathrm{AGM}_n(1,\, x) &= \mathrm{AGM}_n\left(1 + (n-1)\, x',\, 1 - x'\right) && (20.156a) \\ &= \left(1 + (n-1)\, x'\right)\, \mathrm{AGM}_n\left(1,\, \frac{1 - x'}{1 + (n-1)\, x'}\right) && (20.156b) \end{aligned}$$

where $x' = \sqrt[n]{1 - x^n}$. Setting

$$F_n(x) = \frac{1}{\mathrm{AGM}_n(1, x')} \qquad (20.157)$$

leads to the product given above. We have identified $F_2(x)$ and $F_3(x)$ (see [28] for the details). However, the functions $F_n(x)$ have not yet been identified for $n > 3$.

In the Taylor series

$$F_4(x) = 1 + \frac{3}{16}\, x^4 + \frac{27}{256}\, x^8 + \frac{303}{4096}\, x^{12} + \frac{59955}{1048576}\, x^{16} + \ldots \qquad (20.158)$$

only the first non-constant term agrees with the most obvious generalization:

$$F\left(\left.\begin{matrix} \frac{1}{4},\, \frac{3}{4} \\ 1 \end{matrix}\right| x^4\right) = 1 + \frac{3}{16}\, x^4 + \frac{105}{1024}\, x^8 + \frac{1155}{16384}\, x^{12} + \frac{225225}{4194304}\, x^{16} + \ldots \qquad (20.159)$$

It is not difficult to invent alternative iterations that allow fast computation of some function that is analytical near zero, the problem is to identify the function.

20.5 AGM based algorithms for hypergeometric functions

We give AGM based algorithms for $F\left(\left.\begin{matrix} 1/2-s,\, 1/2+s \\ 1 \end{matrix}\right| z\right)$ where $s \in \{0,\, 1/6,\, 1/4,\, 1/3\}$. These are taken from [25] and [45], both papers are recommended for further studies.

Algorithms for $F\left(\left.\begin{matrix} 1/2,\, 1/2 \\ 1 \end{matrix}\right| z\right)$

The following is relation 17.55a, the classical AGM algorithm which has quadratical convergence:

$$\begin{aligned} F\left(\left.\begin{matrix} 1/2,\, 1/2 \\ 1 \end{matrix}\right| z\right) &= 1/M\left(1,\, \sqrt{1 - z}\right) \quad \text{where} && (20.160) \\ M(a, b) &:= \left[(a + b)/2,\, \sqrt{a\, b}\right] \end{aligned}$$

We write the AGM as $M := [f(a, b),\, g(a, b)]$ in the obvious way.

A fourth order algorithm obtained by combining two steps of the classical AGM:

$$\begin{aligned} F\left(\left.\begin{matrix} 1/2,\, 1/2 \\ 1 \end{matrix}\right| z\right) &= 1/M\left(1,\, \sqrt[4]{1 - z}\right)^2 \quad \text{where} && (20.161) \\ M(a, b) &:= \left[(a + b)/2,\, \sqrt{a\, b\, (a^2 + b^2)/2}\right] \end{aligned}$$

Algorithms for $F\left(\begin{matrix}1/3,\,2/3\\1\end{matrix}\middle|\,z\right)$

A third order algorithm:

$$\begin{aligned} F\left(\begin{matrix}1/3,\,1/3\\1\end{matrix}\middle|\,z\right) &= 1/M\left(\sqrt[3]{1-z},\,1\right) = \sqrt[3]{1-z}\,F\left(\begin{matrix}2/3,\,2/3\\1\end{matrix}\middle|\,z\right) \quad \text{where} \\ M(a,b) &:= \left[(a+2\,b)/3,\ \sqrt[3]{b\,(a^2+a\,b+b^2)\,/3}\right] \end{aligned} \tag{20.162}$$

One further has

$$F\left(\begin{matrix}1/3,\,2/3\\1\end{matrix}\middle|\,z\right) = 1/M\left(1,\,\sqrt[3]{1-z}\right) \tag{20.163}$$

A quadratic algorithm:

$$\begin{aligned} F\left(\begin{matrix}1/3,\,2/3\\1\end{matrix}\middle|\,z\right) &= 1/M\left(1,\,\sqrt[3]{1-z}\right) \quad \text{where} \\ M(a,b) &:= \left[\frac{1}{2}\left(\sqrt[3]{2p-a^3}+\sqrt[3]{2m-a^3}\right),\ \frac{1}{2}\left(\sqrt[3]{p}+\sqrt[3]{m}\right)\right] \qquad \text{and} \\ p &:= b^3+t, \qquad m := b^3-t, \qquad t := \sqrt{b^6-a^3\,b^3} \end{aligned} \tag{20.164}$$

And again (see relation 20.162):

$$F\left(\begin{matrix}1/3,\,1/3\\1\end{matrix}\middle|\,z\right) = 1/M\left(\sqrt[3]{1-z},\,1\right) = \sqrt[3]{1-z}\,F\left(\begin{matrix}2/3,\,2/3\\1\end{matrix}\middle|\,z\right) \tag{20.165}$$

Another quadratic algorithm is

$$\begin{aligned} F\left(\begin{matrix}1/3,\,2/3\\1\end{matrix}\middle|\,z\right) &= 1/M\left(1,\,W\right) \quad \text{where} \\ M(a,b) &:= \left[(a+b)/2,\ \left(3\,\sqrt{b\,(b+2a)/3}-b\right)/2\right] \qquad \text{and} \\ W &:= \frac{1-R+R^2}{R}, \quad R := \left[\sqrt{u^2-1}+u\right]^{1/3}, \quad u := 1-2\,z \end{aligned} \tag{20.166}$$

It is given in the form

$$1/F\left(\begin{matrix}1/3,\,2/3\\1\end{matrix}\middle|\,(1-x)\,(1+x/2)^2\right) = M\left(1,\,x\right) \tag{20.167}$$

Algorithms for $F\left(\begin{matrix}1/4,\,3/4\\1\end{matrix}\middle|\,z\right)$

A quadratic algorithm:

$$\begin{aligned} F\left(\begin{matrix}1/4,\,3/4\\1\end{matrix}\middle|\,z\right) &= 1/M\left(1,\,\sqrt{1-z}\right)^{1/2} \quad \text{where} \\ M(a,b) &:= \left[(a+3b)/4,\ \sqrt{b\,(a+b)/2}\right] \end{aligned} \tag{20.168}$$

One further has

$$F\left(\begin{matrix}1/4,\,1/4\\1\end{matrix}\middle|\,z\right) = 1/M\left(\sqrt{1-z},\,1\right)^{1/2} = \sqrt{1-z}\,F\left(\begin{matrix}3/4,\,3/4\\1\end{matrix}\middle|\,z\right) \tag{20.169}$$

Now set $A_k := \sqrt{(a_k + b_k)/2}$ and $B_k := \sqrt{b_k}$, then

$$A_{k+1} = \frac{1}{2}\,(A_k + B_k) \tag{20.170a}$$

$$B_{k+1} = \sqrt{A_k\,B_k} \tag{20.170b}$$

This is the iteration of the classical AGM. Thereby

$$M(a,b)^{1/2} = \mathrm{AGM}\left(\sqrt{\frac{a+b}{2}},\,\sqrt{b}\right) \tag{20.171}$$

and one can employ the AGM scheme of Schönhage (relations 17.4a... 17.4k on page 430).

Finally, we list algorithms for hypergeometric functions of other forms that are given in [45]:

Algorithms for $F\left(\begin{smallmatrix}1/3,\,1/6\\1\end{smallmatrix}\middle|\,z\right)$ and $F\left(\begin{smallmatrix}1/12,\,5/12\\1\end{smallmatrix}\middle|\,z\right)$

A quadratic algorithm is given as

$$1/\left[F\left(\begin{matrix}1/6,\,1/3\\1\end{matrix}\middle|\,27\,x^2\,(1-x)/4\right)\right]^2 = M\,(1,\,x) \tag{20.172}$$

One can solve for the argument of F and obtains

$$F\left(\begin{matrix}1/6,\,1/3\\1\end{matrix}\middle|\,z\right) = 1/M\,(1,\,W)^{1/2} \quad \text{where} \tag{20.173}$$

$$M(a,b) := \left[(a+3b)/4,\,(\sqrt{ab}+b)/2\right] \qquad \text{and}$$

$$W := \frac{1+R+R^2}{3\,R}, \quad R := \left[\sqrt{z^2-1}+z\right]^{1/3}$$

The following algorithm has quadratic convergence, W is as above:

$$F\left(\begin{matrix}1/12,\,5/12\\1\end{matrix}\middle|\,z\right) = 1/M\,(1,\,W)^{1/4} \quad \text{where} \tag{20.174}$$

$$M(a,b) := \left[(a+15b)/16,\,\left(\sqrt{b\,(a+3b)/4}+b\right)/2\right]$$

20.6 Multiplication of hypercomplex numbers *

An n-dimensional vector space (over a field) together with component-wise addition and a *multiplication table* that defines the product of two any (vector) components defines an *algebra*.

The product of two elements $x = \sum_k \alpha_k\,e_k$ and $y = \sum_j \beta_j\,e_j$ of the algebra is defined as

$$x \cdot y = \sum_{k,j=0}^{n-1} \left[(\alpha_k \cdot \beta_j)\,e_k\,e_j\right] \tag{20.175}$$

The quantities $e_k\,e_j$ are given in the multiplication table of the algebra. These can be arbitrary elements of the algebra, that is, linear combinations of the components. For example, a 2-dimensional algebra over the reals could have the following multiplication table:

```
          e0                  e1
e0:  (5*e1 + 3*e0)   (239*e0 + 3.1415*e1)
e1:     (0)          (17*e1 + 2.71828*e0)
```

Note that there is no neutral element of multiplication ('one'). Further, the algebra has *zero divisors*: the equation $x \cdot y = 0$ has a solution where neither element is zero, namely $x = e_1$, $y = e_0$. As almost all randomly defined algebras, it is completely uninteresting.

In what follows we will only consider algebras over the reals where the product of two components equals ± 1 times another component. For example, the *complex* numbers are a two-dimensional algebra with the multiplication table

```
      e0   e1
e0:  +e0  +e1
e1:  +e1  -e0
```

Which is, using the symbols '`1`' and '`i`', just

```
     1   i
1:  +1  +i
i:  +i  -1
```

We will denote the components of an n-dimensional algebra by the numbers $0, 1, \ldots, n-1$. The multiplication table for the complex numbers would thus be written as

```
     0   1
0:  +0  +1
1:  +1  -1
```

20.6.1 The Cayley-Dickson construction

The *Cayley-Dickson construction* allows to recursively generate multiplication tables for certain algebras where the dimension is a power of two. Let a, A, b and B be elements of a 2^{n-1}-dimensional algebra U. Define the multiplication rule for an algebra V (of dimension 2^n) written as pairs of elements of U via

$$(a, b) \cdot (A, B) \quad := \quad (a \cdot A - B \cdot b^*,\ a^* \cdot B + A \cdot b) \tag{20.176}$$

where the *conjugate* of an element is defined as

$$(a, b)^* \ := \ (a^*, -b) \tag{20.177}$$

and the conjugate of a real number a equals a (unmodified).

```
      0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f
0:   +0  +1  +2  +3  +4  +5  +6  +7  +8  +9  +a  +b  +c  +d  +e  +f
1:   +1  -0  -3  +2  -5  +4  +7  -6  -9  +8  +b  -a  +d  -c  -f  +e
2:   +2  +3  -0  -1  -6  -7  +4  +5  -a  -b  +8  +9  +e  +f  -c  -d
3:   +3  -2  +1  -0  -7  +6  -5  +4  -b  +a  -9  +8  +f  -e  +d  -c
4:   +4  +5  +6  +7  -0  -1  -2  -3  -c  -d  -e  -f  +8  +9  +a  +b
5:   +5  -4  +7  -6  +1  -0  +3  -2  -d  +c  -f  +e  -9  +8  -b  +a
6:   +6  -7  -4  +5  +2  -3  -0  +1  -e  +f  +c  -d  -a  +b  +8  -9
7:   +7  +6  -5  -4  +3  +2  -1  -0  -f  -e  +d  +c  -b  -a  +9  +8

8:   +8  +9  +a  +b  +c  +d  +e  +f  -0  -1  -2  -3  -4  -5  -6  -7
9:   +9  -8  +b  -a  +d  -c  -f  +e  +1  -0  +3  -2  +5  -4  -7  +6
a:   +a  -b  -8  +9  +e  +f  -c  -d  +2  -3  -0  +1  +6  +7  -4  -5
b:   +b  +a  -9  -8  +f  -e  +d  -c  +3  +2  -1  -0  +7  -6  +5  -4
c:   +c  -d  -e  -f  -8  +9  +a  +b  +4  -5  -6  -7  -0  +1  +2  +3
d:   +d  +c  -f  +e  -9  -8  -b  +a  +5  +4  -7  +6  -1  -0  -3  +2
e:   +e  +f  +c  -d  -a  +b  -8  -9  +6  +7  +4  -5  -2  +3  -0  -1
f:   +f  -e  +d  +c  -b  -a  +9  -8  +7  -6  +5  +4  -3  -2  +1  -0
```

Figure 20.4: Multiplication table for the sedenions. The entry in row R, column C gives the product $R \cdot C$ of the components R and C (hexadecimal notation).

Figure 20.4 gives the multiplication table for a 16-dimensional algebra, the so-called *sedenions*. The upper left (8×8) quarter gives the multiplication rule for the *octonions* (or *Cayley numbers*), the upper left 4×4 subtable gives the rule for the *quaternions* and the upper left 2×2 subtable corresponds to the complex numbers. Note that multiplication is in general neither commutative (only up to dimension 2) nor associative (only up to dimension 4).

The 2^n-dimensional algebras are (for $n > 1$) referred to as *hypercomplex numbers*. There is no generally accepted naming scheme for the algebras beyond dimension 16. We will use the names '2^n-ions'.

This form (relation 20.176) of the construction is given in [8], an alternative form is used in [Num]:

$$(a,\, b) \cdot (A,\, B) \quad := \quad (a \cdot A - B^* \cdot b,\; b \cdot A^* + B \cdot a) \tag{20.178}$$

It leads to a table that is the transposed of figure 20.4 on the preceding page.

```
          0 1 2 3 4 5 6 7  8 9 a b c d e f

    0:    + + + + + + + +  + + + + + + + +
    1:    + - - + - + + -  - + + - + - - +
    2:    + + - - - - + +  - - + + + + - -
    3:    + - + - - + - +  - + - + + - + -
    4:    + + + + - - - -  - - - - + + + +
    5:    + - + - + - + -  - + - + - + - +
    6:    + - - + + - - +  - + + - - + + -
    7:    + + - - + + - -  - - + + - - + +

    8:    + + + + + + + +  - - - - - - - -
    9:    + - + - + - - +  + - + - + - - +
    a:    + - - + + + - -  + - - + + + - -
    b:    + + - - + - + -  + + - - + - + -
    c:    + - - - - + + +  + - - - - + + +
    d:    + + - + - - - +  + + - + - - - +
    e:    + + + - - + - -  + + + - - + - -
    f:    + - + + - - + -  + - + + - - + -
```

Figure 20.5: Signs in the multiplication table for sedenions.

By construction, $e_0^2 = e_0$, $e_k^2 = -e_0$, $e_0\, e_k = e_k\, e_0 = e_k$ and $e_k\, e_j = -e_j\, e_k$ whenever both of k and j are nonzero. Further,

$$e_k\, e_j \quad = \quad \pm e_x \quad \text{where} \quad x = k \,\text{XOR}\, j \tag{20.179}$$

where the sign is to be determined. Figure 20.5 shows the pattern of the signs of the sedenion algebra. The lower left quarter is the transposed of the upper left quarter, so is the lower right quarter, except for its top row. These observations, together with the partial antisymmetry can be cast into an algorithm to compute the signs [FXT: demo arith/cayley-dickson-demo.cc]:

```
int CD_sign(ulong r, ulong c, ulong n)
// Signs in the multiplication table for the
//   algebra of n-ions (where n is a power of two)
//   that is obtained by the Cayley-Dickson construction:
// If component r is multiplied with component c then the
//   result is CD_sign(r,c,n) * (r XOR c).
{
    if ( (r==0) || (c==0) )  return +1;
    if ( c>=r )
    {
        if ( c>r )   return  -CD_sign(c, r, n);
        else  return -1;  // r==c
    }
    // here r>c (triangle below diagonal)
    ulong h = n>>1;
    if ( c>=h )  // right
    {
        // (upper right unreached)
        return   CD_sign(c-h, r-h, h); // lower right
    }
    else  // left
    {
        if ( r>=h )  return   CD_sign(c, r-h, h); // lower left
        else         return   CD_sign(r, c, h); // upper left
    }
}
```

The function uses at most $2 \cdot \log_2(n)$ steps. Note that the second row in the table is (the signed version of) the *Thue-Morse sequence*, see section 1.16.1 on page 34. The matrix filled with entries ± 1 according to figure 20.5 is a *Hadamard matrix*, see section 10.12 on page 334.

An iterative version of the function is [FXT: demo arith/cayley-dickson-demo.cc]:

```
inline void cp2(ulong a, ulong b, ulong &u, ulong &v)  { u=a; v=b; }
//
inline int CD_sign(ulong r, ulong c, ulong n)
{
    int s = +1;
start:
    if ( (r==0) || (c==0) )  return s;
    if ( c==r )  return -s;
    if ( c>r )   { swap2(r,c); s=-s; }
    n >>= 1;
    if ( c>=n )  cp2(c-n, r-n, r, c);
    else if ( r>=n )  cp2(c, r-n, r, c);
    goto start;
}
```

The computation of all $2^{24} = 16,777,216$ signs in the multiplication table for the '2^{12}-ions' takes less than 2 seconds.

```
          0  1  2  3  4  5  6  7

    0:  +0 +1 +2 +3 +4 +5 +6 +7          + + + + + + + +
    1:  +1 -0 +6 +4 -3 +7 -2 -5          + - + + - + - -
    2:  +2 -6 -0 +7 +5 -4 +1 -3          + - - + + - + -
    3:  +3 -4 -7 -0 +1 +6 -5 +2          + - - - + + - +
    4:  +4 +3 -5 -1 -0 +2 +7 -6          + + - - - + + -
    5:  +5 -7 +4 -6 -2 -0 +3 +1          + - + - - - + +
    6:  +6 +2 -1 +5 -7 -3 -0 +4          + + - + - - - +
    7:  +7 +5 +3 -2 +6 -1 -4 -0          + + + - + - - -
```

Figure 20.6: Alternative multiplication table for the octonions (left) and its sign pattern (right).

An alternative multiplication table for the octonions is given in figure 20.6. Its sign pattern is the 8×8 Hadamard matrix shown in figure 10.16 on page 335. Properties of this representation and the relation to shift register sequences are described in [40].

20.6.2 Fast multiplication of quaternions

```
   +--  0   1   2   3      +--  0   1   2   3        +--  0   1   2   3
   |                      |                        |
   0:   0   1   2   3      0:  -0   1   2   3      1 0:   0*  1   2   3
   1:   1   0   3   2      1:   1  -0   3   2      i 1:   1  -0   3  -2*
   2:   2   3   0   1      2:   2   3  -0   1      j 2:   2  -3* -0   1
   3:   3   2   1   0      3:   3   2   1  -0      k 3:   3   2  -1* -0
```

Figure 20.7: Scheme for the length-4 dyadic convolution (left), same with bucket zero negated (middle) and the multiplication table for the units of the quaternions (entries correspond to 'row unit' times 'column unit', right). The asterisks mark those entries where the sign is different from the scheme in the middle.

Quaternion multiplication can be achieved in eight real multiplication using the dyadic convolution (see section 10.6 on page 317): The scheme in figure 20.7 suggests to use the dyadic convolution with bucket zero negated as a starting point which costs 4 multiplications. Some entries have to be corrected then which costs four more multiplications.

```
// f[] == [ re1, i1, j1, k1 ]
// g[] == [ re2, i2, j2, k2 ]
c0 := f[0] * g[0]
c1 := f[3] * g[2]
c2 := f[1] * g[3]
c3 := f[2] * g[1]

// length-4 dyadic convolution:
```

```
walsh(f[]);
walsh(g[]);
for i:=0 to 3  g[i] := (f[i] * g[i])
walsh(g[]);

// normalization and correction:
g[0] :=   2 * c0 - g[0] / 4
g[1] := - 2 * c1 + g[1] / 4
g[2] := - 2 * c2 + g[2] / 4
g[3] := - 2 * c3 + g[3] / 4
```

The algorithm is taken from [62] which also gives a second variant.

The complex multiplication by three real multiplications corresponds to one length-2 Walsh dyadic convolution and the correction for the product of the imaginary units:

```
// f[] == [ re1, im1 ]
// g[] == [ re2, im2 ]
c0 := f[1] * g[1]  // == im1 * im2
// length-2 dyadic convolution:
{ f[0], f[1] } := { f[0]+f[1], f[0]-f[1] }
{ g[0], g[1] } := { g[0]+g[1], g[0]-g[1] }
g[0] := f[0] * g[0]
g[1] := f[1] * g[1]
{ g[0], g[1] } := { g[0]+g[1], g[0]-g[1] }
// normalization:
f[0] := f[0] / 2
g[0] := g[0] / 2
// correction:
g[0] := - 2 * c0 + g[0]
// here:  g[] == [ re1 * re2 - im1 * im2,  re1 * im2 + im1 * re2 ]
```

For complex numbers of high precision multiplication is asymptotically equivalent to two real multiplications as one FFT based (complex linear) convolution can be used for the computation. Similarly, high precision quaternion multiplication is as expensive as four real multiplications. Figure 20.8 shows an equivalent construction for the octonions leading to an algorithm with 30 multiplications. There are likely better schemes for octonion multiplication.

```
-0  1  2  3   4  5  6  7       0  1  2  3   4  5  6  7       #0  1  2  3   4  5  6  7
 1 -0  3  2   5  4  7  6       1 -0  3 -2   5 -4 -7  6        1  0  3 #2   5 #4 #7  6
 2  3 -0  1   6  7  4  5       2 -3 -0  1   6  7 -4 -5        2 #3  0  1   6  7 #4 #5
 3  2  1 -0   7  6  5  4       3  2 -1 -0   7 -6  5 -4        3  2 #1  0   7 #6  5 #4

 4  5  6  7  -0  1  2  3       4 -5 -6 -7  -0  1  2  3        4 #5 #6 #7   0  1  2  3
 5  4  7  6   1 -0  3  2       5  4 -7  6  -1 -0 -3  2        5  4 #7  6  #1  0 #3  2
 6  7  4  5   2  3 -0  1       6  7  4 -5  -2  3 -0 -1        6  7  4 #5  #2  3  0 #1
 7  6  5  4   3  2  1 -0       7 -6  5  4  -3 -2  1 -0        7 #6  5  4  #3 #2  1  0
```

Figure 20.8: Scheme for the length-8 dyadic convolution with bucket zero negated (left) and multiplication table for the octonions (middle, taken from [Num]). There are 22 places where the signs differ (right, marked with '#'). This leads to an algorithm involving $8 + 22 = 30$ multiplications.

Chapter 21

Arithmetic modulo m and some number theory

This chapter introduces some basic concepts of number theory. We start by implementing the arithmetical operations modulo m. Basic concepts of number theory, like the order of an element, quadratic residues and primitive roots are introduced. Selected algorithms such as the Rabin-Miller compositeness test and several primality tests are presented. The chapter ends with recreational material, such as multigrades and solutions of certain Diophantine equations.

Modular arithmetic and the concepts of number theory are fundamental for many areas like cryptography, error correcting codes, and signal processing.

21.1 Implementation of the arithmetic operations

The first part in the implementation of modular arithmetics consists of the implementing the operations addition, subtraction, multiplication, powering and division. However, for a practically useful C++ class one also has to implement several other routines like the determination of the order of an element (which involves integer factorization and computation of Euler's $\varphi(.)$) computation of square roots and elements of given order.

Addition and subtraction

Addition and subtraction modulo m can easily be implemented as [FXT: file mod/modarith.h]:

```
inline umod_t sub_mod(umod_t a, umod_t b, umod_t m)
{
    if ( a>=b )  return  a - b;
    else         return  m - b + a;
}

inline umod_t add_mod(umod_t a, umod_t b, umod_t m)
{
    if ( 0==b )  return  a;
    // return sub_mod(a, m-b, m);
    b = m - b;
    if ( a>=b )  return  a - b;
    else         return  m - b + a;
}
```

The type `umod_t` is an unsigned 64-bit integer. Care has been taken to avoid any overflow of intermediate results. A 'set'-, increment-, decrement- and negation function will further be useful:

```
inline umod_t set_mod(umod_t x, umod_t m)
{ if ( x>=m )  x %= m;  return x; }

inline umod_t incr_mod(umod_t a, umod_t m)
{ a++; if ( a==m )  a = 0;   return a; }

inline umod_t decr_mod(umod_t a, umod_t m)
{ if ( a==0 )  a = m - 1;  else a--;  return a; }

inline umod_t neg_mod(umod_t b, umod_t m)
{ if ( 0==b )  return 0;  else  return m - b; }
```

Two addition tables for the moduli 13 and 9 are shown in figure 21.1

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	6	7	8	9	10	11	12	0
2	2	3	4	5	6	7	8	9	10	11	12	0	1
3	3	4	5	6	7	8	9	10	11	12	0	1	2
4	4	5	6	7	8	9	10	11	12	0	1	2	3
5	5	6	7	8	9	10	11	12	0	1	2	3	4
6	6	7	8	9	10	11	12	0	1	2	3	4	5
7	7	8	9	10	11	12	0	1	2	3	4	5	6
8	8	9	10	11	12	0	1	2	3	4	5	6	7
9	9	10	11	12	0	1	2	3	4	5	6	7	8
10	10	11	12	0	1	2	3	4	5	6	7	8	9
11	11	12	0	1	2	3	4	5	6	7	8	9	10
12	12	0	1	2	3	4	5	6	7	8	9	10	11

	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8	0
2	2	3	4	5	6	7	8	0	1
3	3	4	5	6	7	8	0	1	2
4	4	5	6	7	8	0	1	2	3
5	5	6	7	8	0	1	2	3	4
6	6	7	8	0	1	2	3	4	5
7	7	8	0	1	2	3	4	5	6
8	8	0	1	2	3	4	5	6	7

Figure 21.1: Addition modulo 13 (left) and modulo 9 (right).

Multiplication

Multiplication is a bit harder: If one would use something like

```
inline umod_t mul_mod(umod_t a, umod_t b, umod_t m)
{
    return  (a * b) % m;
}
```

Then the modulus would be restricted to half of the word size.

One can use almost all bits for the modulus if the following trick is used. The technique also works if the product $a \cdot b$ does not fit into a machine integer.

Let $\langle x \rangle_y$ denote x modulo y, let $\lfloor x \rfloor$ denote the integer part of x. For $0 \leq a, b < m$:

$$a \cdot b \quad = \quad \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m + \langle a \cdot b \rangle_m \tag{21.1}$$

Rearranging and taking both sides modulo $z > m$ (where $z = 2^k$ on a k-bit machine):

$$\left\langle a \cdot b - \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m \right\rangle_z \quad = \quad \langle \langle a \cdot b \rangle_m \rangle_z \tag{21.2}$$

The right hand side equals $\langle a \cdot b \rangle_m$ because $m < z$.

$$\langle a \cdot b \rangle_m \quad = \quad \left\langle \langle a \cdot b \rangle_z - \left\langle \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m \right\rangle_z \right\rangle_z \tag{21.3}$$

The expression on the right can be translated into a few lines of C-code. The code given here assumes that one has 64-bit integer types `int64` (signed) and `uint64` (unsigned) and a floating point type with 64-bit mantissa, `float64` (typically `long double`).

```
uint64 mul_mod(uint64 a, uint64 b, uint64 m)
{
    uint64 y = (uint64)((float64)a*(float64)b/m+(float64)1/2);  // floor(a*b/m)
    y = y * m;           // m*floor(a*b/m) mod z
    uint64 x = a * b;    // a*b mod z
    uint64 r = x - y;    // a*b mod z - m*floor(a*b/m) mod z
    if ( (int64)r < 0 )  // normalization needed ?
    {
        r = r + m;
        y = y - 1;       // (a*b)/m  quotient, omit line if not needed
    }
    return  r;           // (a*b)%m  remnant
}
```

The technique uses the fact that integer multiplication computes the least significant bits of the result $\langle a \cdot b \rangle_z$ whereas float multiplication computes the most significant bits of the result. The above routine works if $0 <= a, b < m < 2^{63} = \frac{z}{2}$. The normalization is not necessary if $m < 2^{62} = \frac{z}{4}$.

When working with a fixed modulus the division by `p` may be replaced by a multiplication with the inverse modulus, that only needs to be computed once:

Precompute: `float64 i = (float64)1/m;`

and replace the line `uint64 y = (uint64)((float64)a*(float64)b/m+(float64)1/2);`

by `uint64 y = (uint64)((float64)a*(float64)b*i+(float64)1/2);`

so any division inside the routine is avoided. But beware, the routine then cannot be used for $m >= 2^{62}$: it very rarely fails for moduli of more than 62 bits. This is due to the additional error when inverting and multiplying as compared to dividing alone.

This trick is ascribed to Peter Montgomery.

The implementation in [FXT: file mod/modarith.h] is

```
inline umod_t mul_mod(umod_t a, umod_t b, umod_t m)
{
    umod_t x = a * b;
    umod_t y = m * (umod_t)( (ldouble)a * (ldouble)b/m + (ldouble)1/2 );
    umod_t r = x - y;
    if ( (smod_t)r < 0 )  r += m;
    return  r;
}
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12
2	0	2	4	6	8	10	12	1	3	5	7	9	11
3	0	3	6	9	12	2	5	8	11	1	4	7	10
4	0	4	8	12	3	7	11	2	6	10	1	5	9
5	0	5	10	2	7	12	4	9	1	6	11	3	8
6	0	6	12	5	11	4	10	3	9	2	8	1	7
7	0	7	1	8	2	9	3	10	4	11	5	12	6
8	0	8	3	11	6	1	9	4	12	7	2	10	5
9	0	9	5	1	10	6	2	11	7	3	12	8	4
10	0	10	7	4	1	11	8	5	2	12	9	6	3
11	0	11	9	7	5	3	1	12	10	8	6	4	2
12	0	12	11	10	9	8	7	6	5	4	3	2	1

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8
2	0	2	4	6	8	1	3	5	7
3	0	3	6	0	3	6	0	3	6
4	0	4	8	3	7	2	6	1	5
5	0	5	1	6	2	7	3	8	4
6	0	6	3	0	6	3	0	6	3
7	0	7	5	3	1	8	6	4	2
8	0	8	7	6	5	4	3	2	1

Figure 21.2: Multiplication modulo 13 (left) and modulo 9 (right).

Two multiplication tables for the moduli 13 and 9 are shown in figure 21.2. Note that for the modulus 9 some products $a \cdot b$ are zero though neither of a or b is zero. The tables can be obtained with [FXT: demo mod/modarithtables-demo.cc].

Exponentiation

The algorithm used for exponentiation (powering) is the binary exponentiation algorithm shown in section 14.6 on page 395:

```
inline umod_t pow_mod(umod_t a, umod_t e, umod_t m)
{
    if ( 0==e )  { return 1; }
    else
    {
        umod_t z = a;
        umod_t y = 1;
        while ( 1 )
        {
            if ( e&1 )  y = mul_mod(y, z, m);  // y *= z;
            e /= 2;
            if ( 0==e )  break;
            z = mul_mod(z, z, m);  // z *= z;
        }
        return  y;
    }
}
```

Inversion and division

Subtraction is the inverse of addition. In order to subtract an element b form another element a one can add the *additive inverse* $-b := m - b$ to a. Each element has an additive inverse.

Division is the inverse of multiplication. In order to divide an element a by another element b one can multiply a by the *multiplicative inverse* b^{-1}. The bad news is that not all elements have a multiplicative inverse.

In fact, only those elements b that are *coprime* to the modulus m ($\gcd(b, m) = 1$ or $b \perp m$) have inverses. Elements that have inverses are called *invertible* or *units*. For a prime modulus all elements except zero are invertible.

The computation of the GCD uses the *Euclidean algorithm* [FXT: file mod/gcd.h]:

```
template <typename Type>
Type gcd(Type a, Type b)
// Return greatest common divisor of a and b.
{
    if ( a < b )  swap2(a, b);
    if ( b==0 )  return a;
    Type r;
    do
    {
        r = a % b;
        a = b;
        b = r;
    }
    while ( r!=0 );
    return a;
}
```

A variant of the algorithm that avoids most of the (expensive) modular reductions is called the *binary GCD algorithm* [FXT: file mod/binarygcd.h]:

```
template <typename Type>
Type binary_ugcd(Type a, Type b)
// Return greatest common divisor of a and b.
// Version for unsigned types.
{
    if ( a < b )  swap2(a, b);
    if ( b==0 )  return a;

    Type r = a % b;
    a = b;
    b = r;
    if ( b==0 )  return a;

    ulong k = 0;
```

```
    while ( !((a|b)&1) )  // both even
    {
        k++;
        a >>= 1;
        b >>= 1;
    }
    while ( !(a&1) )  a >>= 1;
    while ( !(b&1) )  b >>= 1;
    while ( 1 )
    {
        if ( a==b )  return  a << k;
        if ( a < b )  swap2(a, b);
        Type t = (a-b) >> 1;  // t>0
        while ( !(t&1) )  t >>= 1;
        a = t;
    }
}
```

We note that the least common multiple (LCM) of two numbers can be computed as

$$\mathrm{lcm}(a,\, b) \quad = \quad \frac{a\, b}{gcd(a,\, b)} \tag{21.4}$$

With integer types of fixed size one can avoid overflow of the intermediate result $a\, b$ by computing the LCM as

$$\mathrm{lcm}(a,\, b) \quad = \quad \left[\frac{a}{gcd(a,\, b)}\right] b \tag{21.5}$$

For modular inversion one can use the extended Euclidean algorithm (EGCD), which for two integers a and b finds $d = \gcd(a, b)$ and u, v so that $a\, u + b\, v = d$. Applying EGCD to b and m where $b \perp m$ one gets u and v so that $m\, u + b\, v = 1$. That is: $b\, v \equiv 1 \pmod{m}$, so v is the inverse of b modulo m and $a/b := a\, b^{-1} = a\, v$.

The following code implements the EGCD algorithm as given in [DEK2]:

```
template <typename Type>
Type egcd(Type u, Type v, Type &tu1, Type &tu2)
// Return u3 and set u1,v1 so that
//   gcd(u,v) == u3 == u*u1 + v*u2
// Type must be a signed type.
{
    Type u1 = 1,  u2 = 0;
    Type v1 = 0,  v3 = v;
    Type u3 = u,  v2 = 1;
    while ( v3!=0 )
    {
        Type q = u3 / v3;
        Type t1 = u1 - v1 * q;
        u1 = v1;  v1 = t1;
        Type t3 = u3 - v3 * q;
        u3 = v3;  v3 = t3;
        Type t2 = u2 - v2 * q;
        u2 = v2;  v2 = t2;
    }
    tu1 = u1;  tu2 = u2;
    return u3;
}
```

Another algorithm for the computation of the modular inversion uses exponentiation. It is given only after the concept of the *order* of an element has been introduced (section 21.5 on page 529).

21.2 Modular reduction with structured primes

The modular reduction with Mersenne primes $p = 2^k - 1$ is especially easy: Let u and v be in the range $0 \le u, v < p = 2^k - 1$, then with the ordinary (non-reduced) product split as $u\,v = 2^k\,r + s$ (so $0 \le r, s < p = 2^k - 1$) the reduction is simply $u\,v \equiv r + s \pmod p$.

A modular reduction algorithm that uses only shifts, additions and subtractions can be found also for *structured primes* (sometimes called *generalized Mersenne primes*). Let

$$p \quad = \quad x^n + \sum_{i=0}^{n-1} d_i\,x^i \tag{21.6}$$

where $x = 2^k$, $d_i = \pm 1$ and $d_{n-1} = -1$. Then the reduction algorithm can be found using linear feedback shift registers (see section 6.1 on page 217): Let $0 \le u, v < p$ and write the non-reduced product as

$$u\,v \quad = \quad \sum_{i=0}^{2n-1} a_i\,x^i \tag{21.7}$$

and for $i \ge n$ express the a_i as linear combinations $a_i = \sum_{j=0}^{n-1} b_{i,j}\,a_j$ (where the $b_{i,j}$ equal ± 1). Then the reduced product

$$u\,v \quad \equiv \quad \sum_{i=0}^{n-1} A_i\,x^i \pmod p \tag{21.8}$$

(where $0 \le A_i < x$) can be read of the columns.

For example, with $p = 2^{64} - 2^{32} + 1 = x^2 - x + 1$ where $x = 2^{32}$ we have $u\,v = \sum_{i=0}^{5} a_i\,x^i$ and

```
a_i = (x^i mod x^2-x+1)

a_0 =     +1
a_1 =  +x
a_2 =  +x -1
a_3 =     -1
```

That is, with $u\,v \equiv \sum_{i=0}^{1} A_i \pmod p$ we have $A_0 = a_0 - a_2 - a_3$ and $A_1 = a_1 + a_2$. The first primes of the form $x^2 - x + 1$ with $x = 2^k$, $k < 1,000$ are

```
 k  p
 1  3
 2  13
 4  241
32  18446744069414584321
```

A second example, let $p = x^3 - x + 1$

```
a_i = (x^i mod x^3-x+1)

a_0 =           +1
a_1 =       +x
a_2 = +x^2
a_3 =       +x  -1
a_4 = +x^2  -x
a_5 = -x^2  +x  -1
```

And we get $A_0 = a_0 - a_3 - a_5$, $A_1 = a_1 + a_3 - a_4 + a_5$ and $A_2 = a_2 + a_4 - a_5$. The first few primes of the form $x^3 - x + 1$ with $x = 2^k$, $k < 1,000$ are

```
  k  p
  1  7
  2  61
  8  16776961
 17  2251799813554177
 32  7922816251426433759248983041
 33  63382530011411470073976166809 7
 68  25711008708143844408671393477458601640354952752619506011996161
 72  10531229166855718669791802768367043231889509067818262838466576384 1
105  66749594872528440074844428317798503581334516323645399060 84 \
     50502038795472233416763403230626447 37
141  ...
168  ...
558  ...
741  ...
```

```
2^64-2^32+1 == 2^(32*2)-2^(32*1)+1
2^96-2^32+1 == 2^(32*3)-2^(32*1)+1
2^224-2^96+1 == 2^(32*7)-2^(32*3)+1
2^320-2^288+1 == 2^(32*10)-2^(32*9)+1
2^512-2^32+1 == 2^(32*16)-2^(32*1)+1
2^512-2^288+1 == 2^(32*16)-2^(32*9)+1
2^544-2^32+1 == 2^(32*17)-2^(32*1)+1
2^544-2^96+1 == 2^(32*17)-2^(32*3)+1
2^576-2^512+1 == 2^(64*9)-2^(64*8)+1
2^672-2^192+1 == 2^(32*21)-2^(32*6)+1
2^832-2^448+1 == 2^(64*13)-2^(64*7)+1
2^992-2^832+1 == 2^(32*31)-2^(32*26)+1
2^1088-2^608+1 == 2^(32*34)-2^(32*19)+1
2^1184-2^768+1 == 2^(32*37)-2^(32*24)+1
2^1376-2^32+1 == 2^(32*43)-2^(32*1)+1
2^1664-2^256+1 == 2^(128*13)-2^(128*2)+1
2^1856-2^1056+1 == 2^(32*58)-2^(32*33)+1
2^1920-2^384+1 == 2^(128*15)-2^(128*3)+1
2^1984-2^544+1 == 2^(32*62)-2^(32*17)+1
```

```
2^80-2^48+1 == 2^(16*5)-2^(16*3)+1
2^176-2^48+1 == 2^(16*11)-2^(16*3)+1
2^176-2^80+1 == 2^(16*11)-2^(16*5)+1
2^368-2^336+1 == 2^(16*23)-2^(16*21)+1
2^384-2^80+1 == 2^(16*24)-2^(16*5)+1
2^400-2^160+1 == 2^(16*25)-2^(16*10)+1
2^528-2^336+1 == 2^(16*33)-2^(16*21)+1
2^544-2^304+1 == 2^(16*34)-2^(16*19)+1
2^560-2^112+1 == 2^(16*35)-2^(16*7)+1
2^576-2^240+1 == 2^(16*36)-2^(16*15)+1
2^672-2^560+1 == 2^(16*42)-2^(16*35)+1
2^688-2^96+1 == 2^(16*43)-2^(16*6)+1
2^784-2^48+1 == 2^(16*49)-2^(16*3)+1
2^832-2^432+1 == 2^(16*52)-2^(16*27)+1
2^880-2^368+1 == 2^(16*55)-2^(16*23)+1
2^912-2^32+1 == 2^(16*57)-2^(16*2)+1
2^944-2^784+1 == 2^(16*59)-2^(16*49)+1
2^1008-2^144+1 == 2^(16*63)-2^(16*9)+1
2^1024-2^880+1 == 2^(16*64)-2^(16*55)+1
```

Figure 21.3: The complete list of primes of the form $p = x^k - x^j + 1$ where $x = 2^G$, $G = 2^i$, $G \geq 32$ and p up to 2048 bits (left) and the equivalent list for $x = 2^{16}$ (and p up to 1024 bits, right).

The second example is from the paper [97] were the reader is referred for further studies.

The multiplication rules and corresponding primes can be found using a few lines of pari/gp code:

```
pp(x)=x^3-x+1
p=pp(x)
n=poldegree(p)
{ for (i=0, 2*n-1,
    z=(x^i) % p;
    print("a_"i, " = ", z);
  ); }
/* find primes: */
for(j=1,100, p=pp(2^j); if(isprime(p), print(j, "  ",p)))
```

The reduction rule for primes of the form $x^3 - x^2 + 1$ is:

```
a_i = (x^i mod x^3-x^2+1)
a_0 =         +1
a_1 =      +x
a_2 = +x^2
a_3 = +x^2    -1
a_4 = +x^2 -x -1
a_5 =      -x -1
```

We get $A_0 = a_0 - a_3 - a_4 - a_5$, $A_1 = a_1 - a_4 + a_5$ and $A_2 = a_2 + a_3 + a_4$. The first few primes of the form $x^3 - x^2 + 1$ with $x = 2^k$, $k < 1,000$ are

```
  k  p
  1  5
  3  449
 19  144114913197948929
 34  5070602400617769700807459995649
 51  114179815416476739778638868426783551042491514 89
318  ...
```

A list of primes of the form $p = x^k - x^j + 1$ where $x = 2^{2^i}$ is shown in figure 21.3.

21.3 The sieve of Eratosthenes

Several number theoretic algorithms can take advantage of a precomputed list of primes. A simple and quite efficient algorithm, called the *sieve of Eratosthenes* computes all primes up to a given limit. It uses a tag-array where all entries ≥ 2 are initially marked as potential primes. The algorithm proceeds by searching for the next marked entry and deleting all multiples of it.

An implementation that uses the `bitarray` class (see section 4.6 on page 145) is given in [FXT: demo mod/eratosthenes-demo.cc]:

```
void
eratosthenes(bitarray &ba)
{
    ba.set_all();
    ba.clear(0);
    ba.clear(1);
    ulong n = ba.n_;
    ulong k = 0;
    while ( (k=ba.next_set(k+1)) < n )
    {
        for (ulong j=2, i=j*k;  i<n;  ++j, i=j*k)  ba.clear(i);
    }
}
```

The program prints the resulting list of primes (code slightly simplified):

```
int
main(int argc, char **argv)
{
    ulong n = 100;
    NXARG(n, "Upper limit for prime search");
    bitarray ba(n);
    eratosthenes(ba);
    ulong k = 0;
    ulong ct = 0;
    while ( (k=ba.next_set(k+1)) < n )
    {
        ++ct;
        cout << " " << k;
    }
    cout << endl;
    cout << "Found " << ct << " primes below " << n << "." << endl;
    return 0;
}
```

The output for the default ($n = 100$) is:

```
 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Found 25 primes below 100.
```

A little thought leads to a faster variant: When deleting the multiples $k \cdot p$ of the prime p from the list we only need to care about the values of k that are greater than all primes found so far. Further, values $k \cdot p$ containing only prime factors smaller than p have already been deleted. That is, we only need to delete the values $\{p^2, p^2 + p, p^2 + 2p, p^2 + 3p, \ldots\}$. If we further extract the loop for the prime 2 then for the odd primes, we need to delete only the values $\{p^2, p^2 + 2p, p^2 + 4p, \ldots\}$.

The implementation is

```
void
eratosthenes_opt(bitarray &ba)
{
    ba.set_all();
    ba.clear(0);
    ba.clear(1);
    ulong n = ba.n_;
    for (ulong k=4; k<n; k+=2)  ba.clear(k);
    ulong r = isqrt(n);
    ulong k = 0;
    while ( (k=ba.next_set(k+1)) < n )
    {
        if ( k > r )  break;
        for (ulong j=k*k; j<n; j+=k*2)  ba.clear(j);
    }
}
```

The routine is included in the demo, the second argument chooses whether the optimized routine is used.

When computing the primes up to a limit N then about N/p values are deleted after finding the prime p. If we slightly overestimate the computational work W by

$$W \approx N \sum_{\substack{p \text{ prime} \\ p<N}} \frac{1}{p} \tag{21.9}$$

then we get $W \approx N \log(\log(N))$ which is linear in very good approximation. Practically, much of the time used with larger values of N is lost waiting for memory access. Thereby, further improvements should rather address machines specific optimizations than additional algorithmic refinements.

The described algorithmic improvement can be deduced from the series acceleration of the Lambert series

$$\sum_{k>k} d(k)\, x^k \quad = \quad \sum_{k>0} \frac{x^k}{1-x^k} \quad = \quad \sum_{k>0} \frac{1+x^k}{1-x^k}\, x^{k^2} \tag{21.10}$$

where $d(k)$ is the sum of the divisors of k.

One can save half of the space by recording only the odd primes. A C++ implementation of the modified algorithm is [FXT: `make_oddprime_bitarray()` in mod/eratosthenes.cc]. The corresponding table is created upon startup of programs linking the FXT-library. The data can be used to verify the primality of small numbers [FXT: `is_small_prime()` in mod/primes.cc]. The function `next_small_prime()` in the same file uses the data to return the next prime greater or equal to its argument or zero if the argument is too big.

Further optimizations and an asymptotically slightly faster algorithm for the generation of primes is given in [7].

21.4 The Chinese Remainder Theorem (CRT)

Let m_1, m_2, ..., m_f be pairwise coprime (that is, $\gcd(m_i, m_j) = 1$ for all $i \neq j$). If $x \equiv x_i \pmod{m_i}$ for $i = 1, 2, \ldots, f$ then x is unique modulo the product $M = m_1 \cdot m_2 \cdots m_f$. This is the *Chinese remainder theorem* (CRT). Note that it is not assumed that any of the m_i is prime.

The theorem tells us that is we want to do computations modulo a composite number M we can instead do all computations (separately) modulo the coprime factors of M and recover the result only at the end (see below). For example, when computing the exact convolution of a long sequence via number theoretic transforms (NTTs, see section 13.3 on page 385) the largest term of the result must be smaller than the modulus. If efficient modular arithmetic is available for moduli of a maximal size only the convolution cannot be computed directly. Now choose several coprime moduli whose product M is greater than the largest element of the result, compute the NTTs separately and only at the very end compute the elements modulo M.

Efficient computation of x

For two moduli m_1, m_2 compute x with $x \equiv x_1 \pmod{m_1}$ and $x \equiv x_2 \pmod{m_2}$ as suggested by the following pseudo code:

```
function crt2(x1, m1, x2, m2)
{
    c := m1**(-1) mod m2    // inverse of m1 modulo m2
    s := ((x2-x1)*c) mod m2
    return  x1 + s * m1
}
```

For repeated CRT calculations with the same moduli one will use precomputed $c = m_1^{-1} \bmod m_2$.

For more than two moduli use the above algorithm repeatedly. Pseudo code to perform the CRT for several moduli:

```
function crt(x[0,...,f-1], m[0,...,f-1], f)
{
    x1 := x[0]
    m1 := m[0]
    i := 1
    do
```

```
    {
        x2 := x[i]
        m2 := m[i]
        x1 := crt2(x1, m1, x2, m2)
        m1 := m1 * m2
        i := i + 1
    }
    while i < f
    return x1
}
```

Idea of proof

To see why these functions really work we do formulate a more general CRT procedure that specializes to the functions above. Define

$$T_i \quad := \quad \prod_{k \neq i} m_k \tag{21.11}$$

and

$$c_i \quad := \quad T_i^{-1} \mod m_i \tag{21.12}$$

Then for

$$X_i \quad := \quad x_i \, c_i \, T_i \tag{21.13}$$

one has

$$X_i \mod m_j \quad = \quad \begin{cases} x_i & \text{for} \quad j = i \\ 0 & \text{else} \end{cases} \tag{21.14}$$

Therefore

$$\sum_k X_k \quad = \quad x_i \mod m_i \tag{21.15}$$

For the special case of two moduli m_1, m_2 one has

$$T_1 \quad = \quad m_2 \tag{21.16a}$$
$$T_2 \quad = \quad m_1 \tag{21.16b}$$
$$c_1 \quad = \quad m_2^{-1} \mod m_1 \tag{21.16c}$$
$$c_2 \quad = \quad m_1^{-1} \mod m_2 \tag{21.16d}$$

The quantities are related by

$$c_1 \, m_2 + c_2 \, m_1 \quad = \quad 1 \tag{21.17a}$$
$$\sum_k X_k \quad = \quad x_1 \, c_1 \, T_1 + x_2 \, c_2 \, T_2 \tag{21.17b}$$
$$= \quad x_1 \, c_1 \, m_2 + x_2 \, c_2 \, m_1 \tag{21.17c}$$
$$= \quad x_1 \, (1 - c_2 \, m_1) + x_2 \, c_2 \, m_1 \tag{21.17d}$$
$$= \quad x_1 + (x_2 - x_1) \, (m_1^{-1} \mod m_2) \, m_1 \tag{21.17e}$$

as given in the code.

	0	1	2	3	4	5	6	7	8	9	10	11	12	<--= exponent
														[order]
0	1	0	0	0	0	0	0	0	0	0	0	0	0	[--]
1	1	1	1	1	1	1	1	1	1	1	1	1	1	[1]
2	1	2	4	8	3	6	12	11	9	5	10	7	1	[12]
3	1	3	9	1	3	9	1	3	9	1	3	9	1	[3]
4	1	4	3	12	9	10	1	4	3	12	9	10	1	[6]
5	1	5	12	8	1	5	12	8	1	5	12	8	1	[4]
6	1	6	10	8	9	2	12	7	3	5	4	11	1	[12]
7	1	7	10	5	9	11	12	6	3	8	4	2	1	[12]
8	1	8	12	5	1	8	12	5	1	8	12	5	1	[4]
9	1	9	3	1	9	3	1	9	3	1	9	3	1	[3]
10	1	10	9	12	3	4	1	10	9	12	3	4	1	[6]
11	1	11	4	5	3	7	12	2	9	8	10	6	1	[12]
12	1	12	1	12	1	12	1	12	1	12	1	12	1	[2]

Figure 21.4: Powers and orders modulo 13, the maximal order is $R(13) = 12 = \varphi(13)$.

	0	1	2	3	4	5	6	7	8	<--= exponent
										[order]
0	1	0	0	0	0	0	0	0	0	[--]
1	1	1	1	1	1	1	1	1	1	[1]
2	1	2	4	8	7	5	1	2	4	[6]
3	1	3	0	0	0	0	0	0	0	[--]
4	1	4	7	1	4	7	1	4	7	[3]
5	1	5	7	8	4	2	1	5	7	[6]
6	1	6	0	0	0	0	0	0	0	[--]
7	1	7	4	1	7	4	1	7	4	[3]
8	1	8	1	8	1	8	1	8	1	[2]

	0	1	2	3	4	5	6	<--= exponent
								[order]
1	1	1	1	1	1	1	1	[1]
2	1	2	4	8	7	5	1	[6]
4	1	4	7	1	4	7	1	[3]
5	1	5	7	8	4	2	1	[6]
7	1	7	4	1	7	4	1	[3]
8	1	8	1	8	1	8	1	[2]

Figure 21.5: Powers and orders modulo 9 (left), the maximal order is $R(9) = 6 = \varphi(9)$. The order modulo m is defined only for elements a where $\gcd(a, m) = 1$. The table of powers for the group of units $(\mathbb{Z}/9\mathbb{Z})^*$ (right) is obtained by dropping all elements for which the order is undefined.

21.5 The order of an element

The *order* $r = \mathrm{ord}(a)$ of an element a is the smallest (nonzero) exponent so that $a^r = 1$. For elements that are not invertible ($\gcd(a, m) \neq 1$) the order is not defined. Figure 21.4 shows the powers of all elements modulo the prime 13. The rightmost column gives the order of those elements that are invertible.

An element a whose r-th power equals one is called an *r-th root of unity*: $a^r = 1$. Modulo 9 both elements 2 and 4 are a 6th roots of unity (figure 21.5).

If $a^r = 1$ but $a^x \neq 1$ for all $x < r$ then a is called a *primitive r-th root of unity*. Modulo 9 the element 2 is a primitive 6th root of unity; the element 4 is not, it is a primitive 3rd root of unity. An element of order r is an r-th primitive root of unity.

The *maximal order* $R(m)$ is the maximum of the orders of all elements for a fixed modulus m:

$$R(m) \quad := \quad \max_{a \in \mathbb{Z}/m\mathbb{Z}^*} (\mathrm{ord}(a)) \tag{21.18}$$

For prime modulus p the maximal order equals $R(p) = p - 1$. When it cannot cause confusion we omit the argument to the maximal order in what follows.

Roots of unity of an order different from R are available only for the divisors d_i of R: With g an element

of maximal order g^{R/d_i} is a d_i-th root of unity:

$$\operatorname{ord}\left(g^{R/d_i}\right) \quad = \quad d_i \tag{21.19}$$

This is because $(g^{R/d_i})^{d_i} = g^R = 1$.

The concept of the order comes from group theory. The invertible elements modulo m with multiplication form a group. The neutral element is one. The order defined above is the order in this group, it tells us how often one has to multiply the element a to one to obtain one. We restrict orders to positive values, else every element would have order zero.

With addition things are simpler: All elements with addition from a group with zero as neutral element. The order of an element a in this group tells us how often one has to add a to zero to obtain zero. The order here is simply $m/\gcd(a, m)$.

The maximal order of a group is sometimes called the *exponent of the group.*

21.6 Prime modulus: the field $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p = GF(p)$

If the modulus is a prime p then $\mathbb{Z}/p\mathbb{Z}$ is the field $\mathbb{F}_p = GF(p)$: All elements except 0 have inverses and thereby division is possible in $\mathbb{F}_p$.

The maximal order R equals $p - 1$. Roots of unity are available for the order R and its divisors: The elements of maximal order in $\mathbb{Z}/p\mathbb{Z}$ are called *primitive roots* (sometimes also: *primitive elements*) or *generators* modulo p.

If g is a generator, then every element in $\mathbb{F}_p$ different from 0 is equal to some power g^e $(1 \le e < p)$ of g and its order is R/e. To test whether g is a primitive n-th root of unity in $\mathbb{F}_p$ one does not need to check whether $g^k \neq 1$ for all $k < n$. It suffices to do the check for exponents k that are prime factors of n. This is because the order of any element divides the maximal order.

To find a primitive root in $\mathbb{F}_p$ proceed as indicated by the following pseudo code:

```
function primroot(p)
{
    if p==2 then  return 1
    f[] := distinct_prime_factors(p-1)
    for r:=2 to p-1
    {
        x := TRUE
        foreach q in f[]
        {
            if r**((p-1)/q)==1 then x:=FALSE
        }
        if x==TRUE then  return r
    }
    error("no primitive root found")  // p cannot be prime !
}
```

The algorithm is a simple search and might seem ineffective. In practice the root is found after only a few tries.

An element of order n in $\mathbb{F}_p$ is returned by this function:

```
function element_of_order(n, p)
{
    R := p-1  // maxorder
    if (R/n)*n != R then  error("order n must divide maxorder p-1")
    r := primroot(p)
    x := r**(R/n)
    return x
}
```

21.7 Composite modulus: the ring $\mathbb{Z}/m\mathbb{Z}$

In what follows we will need the function $\varphi(.)$, the *totient function* (or *Euler's totient function*). The function $\varphi(m)$ counts the number of integers coprime to and less than m:

$$\varphi(m) \quad := \quad \sum_{\substack{1\le k<m \\ \gcd(k,m)=1}} 1 \tag{21.20}$$

For $m = p$ prime one has $\varphi(p) = p - 1$. For m composite $\varphi(m)$ is always less than $m - 1$. For $m = p^k$ a prime power

$$\varphi(p^k) \quad = \quad p^k - p^{k-1} \; = \; p^{k-1}\,(p-1) \tag{21.21}$$

The totient function is a *multiplicative* function: One has $\varphi(p_1\,p_2) = \varphi(p_1)\,\varphi(p_2)$ for *coprime* p_1, p_2: $\gcd(p_1, p_2) = 1$ but p_1, p_2 are not necessarily prime. Thereby, if the factorization of n into distinct prime powers is $n = \prod_i p_i^{e_i}$, then

$$\varphi(n) \quad = \quad \prod_i \varphi(p_i^{e_i}) \tag{21.22}$$

An alternative expression for $\varphi(n)$ where is

$$\varphi(n) \quad = \quad n \prod_{p_k} \left(1 - \frac{1}{p_k}\right) \qquad \text{where} \quad n = \prod_k p_k^{e_k} \tag{21.23}$$

The expression nicely generalizes as follows: The number of s-element sets of numbers $\le n$ whose greatest common divisor is coprime to n is

$$\varphi_s(n) \quad = \quad n^s \prod_{p_k} \left(1 - \frac{1}{p_k^s}\right) \qquad \text{where} \quad n = \prod_k p_k^{e_k} \tag{21.24}$$

For $s = 1$ one gets relation 21.23.

Pseudo code to compute $\varphi(m)$ for arbitrary m:

```
function phi(m)
{
    {n, p[], x[]} := factorization(m)  // m==product(i=0..n-1,p[i]**x[i])
    ph := 1
    for i:=0 to n-1
    {
        ph := ph * phi_pp(p[i], x[i])
    }
}
```

Where the function `phi_pp(p, x)` shall return $\varphi(p^x)$.

Further we need the notion of $(\mathbb{Z}/m\mathbb{Z})^*$, the group of units in $\mathbb{Z}/m\mathbb{Z}$. The number of units is

$$\left|(\mathbb{Z}/m\mathbb{Z})^*\right| \quad = \quad \varphi(m) \tag{21.25}$$

If m factorizes as $m = 2^{k_0} \cdot p_1^{k_1} \cdot \ldots \cdot p_q^{k_q}$ where p_i are pairwise distinct primes then

$$\left|(\mathbb{Z}/m\mathbb{Z})^*\right| \quad = \quad \varphi(2^{k_0}) \cdot \varphi(p_1^{k_1}) \cdot \ldots \cdot \varphi(p_q^{k_q}) \tag{21.26}$$

Further, $(\mathbb{Z}/m\mathbb{Z})^*$ is isomorphic to the direct product $\mathbb{Z}/2^{k_0}\mathbb{Z} \times \mathbb{Z}/p_1^{k_1}\mathbb{Z} \times \cdots \times \mathbb{Z}/p_q^{k_q}\mathbb{Z}$.

The sequence of the values $\varphi(n)$ is entry A000010 in [OEIS]. It starts as

n:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$\varphi(n)$:	1	1	2	2	4	2	6	4	6	4	10	4	12	6	8	8	16	6	18	8

21.7.1 Cyclic and noncyclic rings

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	[--]
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	[1]
2	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	[4]
3	1	3	9	12	6	3	9	12	6	3	9	12	6	3	9	[--]
4	1	4	1	4	1	4	1	4	1	4	1	4	1	4	1	[2]
5	1	5	10	5	10	5	10	5	10	5	10	5	10	5	10	[--]
6	1	6	6	6	6	6	6	6	6	6	6	6	6	6	6	[--]
7	1	7	4	13	1	7	4	13	1	7	4	13	1	7	4	[4]
8	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	[4]
9	1	9	6	9	6	9	6	9	6	9	6	9	6	9	6	[--]
10	1	10	10	10	10	10	10	10	10	10	10	10	10	10	10	[--]
11	1	11	1	11	1	11	1	11	1	11	1	11	1	11	1	[2]
12	1	12	9	3	6	12	9	3	6	12	9	3	6	12	9	[--]
13	1	13	4	7	1	13	4	7	1	13	4	7	1	13	4	[4]
14	1	14	1	14	1	14	1	14	1	14	1	14	1	14	1	[2]

	0	1	2	3	4	
1	1	1	1	1	1	[1]
2	1	2	4	8	1	[4]
4	1	4	1	4	1	[2]
7	1	7	4	13	1	[4]
8	1	8	4	2	1	[4]
11	1	11	1	11	1	[2]
13	1	13	4	7	1	[4]
14	1	14	1	14	1	[2]

Figure 21.6: Powers and orders modulo 15 (left). The ring $\mathbb{Z}/15\mathbb{Z}$ is noncyclic: There are $\varphi(15) = 8$ invertible elements but no element generates all of them as the maximal order is $R(15) = 4 < \varphi(15)$. The table of powers for the group of units $(\mathbb{Z}/15\mathbb{Z})^*$ (right) is obtained by dropping all elements for which the order is undefined.

It turns out that the maximal order R can be equal to or less than $|(\mathbb{Z}/m\mathbb{Z})^*|$, the ring $(\mathbb{Z}/m\mathbb{Z})^*$ is then called *cyclic* or *noncyclic*, respectively. The term 'cyclic' reflects that in those rings the powers of any element of maximal order 'cycle through' all elements of $(\mathbb{Z}/m\mathbb{Z})^*$. An element of maximal order in cyclic rings is called a *generator* as its powers 'generate' all elements. Strictly spoken, the terms cyclic and noncyclic refer to the group of invertible elements (with multiplication as group operation). As the additive group (of all elements, with addition as group operation) is always cyclic (one is always a generator) there is no ambiguity when speaking of a cyclic ring.

Figure 21.6 shows the powers and orders of the noncyclic ring $\mathbb{Z}/15\mathbb{Z}$ where no element generates all units. The rings $\mathbb{Z}/13\mathbb{Z}$ and $\mathbb{Z}/9\mathbb{Z}$ are cyclic, see figure 21.4 on page 529 and 21.5 on page 529.

For prime modulus $m = p$ the group $(\mathbb{Z}/m\mathbb{Z})^*$ contains all nonzero elements and any element of maximal order is a generator of the group.

For m a power of an odd prime p the maximal order R in $(\mathbb{Z}/m\mathbb{Z})^*$ (and also in $\mathbb{Z}/m\mathbb{Z}$) is

$$R(p^k) \quad = \quad \varphi(p^k) \tag{21.27}$$

while for m a power of two a tiny irregularity enters:

$$R(2^k) \quad = \quad \begin{cases} 1 & \text{for } k = 1 \\ 2 & \text{for } k = 2 \\ 2^{k-2} & \text{for } k \geq 3 \end{cases} \tag{21.28}$$

That is, for powers of two greater than 4 the maximal order deviates from $\varphi(2^k) = 2^{k-1}$ by a factor of 2. For the general modulus $m = 2^{k_0} \cdot p_1^{k_1} \cdot \ldots \cdot p_q^{k_q}$ the maximal order is

$$R(m) \quad = \quad \operatorname{lcm}\Big(R(2^{k_0}),\, R(p_1^{k_1}),\, \ldots,\, R(p_q^{k_q})\Big) \tag{21.29}$$

where lcm(.) denotes the least common multiple. Pseudo code to compute $R(m)$, the maximal order of an element in $\mathbb{Z}/m\mathbb{Z}$:

```
function maxorder(m)
{
    {n, p[], k[]} := factorization(m)  // m==product(i=0..n-1,p[i]**k[i])
    R := 1
    for i:=0 to n-1
    {
        t := phi_pp(p[i],k[i])  // ==phi(p[i]**k[i])
        if p[i]==2 AND k[i]>=3 then  t := t / 2
        R := lcm(R,t)
    }
    return R
}
```

Now we can see for which moduli m the ring $(\mathbb{Z}/m\mathbb{Z})^*$ will be cyclic: the maximal order must equal $\varphi(m)$, thereby

$$(\mathbb{Z}/m\mathbb{Z})^* \quad \text{is cyclic for} \quad m = 2,\ 4,\ p^k,\ 2 \cdot p^k \qquad \text{where } p \text{ is an odd prime} \tag{21.30}$$

If the factorization of m contains two different odd primes p_a and p_b then

$$R(m) \quad = \quad \text{lcm}(\dots, \varphi(p_a), \dots, \varphi(p_b), \dots)$$

is at least by a factor of two smaller than

$$\varphi(m) \quad = \quad \dots \cdot \varphi(p_a) \cdot \dots \cdot \varphi(p_b) \cdot \dots$$

because both $\varphi(p_a)$ and $\varphi(p_b)$ are even. Thereby $(\mathbb{Z}/m\mathbb{Z})^*$ cannot be cyclic in that case. The same argument holds for $m = 2^{k_0} \cdot p^k$ if $k_0 > 1$. For $m = 2^k$ the ring $(\mathbb{Z}/m\mathbb{Z})^*$ is cyclic only for $k = 1$ and $k = 2$ because of the above mentioned (relation 21.28 on the facing page) irregularity of $R(2^k)$.

Pseudo code for a function that returns the order of a given element x in $(\mathbb{Z}/m\mathbb{Z})^*$:

```
function order(x, m)
{
    if gcd(x,m)!=1 then  return 0  // x not a unit
    h := phi(m)  // number of elements of ring of units
    e := h
    {n, p[], k[]} := factorization(h)  // h==product(i=0..n-1,p[i]**k[i])
    for i:=0 to n-1
    {
        f := p[i]**k[i]
        e := e / f
        g1 := x**e mod m
        while g1!=1
        {
            g1 := g1**p[i] mod m
            e := e * p[i]
        }
    }
    return e
}
```

Pseudo code for a function that returns an element x in $(\mathbb{Z}/m\mathbb{Z})^*$ of maximal order:

```
function maxorder_element(m)
{
    R := maxorder(m)
    for x:=1 to m-1
    {
        if order(x, m)==R then  return x
    }
    // never reached
}
```

Again, while the function does a simple search it is efficient in practice. For prime m the function returns a primitive root. A C++ implementation is [FXT: `maxorder_element_mod()` in mod/maxorder.cc]. Note that for noncyclic rings the returned element does not necessarily have maximal order modulo all factors of the modulus. We list all elements of $(\mathbb{Z}/15\mathbb{Z})^*$ together with their orders modulo 15, 3, and 5:

```
 1:  r=1  r3=1  r5=1
 2:  r=4  r3=2  r5=4
 4:  r=2  r3=1  r5=2
```

```
  7:  r=4  r3=1  r5=4  <--=
  8:  r=4  r3=2  r5=4
 11:  r=2  r3=2  r5=1
 13:  r=4  r3=1  r5=4  <--=
 14:  r=2  r3=2  r5=2
```

The two elements marked with an arrow have maximal order modulo 15 but not modulo 3.

21.7.2 Generators in cyclic rings

Let G be the set of all generators in a cyclic ring modulo n. Then the number of generators is given by

$$|G| \quad = \quad \varphi(\varphi(n)) \tag{21.31}$$

Let g be a generator, the g^k is a generator exactly if $\gcd(k, \varphi(n)) = 1$.

Let g be a generator modulo a prime p. Then g is a generator modulo $2\,p^k$ for all $k \geq 1$ if g is odd. If g is even then $g + p^k$ is a generator modulo $2\,p^k$.

Further, g is a generator modulo p^k if $g^{p-1} \bmod p^2 \neq 1$. The only primes below $2^{36} \approx 68 \cdot 10^9$ for which the smallest primitive root is not the a generator modulo p^2 are 2, 40487 and 6692367337. Such primes are called *non-generous primes*, see sequence A055578 of [OEIS].

The only known primes p below $32 \cdot 10^{12}$ where $2^{p-1} = 1$ modulo p^2 are 1093 and 3511 (such primes are called *Wieferich primes*). As 2 is not a generator modulo either of the two we see that whenever 2 is a generator modulo $p < 32 \cdot 10^{12}$ then it is also a generator modulo p^k.

21.7.3 Generators in noncyclic rings

When the ring is cyclic an element of maximal order generates all invertible elements. With noncyclic rings one needs more than one generator. Pari/gp's function `znstar()` gives the complete information about the multiplicative group of units. The help text reads:

```
znstar(n): 3-component vector v, giving the structure of (Z/nZ)^*.
 v[1] is the order (i.e. eulerphi(n)),
 v[2] is a vector of cyclic components, and
 v[3] is a vector giving the corresponding  generators.
```

For $2 \leq n \leq 25$ we obtain

```
? for(n=2,25,print(n," ",znstar(n)))
    2 [1,  [],   []]    /* read:  [1, [1], [Mod(1,2)]] */
    3 [2,  [2],  [Mod(2, 3)]]
    4 [2,  [2],  [Mod(3, 4)]]
    5 [4,  [4],  [Mod(2, 5)]]
    6 [2,  [2],  [Mod(5, 6)]]
    7 [6,  [6],  [Mod(3, 7)]]
    8 [4,  [2, 2],  [Mod(5, 8), Mod(3, 8)]]
    9 [6,  [6],  [Mod(2, 9)]]
   10 [4,  [4],  [Mod(7, 10)]]
   11 [10, [10],  [Mod(2, 11)]]
   12 [4,  [2, 2],  [Mod(7, 12), Mod(5, 12)]]
   13 [12, [12],  [Mod(2, 13)]]
   14 [6,  [6],  [Mod(3, 14)]]
   15 [8,  [4, 2],  [Mod(8, 15), Mod(11, 15)]]
   16 [8,  [4, 2],  [Mod(5, 16), Mod(7, 16)]]
   17 [16, [16],  [Mod(3, 17)]]
   18 [6,  [6],  [Mod(11, 18)]]
   19 [18, [18],  [Mod(2, 19)]]
   20 [8,  [4, 2],  [Mod(3, 20), Mod(11, 20)]]
   21 [12, [6, 2],  [Mod(5, 21), Mod(8, 21)]]
   22 [10, [10],  [Mod(13, 22)]]
   23 [22, [22],  [Mod(5, 23)]]
   24 [8,  [2, 2, 2],  [Mod(13, 24), Mod(19, 24), Mod(17, 24)]]
   25 [20, [20],  [Mod(2, 25)]]
```

The ring is cyclic when there is just one generator. In general, when `znstar(n)` gives $[\varphi, [r_1, r_2, \ldots, r_k], [g_1, g_2, \ldots, g_k]]$ then the φ invertible elements u are of the form

$$u \quad = \quad g_1^{e_1}\, g_2^{e_2}\, \ldots\, g_k^{e_k} \tag{21.32}$$

where $0 \le e_i < r_i$ for $1 \le i \le k$. For example, with $n = 15$:

```
? znstar(15)
  [8, [4, 2], [Mod(8, 15), Mod(11, 15)]]
? g1=Mod(8, 15); g2=Mod(11,15);
? for(e1=0,4-1,for(e2=0,2-1,print(e1," ",e2,"  ",g1^e1*g2^e2)))
  0 0  Mod(1, 15)
  0 1  Mod(11, 15)
  1 0  Mod(8, 15)
  1 1  Mod(13, 15)
  2 0  Mod(4, 15)
  2 1  Mod(14, 15)
  3 0  Mod(2, 15)
  3 1  Mod(7, 15)
```

Generators modulo $n = 2^k$

The ring modulo $n = 2^k$ is cyclic only for for $k \le 2$:

```
? for(i=1,6,print(i,": ",znstar(2^i)))
   1: [1, [], []]
   2: [2, [2], [Mod(3, 4)]]
   3: [4, [2, 2], [Mod(5, 8), Mod(3, 8)]]
   4: [8, [4, 2], [Mod(5, 16), Mod(7, 16)]]
   5: [16, [8, 2], [Mod(5, 32), Mod(15, 32)]]
   6: [32, [16, 2], [Mod(5, 64), Mod(31, 64)]]
```

For $k \ge 3$ the multiplicative group is generated by the two elements 5 and -1.

Inversion by exponentiation

For a unit u of order $r = \mathrm{ord}(u)$ one has $u^r = 1$. As r divides the maximal order R also $u^R = 1$ holds and thereby $u^{R-1} \cdot u = 1$. That is, the inverse of any invertible element u equals u to the $(R-1)$-st power:

$$u^{-1} \quad = \quad u^{R-1} \tag{21.33}$$

In fact, one has also $u^{-1} = u^{\varphi(m)-1}$ which involves slightly more work if the ring is noncyclic.

21.8 Quadratic residues

Let p be a prime. The *quadratic residues modulo p* are those values a so that the equation

$$x^2 \quad \equiv \quad a \qquad (\mathrm{mod}\ p) \tag{21.34}$$

has a solution. If the equation has no solution then a is called a *quadratic non-residue modulo p*. A quadratic residue is a square (modulo p) of some number, so we can safely just call it a *square modulo p*.

Let g be a[1] primitive root, then every nonzero element x can uniquely be written as $x = g^e$ where $0 < e < p$. Rewriting equation 21.34 as $x^2 = (g^e)^2 = g^{2e} = a$ it is apparent that the quadratic residues are the even powers of g. The non-residues are the odd powers of g. All generators are non-residues.

Let us compute $f(x) := x^{(p-1)/2}$ for both residues and non-residues: With a quadratic residue g^{2e} we get $f(g^{2e}) = g^{2e(p-1)/2} = 1^e = 1$ where we used $g^{p-1} = 1$. With a non-residue $a = g^k$, k odd, we get

[1] The particular choice of the primitive root does not matter.

$f(a) = f(g^k) = g^{k(p-1)/2} = -1$ where we used $g^{(p-1)/2} = -1$ (the only square root of 1 apart from 1 is -1) and $-1^k = -1$ for k odd.

Apparently we just found a function that can tell residues from non-residues. In fact, we rediscovered the so-called *Legendre symbol* usually written as $\left(\frac{a}{p}\right)$. A surprising result (proved by Gauss) about the Legendre symbol is the *law of quadratic reciprocity*: Let p and q be distinct odd primes, then

$$\left(\frac{p}{q}\right) = (-1)^{\frac{p-1}{2}\,\frac{q-1}{2}} \left(\frac{q}{p}\right) \tag{21.35}$$

Moreover,

$$\left(\frac{-1}{p}\right) = (-1)^{\frac{p-1}{2}} = \begin{cases} +1 & \text{if} \quad p \equiv 1 \pmod 4 \\ -1 & \text{if} \quad p \equiv 3 \pmod 4 \end{cases} \tag{21.36}$$

and

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}} = \begin{cases} +1 & \text{if} \quad p \equiv \pm 1 \pmod 8 \\ -1 & \text{if} \quad p \equiv \pm 3 \pmod 8 \end{cases} \tag{21.37}$$

Further,

$$\left(\frac{3}{p}\right) = 1 \quad \Longleftrightarrow \quad p \equiv \pm 1 \bmod 12 \tag{21.38}$$

and

$$\left(\frac{-3}{p}\right) = 1 \quad \Longleftrightarrow \quad p = 2,\ p = 3 \text{ or } p \equiv 1 \bmod 3 \tag{21.39}$$

If a is a quadratic residue modulo p then the polynomial $x^2 - a$ factors as $(x - r_1)(x - r_2)$ where $r_1^2 \equiv a$ and $r_2^2 \equiv a$. Modulo $41 = 4 \cdot 10 + 1$ minus one is a quadratic residue and we have $x^2 + 1 = (x - 9)(x - 32)$. The polynomial $x^2 + 1$ with coefficients modulo $43 = 4 \cdot 10 + 3$ is irreducible.

The relation between the Legendre symbols of positive and negative arguments is

$$\left(\frac{-a}{p}\right) = (-1)^{\frac{p-1}{2}} \left(\frac{a}{p}\right) = \begin{cases} +(\frac{a}{p}) & \text{if} \quad p = 4k + 1 \\ -(\frac{a}{p}) & \text{if} \quad p = 4k + 3 \end{cases} \tag{21.40}$$

Modulo a prime $p = 4\,k + 3$, if a is a square exactly if $-a$ is not a square. The orders of any two elements a and $-a$ differ by a factor of two.

Modulo a prime $p = 4\,k + 1$, if $-a$ is a square exactly if $-a$ is a square. The order of two non-residues a and $-a$ are identical. The orders of two residues a and $-a$ can be identical or differ by a factor of two.

A special case are primes of the form $p = 2^x + 1$, the *Fermat primes*. Only five Fermat primes are known today: $2^1 + 1 = 3$, $2^2 + 1 = 5$, $2^4 + 1 = 17$, $2^8 + 1 = 257$ and $2^{16} + 1 = 65537$. To be prime the exponent x must actually be a power of two. The primitive roots are exactly the non-residues: The maximal order equals $R = \varphi(p) = 2^x$. There are $\varphi\left(\varphi(p)\right) = 2^{x-1}$ primitive roots. There are $(p - 1)/2 = 2^{x-1}$ squares which all have order at most $R/2$. Remain 2^{x-1} non-residues which must all be primitive roots.

We will not pursue the issue, but it should be noted that there are more efficient ways than powering to compute f and the concept of quadratic residues can be carried over to composite moduli: The so-called *Kronecker symbol* generalizes the Legendre symbol. An efficient implementation for its computation (following [Cohen]) is given in [FXT: `kronecker()` in mod/kronecker.cc]:

```
int
kronecker(umod_t a, umod_t b)
// Return Kronecker symbol (a/b).
// Equal to Legendre symbol (a/b) if b is an odd prime.
{
```

b\a	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36
0:	0	+	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1:	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
2:	0	+	0	−	0	−	0	+	0	−	0	−	0	+	0	+	0	−	0	−	0	+	0	+	0	−	0
3:	0	+	−	0	+	−	0	+	−	0	+	−	0	+	−	0	+	−	0	+	−	0	+	−	0	+	−	0
4:	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0
5:	0	+	−	−	+	0	+	−	−	+	0	+	−	−	+	0	+	−	−	+	0	+	−	−	+	0	+
6:	0	+	0	0	0	+	0	+	0	0	0	+	0	−	0	0	0	−	0	+	0	0	0	+	0	0	0	+	0
7:	0	+	+	−	+	−	−	0	+	+	−	+	−	−	0	+	+	−	+	−	−	0	+	+	−	+	−	−	0	+
8:	0	+	0	−	0	−	0	+	0	+	0	−	0	−	0	+	0	+	0	−	0	−	0	+	0	+	0	−	0
9:	0	+	+	0	+	+	0	+	+	0	+	+	0	+	+	0	+	+	0	+	+	0	+	+	0	+	+	0
10:	0	+	0	+	0	0	0	−	0	+	0	−	0	+	0	0	0	−	0	−	0	−	0	0	0	+	0	−	0	0	0
11:	0	+	−	+	+	+	−	−	−	+	−	0	+	−	+	+	+	−	−	−	+	−	0	+	−	+
12:	0	+	0	0	0	−	0	+	0	0	0	−	0	+	0	0	0	−	0	+	0	0	0	−	0	+	0	0	0	−	0
13:	0	+	−	+	+	−	−	−	−	+	+	−	+	0	+	−	+	+	−	−	−	−	+	+	−	+	+
14:	0	+	0	+	0	+	0	0	0	+	0	−	0	+	0	+	0	−	0	+	0	0	0	+	0	+	0	−	0	−	0	0	0
15:	0	+	+	0	+	0	0	−	+	0	0	−	0	−	−	0	+	+	0	+	0	0	−	0	+	0	−	−	0	+	+	0	+	0	0
16:	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0
17:	0	+	+	−	+	−	−	−	+	+	−	−	−	+	−	+	+	0	+	+	−	+	−	−	−	+	+	−	+	−	−	−	+	−	+	+	0	+	+
18:	0	+	0	0	0	−	0	+	0	0	0	−	0	−	0	0	0	+	0	−	0	0	0	+	0	+	0	0	0	−	0	+	0	0	0	−	0
19:	0	+	−	−	+	+	+	+	−	+	−	−	−	−	+	+	−	0	+	−	−	+	+	+	+	−	+	−	+	−	−	−	−	+	+
20:	0	+	0	−	0	0	0	−	0	+	0	+	0	−	0	0	0	−	0	+	0	+	0	−	0	0	0	−	0	+	0	+	0	−	0	0	0

Figure 21.7: Kronecker symbols $\left(\frac{a}{b}\right)$ for small positive a and b.

```
    static const int  tab2[] = {0, 1, 0, -1, 0, -1, 0, 1};
    // tab2[ a & 7 ] := (-1)^((a^2-1)/8)

    if ( 0==b )  return (1==a);
    if ( 0==((a|b)&1) )  return 0;  // a and b both even ?

    int v = 0;
    while ( 0==(b&1) )  { ++v;  b>>=1; }

    int k;
    if ( 0==(v&1) )  k = 1;
    else             k = tab2[ a & 7 ];

    while ( 1 )
    {
        if ( 0==a )  return ( b>1 ? 0 : k );

        v = 0;
        while ( 0==(a&1) )  { ++v;  a>>=1; }

        if ( 1==(v&1) )  k *= tab2[ b & 7 ];  // k *= (-1)**((b*b-1)/8)

        if ( a & b & 2 )  k = -k;  // k = k*(-1)**((a-1)*(b-1)/4)

        umod_t r = a; // signed:  r = abs(a)
        a = b % r;
        b = r;
    }
}
```

A table of Kronecker symbols $\left(\frac{a}{b}\right)$ for small a and b is shown in figure 21.7. It was created with the program [FXT: demo mod/kronecker-demo.cc].

A curious observation regarding quadratic residues is that exactly for the 29 moduli

```
2, 3, 4, 5, 8, 12, 15, 16, 24, 28, 40, 48, 56, 60, 72, 88, 112, 120,
168, 232, 240, 280, 312, 408, 520, 760, 840, 1320, 1848
```

all quadratic residues are non-prime. This is sequence A065428 of [OEIS]. It can be generated using the program [FXT: demo mod/mod-residues-demo.cc].

See any textbook on number theory for the details of the theory of quadratic residues, [NTC] and [Cohen] for the corresponding algorithms. The lecture notes [SteinNT] give a very readable introduction, too.

21.9 The Rabin-Miller test for compositeness

We describe a probabilistic method to prove compositeness of an integer.

21.9.1 Pseudoprimes and strong pseudoprimes

For a prime p the maximal order of an element equals $p-1$. That is, for all $a \neq 0$

$$a^{p-1} \equiv 1 \mod p \tag{21.41}$$

If for a given number n one succeeds to find an $a > 1$ so that $a^{n-1} \neq 1 \bmod p$ the compositeness of n has been proven. Composite numbers n for which $a^{n-1} = 1 \bmod n$ are called *pseudoprime to base a* (or a-pseudoprime). Example: For $n = 15$ one finds

a :	2	3	4	5	6	7	8	9	10	11	12	13	14
a^{14} :	4	9	1	10	6	4	4	6	10	1	9	4	1

We found that 15 is pseudoprime to the bases 4, 11 and 14 which we also could have read off the rightmost column of figure 21.6 on page 532.

The bad news is that some composite numbers are pseudoprime to very many bases. The smallest such example is number 561 which is pseudoprime to all bases a with $\gcd(a, n) = 1$. Numbers with that property are called *Carmichael numbers*, The first few are 561, 1105, 1729, 2465, 2821, 6601, 8911, ..., this is sequence A002997 of [OEIS]. Finding a base that proves a Carmichael number composite is as difficult as finding a factor.

A significantly better algorithm can be found by a rather simple variation. We use the numbers q and t so that $n - 1 =: q \cdot 2^t$ and examine the values $b := a^q$, b^2, b^4, ..., $b^{2^{t-1}} = a^{(n-1)/2}$. We say that n is a *strong pseudoprime to base a* if either $b \equiv 1$ or $b^{2^e} \equiv -1 \equiv n-1$ for some e where $0 \leq e < t$. We abbreviate strong pseudoprime as SPP. If neither of the conditions hold then n is proven composite. Then n is either not a pseudoprime to base a or we found a square root of 1 that is not equal to $n-1$.

With two different square roots s_1, s_2 modulo n of some number z (here $z = 1$) one has

$$s_1^2 - z \equiv 0 \mod n \tag{21.42a}$$
$$s_2^2 - z \equiv 0 \mod n \tag{21.42b}$$
$$s_1^2 - s_2^2 = (s_1 + s_2)\,(s_1 - s_2) \equiv 0 \mod n \tag{21.42c}$$

So both $s_1 + s_2$ and $s_1 - s_2$ are nontrivial factors of n if $s_1 \neq n - s_2$. Thereby a square root $s \neq -1$ of 1 proves compositeness because both $\gcd(s+1, n)$ and $\gcd(s-1, n)$ are nontrivial factors of n.

Let $B = \left[b, b^2, b^4, \ldots, b^{2^t}\right]$, then for n prime the sequence B must have one of the following forms: Either

$$B = [1, 1, 1, \ldots, 1] \quad \text{or} \tag{21.43a}$$
$$B = [*, \ldots, *, -1, 1, \ldots, 1] \tag{21.43b}$$

where an asterisk denotes any number not equal to $\pm 1 \bmod n$ (notation as in [NTC]). For n composite the sequence B can also be of the form

$$B = [*, \ldots, *] \quad (a^{n-1} \neq 1) \quad \text{or} \tag{21.44a}$$
$$B = [*, \ldots, *, 1, \ldots, 1] \quad \text{(found square root of 1 not equal to } -1) \tag{21.44b}$$

If one of the latter two forms is encountered then n must be composite.

With our example $n = 15$ we have $n - 1 = 7 \cdot 2^1$, thereby $q = 7$ and $t = 1$. We only have to examine the value of b. Values of a for which b is not equal to either $+1$ or -1 prove the compositeness of 15.

a :	2	3	4	5	6	7	8	9	10	11	12	13	14
b :	8	12	4	5	6	13	2	9	10	11	3	7	−1

In our example all bases $\neq 14$ prove 15 composite. As n is always a SPP to base $a = n - 1 \equiv -1$ we restrict our attention to values $2 \leq a \leq n - 2$.

A pari/gp implementation of the test whether n is a SPP to base a:

```
sppq(n,a)=
{ /* Return whether n is a strong pseudoprime to base a */
    local(q, t, b, e);
    q = n-1;
    t = 0;
    while ( 0==bitand(q,1), q/=2; t+=1 );
    /* here  n==2^t*q+1 */
    b = Mod(a, n)^q;
    if ( 1==b, return(1) );
    e = 1;
    while ( e<t,
        if( (b==1) || (b==n-1), break(); );
        b *= b;
        e++;
    );
    return( if ( b!=(n-1), 0, 1 ) );
}
```

The Carmichael number 561 ($561 - 1 = 35 \cdot 2^4$, so $q = 35$ and $t = 4$) is a SPP to 8 out of the 558 interesting bases:

```
a=50:      b=560
a=101:     b=560
a=103:     b=  1
a=256:     b=  1
a=305:     b=560
a=458:     b=560
a=460:     b=  1
a=511:     b=  1
```

The number 561 is not a SPP for any $2 \le a \le 20$:

```
a=2:       b=263  b^2=166  b^4= 67
a=3:       b= 78  b^2=474  b^4=276
a=4:       b=166  b^2= 67  b^4=  1
a=5:       b= 23  b^2=529  b^4=463
a=6:       b=318  b^2=144  b^4=540
a=7:       b=241  b^2=298  b^4=166
a=8:       b=461  b^2=463  b^4= 67
a=9:       b=474  b^2=276  b^4=441
a=10:      b=439  b^2=298  b^4=166
a=11:      b=209  b^2=484  b^4=319
a=12:      b= 45  b^2=342  b^4=276
a=13:      b=208  b^2= 67  b^4=  1
a=14:      b=551  b^2=100  b^4=463
a=15:      b=111  b^2=540  b^4=441
a=16:      b= 67  b^2=  1
a=17:      b=527  b^2= 34  b^4= 34
a=18:      b=120  b^2=375  b^4=375
a=19:      b= 76  b^2=166  b^4= 67
a=20:      b=452  b^2=100  b^4=463
```

Note that with $a = 4$ we found $s = 67$ where $s^2 \equiv 1 \bmod 561$ and thereby the factors $\gcd(67+1, 561) = 17$ and $\gcd(67 - 1, 561) = 33$ of 561.

21.9.2 The Rabin-Miller test

The *Rabin-Miller test* is an algorithm to prove compositeness of a number n by testing strong pseudoprimality with several bases:

```
rm(n, na=20)=
{ /* Rabin Miller test */
    local(a);
    for (a=2, na+2,
        if ( a>n-2,  break() );
        if ( 0==sppq(n, a), return(0) ); /* proven composite */
    );
    return(1); /* composite with probability less than 0.25^na */
}
```

It can be shown that for a composite number the probability to be a SPP to a 'random' base is at most 1/4. Thereby the compositeness of number can in practice quickly be proven. While the algorithm does *not* prove primality, it can be used to rule out compositeness with a very high probability.

A list (created with the program [FXT: demo mod/rabinmiller-demo.cc]) of composites $n \le 10,000$ that are SPP to more than one base $a \le 17$ is shown in figure 21.8 on the next page. The table indicates how effective the Rabin-Miller algorithm actually is: It does not contain a single number pseudoprime to both 2 and 3. The first few composite numbers that are SPP to both $a = 2$ and $a = 3$ are shown in

```
       Bases tested:    2   3  5  6  7 10 11 12 13 14 15 17
             91:  [3]                 10    12             17
            133:  [2]                    11 12
            145:  [2]                       12             17
            276:  [2]                    11    13
            286:  [2]       3                               17
            703:  [2]       3        7
            742:  [2]                                   15 17
            781:  [2]          5                            17
            946:  [2]                7                  15
           1111:  [2]             6                        17
           1729:  [2]                   10    12
           2047:  [2]    2                 11
           2806:  [2]          5                 13
           2821:  [2]                       12             17
           3277:  [3]    2                                14 15
           4033:  [2]    2                                   17
           4187:  [2]                   10                  17
           5662:  [2]          5                            17
           5713:  [2]             6                  14
           6533:  [2]             6    10
           6541:  [2]                                  14 15
           7171:  [2]                                  14    17
           8401:  [2]       3           10
           8911:  [3]       3                    12 13
           9073:  [2]                            12    14
```

Figure 21.8: All numbers $\leq 10,000$ theat are strong pseudoprimes to more than one base ≤ 17 (omitting bases that are perfect prime powers).

```
  1,373,653  ==  1 + 2^2 * 3^3 * 7 * 23 * 79
             ==  829 * 1657  == (1 + 2^2 *3^2 *23) * (1 + 2^3 *3^2 *23)

  1,530,787  ==  1 + 2 * 3 * 103 * 2477
             ==  619 * 2473  == (1 + 2 *3 *103)    * (1 + 2^3 *3 *103)

  1,987,021  ==  1 + 2^2 * 3^2 * 5 * 7 * 19 * 83
             ==  997 * 1993  == (1 + 2^2 *3 *83)   * (1 + 2^3 *3 *83)

  2,284,453  ==  1 + 2^2 * 3^2 * 23 * 31 * 89
             == 1069 * 2137  == (1 + 2^2 *3 *89)   * (1 + 2^3 *3 *89)

  3,116,107  ==  1 + 2 * 3^2 * 7^2 * 3533
             ==  883 * 3529  == (1 + 2 *3^2 *7^2)  * (1 + 2^3 *3^2 *7^2)

  5,173,601  ==  1 + 2^5 * 5^2 * 29 * 223
             ==  929 * 5569  == (1 + 2^5 *29)      * (1 + 2^6 *3 *29)

  6,787,327  ==  1 + 2 * 3 * 7 * 13 * 31 * 401
             == 1303 * 5209  == (1 + 2 *3 *7 *31)  * (1 + 2^3 *3 *7 *31)

 11,541,307  ==  1 + 2 * 3 * 7 * 283 * 971
             == 1699 * 6793  == (1 + 2 *3 *283)    * (1 + 2^3 *3 *283)

 13,694,761  ==  1 + 2^3 * 3^2 * 5 * 109 * 349
             == 2617 * 5233  == (1 + 2^3 *3 *109)  * (1 + 2^4 *3 *109)
```

Figure 21.9: The first composite numbers that are SPP to both bases 2 and 3.

```
   25,326,001  ==  1 + 2^4 * 3^3 * 5^3 * 7 * 67
               == 2251 * 11251  == (1 + 2 *3^2 *5^3)  * (1 + 2 *3^2 *5^4)

  161,304,001  ==  1 + 2^6 * 3 * 5^3 * 11 * 13 * 47
               == 7333 * 21997  == (1 + 2^2 *3 *13 *47)  * (1 + 2^2 *3^2 *13 *47)

  960,946,321  ==  1 + 2^4 * 3 * 5 * 29 * 101 * 1367
               == 11717 * 82013  == (1 + 2^2 *29 *101)  * (1 + 2^2 *7 *29 *101)

1,157,839,381  ==  1 + 2^2 * 3^3 * 5 * 401 * 5347
               == 24061 * 48121  == (1 + 2^2 *3 *5 *401)  * (1 + 2^3 *3 *5 *401)

3,215,031,751  ==  1 + 2 * 3^4 * 5^3 * 7 * 37 * 613
               == 151 * 751 * 28351
               == (1 + 2 *3 *5^2)  * (1 + 2 *3 *5^3)  * (1 + 2 *3^4 *5^2 *7)

3,697,278,427  ==  1 + 2 * 3^3 * 31 * 563 * 3923
               == 30403 * 121609  == (1 + 2 *3^3 *563)  * (1 + 2^3 *3^3 *563)
```

Figure 21.10: The first composite numbers that are SPP to all bases 2, 3 and 5.

figure 21.9 on the facing page. Composites that are SPP to the three bases 2, 3 and 5 are quite rare, figure 21.10 on the preceding page shows all 6 such composite numbers smaller than 2^{32} (values taken from [82] which lists all such numbers $< 25 \cdot 10^9$). Thereby one can speed up the Rabin-Miller test for small values of n (say, $n < 2^{32}$) by only testing the bases $a = 2, 3, 5$ and, if n is a SPP to these bases, look up the composites in the table.

Note that if the probability of a base not proving compositeness was exactly $1/4$ we would find *much* more entries in figure 21.10. Slightly overestimating the number of composites below N as N, there should be about $(1/4)^3\, N = N/64$ entries, that is $156,250$ for $N = 10^7$. Thereby the Rabin-Miller test is in practice much more efficient than one may initially assume.

```
       Bases tested:  2   3  5  6  7 10 11 12 13 14 15 17
              11476:      3  5                    15 17
              88831:               7    11 12       15 17
             188191:      3               7 10          17
             597871:                      10 11 12 13
             736291:                      7 10    12 13
             765703:                      10 11 12    14
            1024651:      3  5            7       12       15
            1056331:                      7 10    12 13
            1152271:      3               7    11 12 13
            1314631:         5            7    11 12 13
            1373653:   2  3     6                12          17
            1530787:   2  3     6                12
            1627921:      3  5                          14 15
            1857241:         5                 11 12    14
            1987021:   2  3     6                12          17
            2030341:                          11 12    14 15
            2284453:   2  3     6  7          11 12
            2741311:                          11 12    14 15
            3116107:   2  3     6                12
            4181921:   2     5        10             13
            4224533:               6           11          14 15 17  <--=
            5122133:               6  7 10 11
            5173601:   2  3     6                12
            5481451:      3  5                    12          15
            6594901:         5                    12 13 14
            6787327:   2  3     6                12 13
            8086231:         5                    12    14    17
            9504191:                          11 12 13    15
            9863461:   2     5  6     10                      17
```

Figure 21.11: Composites $\leq 10^7$ that are SPP to at least four bases ≤ 17 (omitting bases that are perfect prime powers).

The composites $\leq 10^7$ that are SPP to four or more bases $a < 17$ are shown in figure 21.11. We omit values of a that are perfect powers because if n is a base-a SPP then it is also a base-a^k SPP for all $k > 1$. The entry for $n = 4224533$ (marked with an arrow) shows that a number that is not a SPP to two bases a_1 and a_2 may still be a SPP to the base $a_1 \cdot a_2$ (here $a_1 = 2$, $a_2 = 3$). This indicates that one might want to restrict the tested bases to primes. All odd composite numbers $\leq 10^7$ that are SPP to four or more prime bases $a \leq 17$ are

```
    Bases tested:   2  3  5  7 11 13 17
   1152271:  [4]       3     7 11 13
   1314631:  [4]          5  7 11 13
   2284453:  [4]    2  3     7 11
```

Note that a number that is a SPP to bases a_1 and a_2 is *not* necessarily SPP to the base $a_1 \cdot a_2$. An example is $n = 9,006,401$ which is a SPP to bases 2 and 5 but not to base 10:

```
  9006401:   2     4  5        8                       16    18
```

All composite $n \leq 10^7$ that are SPP to bases 2 and 3 are also SPP to base 6, same for bases 2 and 5. Out of six composites $n \leq 10^7$ that are SPP to bases 2 and 7 three are not SPP to base 14:

```
   314821:   2     4     6  7  8  9                    16    18
  2269093:   2     4        7  8                 14    16
  2284453:   2  3  4     6  7  8  9    11 12          16    18
  3539101:   2     4        7  8              13       16
  5489641:   2     4        7  8                 14    16
  6386993:   2     4        7  8                 14    16
```

A C++ implementation of the Rabin-Miller test is [FXT: file mod/rabinmiller.cc]:

```
bool
rabin_miller(umod_t n, uint cm/*=0*/)
// Rabin-Miller compositeness test.
// Return true of none of the bases <=cm prove compositeness.
// If false is returned then n is proven composite (also for n=1 or n=0).
// If true is returned the probability
//   that n is composite is less than (1/4)^cm
```

```
{
    if ( n<=1 )  return false;
    if ( n < small_prime_limit )  return  is_small_prime( (ulong)n );

    umod_t q;
    int t;
    n2qt(n, q, t);

    if ( 0==cm )  cm = 20;  // default
    uint c = 0;
    while ( ++c<=cm )
    {
        umod_t a = c + 1;

        // if n is a c-SPP then it also is a c**k (k>1) SPP.
        // That is, powers of a non-witness are non-witnesses.
        // So we skip perfect powers:
        if ( is_small_perfpow(a) )  continue;

        if ( a >= n )  return  true;
        if ( !is_strong_pseudo_prime(n, a, q, t) )  return false;  // proven composite
    }

    return true;  // strong pseudoprime for all tested bases
}
```

The function `is_small_perfpow()` [FXT: file mod/perfpow.cc] returns `true` is its argument is a (small) perfect power. It uses a lookup in a precomputed bit-array.

21.10 Proving primality

We describe several methods to prove primality. Only the first, Pratt's certificate of primality, is applicable for numbers of arbitrary form but not practical in general because it relies on the factorization of $n-1$. The Pocklington-Lehmer test only needs a partial factorization of $n-1$. We give further tests applicable only for numbers of a special forms: Pepin's test and the Lucas-Lehmer test.

As already said, the Rabin-Miller test can only prove compositeness. Even if a candidate 'survives' many passes, we only know that it is prime with a high probability.

21.10.1 Pratt's certificate of primality

Only with a prime modulus p the maximal order equals $R = p-1$. In order to determine the order of an element modulo p one needs the factorization of $p-1$. Now if the factorization of $p-1$ is known and we can find a primitive root then we do know that p is prime. Thereby it is quite easy to prove primality for numbers of certain special forms. For example, let $p := 2 \cdot 3^{30} + 1 = 411,782,264,189,299$. One finds that 3 is a primitive root and so we know that p is prime.

In general, the factorization of $p-1$ can contain large factors whose primality needs to be proven. Recursion leads to a primality certificate in the form of a tree which is called *Pratt's certificate of primality*.

A certificate for the primality of $p = 314,159,311$ is shown in figure 21.12 on the next page. The first line says that 3 is a primitive root of $p = 314159311$ and $p-1$ has the prime factors $2, 3, 5, 199, 1949$ (actually, $p - 1 = 2 \cdot 3^4 \cdot 5 \cdot 199 \cdot 1949$ but we can ignore exponents). In the the second level (that is, indented by 4 characters) appear the prime factors just determined together with their primality certificates: The prime 2 is trivially accepted, all other primes are followed by their (further indented) certificates.

The certificate was produced with the following pari/gp code:

```
indprint(x, ind)=
{ /* print x, indented by ind characters */
    for (k=1, ind, print1(" ") );
    print(x);
}
```

```
          [314159311, [3], [2, 3, 5, 199, 1949]]
              [2, "--"]
              [3, [2], [2]]
                  [2, "--"]
              [5, [2], [2]]
                  [2, "--"]
              [199, [3], [2, 3, 11]]
                  [2, "--"]
                  [3, [2], [2]]
                      [2, "--"]
                  [11, [2], [2, 5]]
                      [2, "--"]
                      [5, [2], [2]]
                          [2, "--"]
              [1949, [2], [2, 487]]
                  [2, "--"]
                  [487, [3], [2, 3]]
                      [2, "--"]
                      [3, [2], [2]]
                          [2, "--"]
```

Figure 21.12: A certificate for the primality of $p = 314,159,311$.

```
pratt(p, ind=0)=
{
    local( a, p1, f, nf, t );
    if ( p<=2,   \\ 2 is trivially prime
        indprint([p, "--"], ind);
        return();
    );
    \\ p-1 is factored here:
    a = component(znprimroot(p),2);
    \\ but we cannot access the factorization,
    \\  so we do it "manually":
    p1 = p-1;
    f = factor(p1);
    nf = matsize(f)[1];
    t = vector(nf,j, f[j,1] );  f = t;  \\ prime factors only
    indprint([p, [a], t], ind);
    \\ recurse on prime factors of p-1:
    for (k=1, nf, pratt(f[k], ind+4));
    return();
}
p=nextprime(Pi*10^8)
pratt(p)
```

The routine has to be taken with a grain of salt as we rely on the fact that `znprimroot(p)` will fail for composite p:

```
? pratt(1000)
  ***   primitive root does not exist in gener
```

The routine has an additional parameter `ind` determining the indentation used with printing. This parameter is incremented with the recursion level, resulting in the tree-like structure of the output. This little trick is often useful with recursive procedures.

With a precomputed table of small primes (see section 21.3) the line

```
    if ( p<=2,   \\ 2 is trivially prime
```

can be changed to something like

```
    if ( (p<=ptable_max) && (ptable[p]==1),   \\ trivial to verify
```

which will shorten the certificate significantly.

A certificate for $p = 3,141,592,653,589,861$ and `ptable_max`=100 is shown in figure 21.13 on the following page. The certificate is 22 lines as opposed to 58 lines using the original routine where the certificates

```
[3141592653589861, [6], [2, 3, 5, 52359877559831]]
    [2, "--"]
    [3, "--"]
    [5, "--"]
    [52359877559831, [7], [2, 5, 53, 73, 2531, 534697]]
        [2, "--"]
        [5, "--"]
        [53, "--"]
        [73, "--"]
        [2531, [2], [2, 5, 11, 23]]
            [2, "--"]
            [5, "--"]
            [11, "--"]
            [23, "--"]
        [534697, [5], [2, 3, 22279]]
            [2, "--"]
            [3, "--"]
            [22279, [3], [2, 3, 47, 79]]
                [2, "--"]
                [3, "--"]
                [47, "--"]
                [79, "--"]
```

Figure 21.13: A shortened certificate for the primality of $p = 3,141,592,653,589,861$ where all primes smaller than 100 are considered trivially verifiable.

for tiny primes like $2, 3, 5$ are repeated quite often. We could of course keep track of the so far proven primes and just reference the first proof when a prime is encountered again later.

Once a certificate is computed it can be verified extremely quickly. Note that this type of primality certificate needs the factorization of $p-1$ and thus may not be feasible for large values of p.

21.10.2 Tests for $n = k\,2^t + 1$

Proth's theorem and Pepin's test

For numbers of the form $p = q \cdot 2^t + 1$ with q odd and $2^t > q$ primality can be proven as follows: If there is an integer a so that $a^{(p-1)/2} \equiv -1$ then p must be prime. This is *Proth's theorem*. The 'FFT-primes' are natural candidates for Proth's theorem. For example, with $p := 2^{57} \cdot 29 + 1 = 4,179,340,454,199,820,289$ one finds that $a^{(p-1)/2} \equiv -1$ for $a = 3$ so p must be prime.

Numbers of the form 2^t+1 are composite unless t is a power of two. The candidates are therefore restricted to the so-called *Fermat numbers* $F_n := 2^{2^n} + 1$. Here it suffices to test whether $3^x \equiv -1 \bmod F_n$ where $x = 2^{t-1}$:

```
pepin(tx)=
{
    local(t, F, x);
    t = 2^tx;
    F = 2^t+1;
    x = 2^(t-1);
    return(  (-1==Mod(3,F)^x) );
}
```

This test is known as *Pepin's test*. As shown in section 21.8 on page 536 all non-residues are primitive roots modulo prime F_n. Three is just the smallest non-residue.

```
for (tx=1,12, print(tx,"  ", pepin(tx)))
   1  1  \\ F_1 = 5
   2  1  \\ F_2 = 17
   3  1  \\ F_3 = 257
   4  1  \\ F_4 = 65537
   5  0
   6  0
  12  0
```

No Fermat prime greater than $F_4 = 65537$ is known today and all F_n, $5 \le n \le 32$ are known to be composite.

Note that F_{n+1} has (about) twice as many bits as F_n. Further the number of squarings involved in the test $(t-1)$ is (about) doubled. If we underestimate the cost of multiplying N-bit numbers with N operations we get a lower bound for the ratio of the costs of testing F_{n+1} and F_n of four. Assuming the computer power doubles every 18 month and Pepin's test of F_n is just feasible today we'd have to wait three years (36 month) before we can test F_{n+1}.

What to consider before doing the Pepin's test

As $2^t \equiv -1 \bmod F_n = 2^t + 1$ we see that the order of two equals $2\,t = 2^{n+1}$. The same is true for factors of composite Fermat numbers. When searching factors of F_n one only needs to consider candidates of the form $1 + k\,2^{n+2}$. A routine that searches for small factors of f_n can be implemented as:

```
ord2pow2(p)=
\\ Return the base-2 logarithm of the order of two modulo p
\\ Must have: ord(2)==2^k for some k
{
    local(m, rx);
    rx = 0;
    m = Mod(2,p);
    while ( m!=1, m*=m; rx++; );
    return( rx );
}

ftrialx(n, mm=10^5, brn=0)=
\\ Try to find small factors of the Fermat number F_n=2^(2^n)+1
\\ Try factors 1+ps, 1+2*ps, ... , 1+mm*ps  where ps=2^(n+2)
\\ Stop if brn factors were found (zero: do not stop)
{
    local(p,ps,ttx,fct);
    ps = 2^(n+2);  \\ factors are of the form 1+k*ps
    p = ps+1;      \\ trial factor
    ttx = 2^(n+1); \\ will test whether Mod(2,p)^ttx==1
    fct = 0;  \\ how many factors were found so far
    for (ct=1, mm,
        if ( (Mod(2,p)^(ttx)==1)  \\ order condition
            && ( (rx=ord2pow2(p)) == n+1 ) \\ avoid factors of smaller Fermat numbers
        , /* then */
            print1(n, ":  ");
            print1(p);
            print1("    p-1=",factor(p-1));
            print();
            fct++;
            if ( fct==brn,  break() );
        );
        p += ps;
    );
    return(fct);
}
```

We create a list of small prime factors of F_n for $5 \le n \le 32$ where the search is restricted to factors $f \le 1 + 10^5\,2^{n+2}$ and stopped when a factor was found:

```
for(n=5,32, ftrialx(n, 10^5, 1); );
   5:  641    p-1=[2, 7; 5, 1]
   6:  274177    p-1=[2, 8; 3, 2; 7, 1; 17, 1]
   9:  2424833    p-1=[2, 16; 37, 1]
  10:  45592577    p-1=[2, 12; 11131, 1]
  11:  319489    p-1=[2, 13; 3, 1; 13, 1]
  12:  114689    p-1=[2, 14; 7, 1]
  15:  1214251009    p-1=[2, 21; 3, 1; 193, 1]
  16:  825753601    p-1=[2, 19; 3, 2; 5, 2; 7, 1]
  18:  13631489    p-1=[2, 20; 13, 1]
  19:  70525124609    p-1=[2, 21; 33629, 1]
  23:  167772161    p-1=[2, 25; 5, 1]
  32:  25409026523137    p-1=[2, 34; 3, 1; 17, 1; 29, 1]
```

A list for $5 \le n \le 300$ is given in [FXT: file data/small-fermat-factors.txt]. Note that an entry

```
 201:  1245698371909569261600129013982869249475211760780421005925626675 21 \
    p-1=[2, 204; 3, 1; 5, 1; 17, 1; 19, 1]
```

asserts the compositeness of the number F_{201} where Pepin's test is out of reach by far. Indeed, its binary representation could not be stored in all existing computer memory combined: F_{201} is a $\log_2(F_{201}) \approx 2^{201} = 3.2138 \cdot 10^{60}$ -bit number.

The status of the (partial) factorizations of Fermat numbers is given in [Proth].

The Pocklington-Lehmer test

Let $p - 1 = F \cdot U$ where $F > U$ and all prime factors of F are known. If, for each prime factor q of F, we can find a_q so that $a_q^{p-1} \equiv 1 \bmod p$ and $\gcd\left(a_q^{(p-1)/q} - 1, p\right) = 1$, then p is prime.

Note that Proth's theorem is the special case $F = 2^t > k = U$.

The following implementation removes entries from the list of prime factors q of F until the list is empty:

```
pocklington_lehmer(F, u, c=10000)=
{ /* Pocklington-Lehmer test for the primality of p=f*u+1.
   * Return last successful base, else zero.
   * F must be the factorization of f.
   * Test bases a=2...c
   * Must have u<f.
   */
    local(n, f, C, p, t, ct);
    n = matsize(F)[1];
    f = prod(j=1, n, F[j,1]^F[j,2]);
    if ( f<=u, return(0) );
    p = f*u + 1;
    C = vector(n, j, (p-1)/F[j,1]);
    ct = n;  \\ number remaining prime divisors of f
    for (a=2, c,
        if ( 1==Mod(a,p)^(p-1),
            for (j=1, n,
                if ( C[j]!=0,  \\ skip entries already removed
                    t = component( Mod(a,p)^C[j], 2);
                    if( 1==gcd(t-1, p),
                        C[j] = 0;  \\ remove entry
                        ct -= 1;   \\ number of remaining entries
                    );
                );
            );
            if ( ct==0, return(a) );
        );
    );
    return( 0 );
}
```

We search all primes of the form $p = F \cdot U + 1$ where $F = 100!$, $U = F - d$, and d lies in the range $1, \ldots, 1000$. Only candidates that are strong pseudoprimes to both bases 2 and 3 are tested:

```
f=100!;
F=factor(f);
{ for (d=1, 1000,
    u = f - d;
    p = f*u+1;
    if ( sppq(p, 2) && sppq(p,3),
        q2 = pocklington_lehmer(F, u);
        print1(d, ":  ");
        print1("  ", q2);
        print();
    );
 ) }
```

We find five such primes $\approx 8.70978248908948 \cdot 10^{315}$ (in about 18 seconds):

```
      d:   last a
     45:    103
    778:    101
    818:    101
    880:    101
    884:    103
```

The returned value a_q is the one that did lead to the removal of the last entry in `C[]`. The value is smaller with less prime factors of F. Using $F = 2^{500}$ we find primes ($\approx 1.07150860718626 \cdot 10^{301}$) of the form $p = F \cdot U + 1$ where $U = F - d$ and $1 \le d \le 3000$ for the following d and maximal a_q:

```
   d:   last a        d:   last a
 214:    5         1383:    3
 294:    3         1801:   13
1023:    3         2041:   11
1114:    5         2481:    3
1321:   17
```

A further refinement of the test is given in [Cohen].

21.10.3 Tests for $n = k\,2^t - 1$

The Lucas-Lehmer test for Mersenne numbers

Define the sequence H by $H_0 = 1$, $H_1 := 2$ and $H_i = 4\,H_{i-1} - H_{i-2}$. The Mersenne number $n = 2^e - 1$ is prime exactly if $H_{2^{e-2}} \equiv 0 \bmod n$. The first few terms of the sequence H are

```
  k: 0 1 2 3  4   5    6     7     8      9      10     11      12
H_k: 1 2 7 26 97 362 1351 5042 18817 70226 262087 978122 3650401 ...
```

An efficient algorithm is obtained via the index doubling formula $H_{2k} = 2\,H_k^2 - 1$. Starting with the value $H_1 = 2$ and computing modulo n the implementation is a s simple as

```
LL(e)=
{
    local(n, h);
    n = 2^e-1;
    h = Mod(2,n);
    for (k=1, e-2, h=2*h*h-1);
    return( 0==h );
}
```

```
? LL(521)
1  \\  2^521-1 is prime
? LL(239)
0  \\  2^239-1 is composite
? LL(9941)
1 \\ 2^9941-1 is prime
? ##
  ***   last result computed in 4,296 ms.
```

The algorithm is called the *Lucas-Lehmer test*.

Note that most sources use the sequence $V = 2, 4, 14, 52, 194, 724, 2702, \ldots$ that satisfies the same recurrence relation. We have $H_k = \frac{1}{2}\,V_k$ (H is *half* of V). The index doubling relation becomes $V_{2k} = V_k^2 - 2$.

An interesting observation about Mersenne numbers is that the following *seems* to be true:

$$n = 2^e - 1 \quad \text{prime} \quad \Longleftrightarrow \quad 3^{2^{e-1}} \equiv -3 \bmod n \tag{21.45}$$

Note that for odd e the condition is equivalent to $3^{(p-1)/2} \equiv -1 \bmod n$ and 3 is a non-residue. Alas, I haven't been able to prove this until now.

What to consider before doing the Lucas-Lehmer test

The exponent e of a Mersenne prime must be prime, else n factors algebraically as

$$2^e - 1 \quad = \quad \prod_{d \backslash e} Y_d(2) \tag{21.46}$$

where $Y_k(x)$ is the k-th cyclotomic polynomial. For example, for $2^{21} - 1 = 2097151$:

```
? m=1; fordiv(21,d,y2=subst(polcyclo(d,x),x,2);m*=y2;print(d,": ",y2)); m
  1: 1
  3: 7
  7: 127
  21: 2359
  2097151
```

The factors obtained are not necessarily prime: here 2359 factors further into $7 \cdot 337$. More information on the multiplicative structure of $b^e \pm 1$ can be found in [Fact].

Before doing the Lucas-Lehmer test one should do a special version of trial division based on the following observation: Any factor f of $m = 2^e - 1$ has the property that $2^e \equiv 1 \bmod f$. That is $2^e - 1 \equiv 0 \bmod f$, so $m \equiv 0 \bmod f$ and f divides m. We further use the fact that factors are of the form $2\,e\,k + 1$ and $f \equiv \pm 1 \bmod 8$. The following routine does not try assert the primality of a candidate factor f as this would render the computation considerably slower.

```
mers_trial(e, mct=10^7, bnf=0)=
\\  try to discover small factors of the Mersenne number 2^e-1
\\  e : exponent of the Mersenne number
\\  mct : how many factors are tried
\\  pfq : stop with the factor found (zero: do not stop)
{
    local(f, fi, ct, fct);
    print("exponent e=",e);
    print("trying up to ", mct, " factors");
    fi=2*e; \\ factors are of the form 2*e*k+1
    f=1;
    ct=0;
    fct=0;  \\ how many factors where found so far
    while (ct < mct,
        f += fi;
        m8 = bitand(f, 7);  \\ factor modulo 8
        if ( (1!=m8) && (7!=m8), next(); );  \\ must equal +1 or -1
        if ( Mod(2, f)^e == Mod(1, f),
            print(f, "  ", isprime(f));  \\ give factor and tell whether it is prime
            fct++;
            if ( fct==bnf , break(); );
       );
       ct++;
    );
}
```

For $m = 2^{10007} - 1$ (3013 decimal digits) we obtain three factors of which all are prime:

```
? e=10007; mers_trial(e,,3);
exponent e=10007
trying up to 10000000 factors
  240169  1
  60282169  1
  136255313  1
? ##
  ***   last result computed in 44 ms.
? ceil((e*log(2.0)/log(10.0)))
  3013 \\ m=2^e-1 has 3,013 decimal digits
```

Sometimes one is lucky with truly huge numbers:

```
? e=2^31-1; mers_trial(e,,1);
exponent e=2147483647
trying up to 10000000 factors
  295257526626031  1
? ##
  ***   last result computed in 583 ms.
? ceil((e*log(2.0)/log(10.0)))
  646456993  \\  m=2^e-1 has 646,456,993 decimal digits
```

Note that we found that $m = 2^e - 1$ is prime if and only if there is no prime $f < m$ where the order of two equals e. A special case is sometimes given as follows: If both $p = 4\,k + 3$ and $q = 2\,p + 1$ are prime then q divides $2^p - 1$ (because the order of 2 modulo q equals p).

By the way, if both $p = 4\,k + 1$ and $q = 2\,p + 1$ are prime then q divides $2^p + 1$ (because the order of 2 modulo q equals $2\,p$ and $2^{2\,p} - 1 = (2^p + 1)\,(2^p - 1)$).

The Lucas test

The Lucas-Lehmer test can be generalized for a less restricted set of candidates. The *Lucas test* can be stated as follows (taken from [HR]):

Let $n = k\,2^t - 1$ were k is odd, $2^t > k$, $n \neq 0 \bmod 3$ and $n + 1 \neq 0 \bmod 3$ (that is, $\gcd(k, 3) = 1$). Then n is prime if and only if $H_{(n+1)/4} \equiv 0 \bmod n$ where H is as given above.

To turn this into an efficient algorithm use the relation $(n + 1)/4 = k\,2^{t-2}$. Firstly, compute H_k as described in section 19.1.1 on page 462:

$$[H_k,\ H_{k+1}] \quad = \quad [H_0,\ H_1] \begin{bmatrix} 0 & -1 \\ 1 & 4 \end{bmatrix}^k \tag{21.47}$$

This is a one-liner in pari/gp:

```
? H(k)= return( ([1,2] * [0, -1; 1, 4]^k)[1] );
? for(k=0,10,print(k,": ",H(k),"  = 1/2 * ",2*H(k)))
  0: 1  = 1/2 * 2
  1: 2  = 1/2 * 4
  2: 7  = 1/2 * 14
  3: 26  = 1/2 * 52
  4: 97  = 1/2 * 194      /*  = 2*7^2-1 = (14^2-2)/2 */
  5: 362  = 1/2 * 724
  6: 1351  = 1/2 * 2702   /*  = 2*26^2-1 = (52^2-2)/2 */
  7: 5042  = 1/2 * 10084
  8: 18817  = 1/2 * 37634
  9: 70226  = 1/2 * 140452
  10: 262087  = 1/2 * 524174
```

To compute $H_{k\,2^{t-2}}$ from H_k use the relation index doubling relation ($H_{2k} = 2\,H_k^2 - 1$) $t - 2$ times. The test can be implemented as

```
H(k, n)= return( (Mod([1,2],n) * Mod([0,-1; 1,4], n)^k)[1] );

ll3(k,t)= /* Lucas-Lehmer test */
{
    local(n, h);
    /* check preconditions: */
    if ( 0==bitand(k,1), return(0) );  \\ k must be odd
    if ( k>=2^t, return(0) );
    n = k*2^t-1;
    if ( n%3!=2, return(0) );  \\ gcd(3,k)!=1 && gcd(3,n)!=1
    /* main loop: */
    h = H(k, n);
    for (j=1,t-2, h*=h; h+=h; h-=1; );  \\ index doubling
    return ( 0==h );
}
```

The divisibility conditions were summed as $n \equiv 2 \bmod 3$. Note that the routine returns 'false' even for primes if the preconditions are not met. With $n = 5 \cdot 2^{12} - 1 = 20479$ we get

```
    n=20479  k=5  t=12
     j  H_{k*2^j} modulo n
     0     362
     1   16339
     2   17832
     3    5581
     4   18482
     5    9686
     6    8593
     7    5228
     8    5516
     9    9402
    10       0
```

which shows that 20479 is prime. Proving $n = 5 \cdot 2^{1340} - 1$ prime takes about 20 milliseconds. The following code finds the first value $t \geq 2500$ so that $n = 5 \cdot 2^t - 1$ is prime:

```
k=5; t=2500;
while ( 0==ll3(k,t), t+=1; )
```

Within 3 seconds we obtain the result $t = 2548$.

21.10.4 Lucas-Lehmer test with floats *

Using the formula $\cosh(2\,x) = 2\,\cosh(x)^2 - 1$ we can give a criterion equivalent to the Lucas-Lehmer condition as follows:

$$\cosh\left(2^{m-2}\,\log\left(2+\sqrt{3}\right)\right) \quad\equiv\quad 0 \mod M_m \qquad\Longrightarrow\qquad M_m \text{ is prime} \tag{21.48}$$

The relation is computationally useless because the quantity to be computed grows doubly-exponentially with m: the number of digits grows exponentially with m. Already for $m = 17$ the calculation has to be carried out with more than $18,741$ decimal digits:

```
? cosh(2^(17-2)*log(2+sqrt(3)))
 1.88889395811398377260975384780566028594658443155 51 E18741
```

The program [hfloat: examples/ex8.cc] does the computations in the obvious (insane) way. Using a precision of 32,768 decimal digits we obtain:

```
cosh(...)=
 +.18888939581139837726097538478056602859465844315551 \
   [... about 18,000 digits ...]
    ... 55797500398006802841700000000000000 ...
                              ^[decimal point after 7]

   00000000000000000000000000000000000000000000000000 \
   [...]
   00000000000000001549695720446140150427588985400185472*10^18742
                  [nonzero due to numerical imprecision]
```

After rounding and computing the modulus the program declares $M_{17} = 2^{17} - 1$ prime. All this using just 4 MB of memory and computations equivalent to about 35 FFTs of length 1 million, taking about 4 seconds. This is many many million times the work needed by the original (sane) version of the test. Even trial division would have been significantly faster.

The number M_{31} would need a bigger machine as the computations would need more than 300 million digits:

```
? (2^(31-2)*log(2+sqrt(3)))/log(10)  /* approx decimal digits */
 307062001.46039800926268312190009204
```

Apart from being insane the computation can be used to test high precision floating point libraries.

21.10.5 Combined primality tests *

The observation given as relation 21.45 on page 547 and the fact that the candidates for the Lucas test are of the form $n = 24\,j + 7$ suggests that a combination of a Lucas type test and the test for strong pseudoprimality might give a test that is more effective.

Numbers of the form $n = 24\,j + 7$

First we list all composite numbers $n \equiv 7 \mod 24$ up to $2^{32} \approx 4 \cdot 10^9$ where $H_{(n+1)/4} \equiv 0 \mod n$ (that is, the Lucas test without the condition $2^t > k$ gives a false positive).

Together with each n up to five numbers a ($a < 100,000$) so that n is a strong pseudoprime to base-a (a-SPP). If no a is listed then n is not an a-SPP to any base $a < 100,000$. The data is given in [FXT: file data/pseudo-7mod24.txt], an extract is shown in figure 21.14 on the next page. Firstly, the extract given contains all (five) n that are SPP to any base $a < 100,000$. Secondly, the *Lucas pseudoprimes* (LPP) tend to get more sparse for greater values of n: there are just five values $3 \cdot 10^9 < n < 4 \cdot 10^9$.

The list suggests that composites of the form $n = 24\,j + 7$ that are both LPP and are SPP to a small base (say, $a = 3$) may be extremely rare. Can you find such a composite?

The implied test would cover 1/8 of all candidates (that are not divisible by 2 or 3), as eight numbers (1, 5, 7, 11, 13, 17, 19 and 23) are prime to 24.

```
1037623:  67191 67192                                    [--snip--]
2211631:  6333 7260 8160 16793 21219 21580          2946282799:
4196191:  9104 26498 93477                          3075304399:
7076623:                                            3145717759:
9100783:                                            3299597407:
11418991:  44936                                    3554502799:
15219559:                                           3554889199:
21148399:                                           4091977039:
 [--snip--]                                         4207009999:
829577839:
887557999:  4899 33982 46674 62180
961315183:
1192222639:
 [--snip--]
```

Figure 21.14: Composite numbers $n < 2^{32}$ of the form $n = 24\,k + 7$ that pass the Lucas type test. Five of them are strong pseudoprime to some base smaller than $100,000$.

Numbers of the form $n = 24\,j + 19$

For numbers of the form $n = 24\,j + 19$ we use a different test: Here we consider as Lucas pseudoprimes numbers n such that $U_{(n+1)/4} \equiv 0 \bmod n$ where $U_0 = 0$, $U_1 = 1$ and $U_k = 4\,U_{k-1} - U_{k-2}$.

```
30739:                                                   [--snip--]
153931:                                                2946282799:
249331:                                                3075304399:
1575859:                                               3145717759:
1960243:                                               3299597407:
2557627:  36814 49266 49267 86080                      3554502799:
3444403:                                               3554889199:
3767347:  26452 79860 94736                            4091977039:
3881179:  47489 67676 72825 73841 84995 87856          4207009999:
3882283:
14324491:
 [--snip--]
```

Figure 21.15: Composite numbers $n < 2^{32}$ of the form $n = 24\,k + 19$ that pass the Lucas type test. Four of them are strong pseudoprime to some base smaller than $100,000$.

The list [FXT: file data/pseudo-19mod24.txt] contains all composites $n < 2^{32}$ that pass the test together with all (but maximal 5) bases $a < 100,0000$ so that n is an a-SPP. An extract is shown in figure 21.15. Just four $n < 2^{32}$ are SPP to any base $a < 100,000$. Can you find a composite $n = 24\,j + 19$ that passes the test and is a SPP to any base $a < 100$?

Numbers of the form $n = 6\,j + 5$

For numbers of the form $n = 6\,j + 5$ (which include $n = 24\,j + r$ where $r \in \{5,\ 11,\ 17,\ 23\}$) one can use a different test: Here we consider as pseudoprimes numbers n such that $U_{(n+s)/2} \equiv 0 \bmod n$ where U as before and $s = (n \bmod 4) - 2$.

The list [FXT: file data/pseudo-5mod6.txt] gives those $n < 2^{32}$ that pass the test together with maximal five bases $a < 1,000$ so that n is a a-SPP. Figure 21.16 on the next page shows an extract. Exactly 62 numbers $n < 2^{32}$ are SPP to any base $a < 1,000$, the following are all n that are SPP to any prime base $a < 20$ ($a \in \{2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19\}$):

```
79786523:     %24=11  2 4 8 16 32 57
1094554151:   %24=23  7 16 49 51 54 60
1608751523:   %24=11  3 9 16 27 48 81
2437837967:   %24=23  7 49 74 264 275 297
2948525171:   %24=11  7 25 49 175 343 625
3120784151:   %24=23  11 12 27 64 114 121
3208902491:   %24=11  2 4 8 16 32 64
3283643351:   %24=23  13 135 169 211 238 333
```

None of the listed numbers is SPP to two prime bases less than 20. All numbers that are SPP to any base $a < 1,000$ are of the form $n = 24\,j + 11$ or $n = 24\,j + 23$ except for two 'small' entries:

```
209:   %24=17                          [--snip--]
989:   %24=5                          49327277:   %24=5
6767:   %24=23                        50156087:   %24=23   656 854 966
6989:   %24=5   418                   51010049:   %24=17
9869:   %24=5                          [--snip--]
13133:   %24=5                        4268561597:   %24=5
13529:   %24=17                       4279710665:   %24=17
20705:   %24=17                       4285858523:   %24=11   16 22 27 67 106 213
22577:   %24=17                       4289479349:   %24=5
 [--snip--]
361391:   %24=23
361397:   %24=5
370229:   %24=5
382523:   %24=11   12 22 37 114 126 144
391169:   %24=17
 [--snip--]
```

Figure 21.16: Composite numbers $n < 2^{32}$ of the form $n = 6\,k + 5$ that pass the Lucas type test. Four of the shown numbers are strong pseudoprime to some base smaller than $1,000$.

```
6989:   %24=5   418
137801: %24=17  30 112 135 332 504 900
```

This suggests that one should use another Lucas-type test for numbers of the form $n = 12\,j + 11$.

Can you find a composite of the form $n = 24\,j + 5$ or $24\,j + 17$ (that is, $n = 12\,j + 5$) that is SPP to a small base and for which $U_{(n+1)/2} \equiv 0 \bmod n$?

The remaining case: $n = 12\,j + 1$

With $n = 24\,j + 1$ we test whether $U_{(n-1)/4} \equiv 0 \bmod n$. The list of pseudoprimes $n < 2^{32}$ is given in [FXT: file data/pseudo-1mod24.txt]. For $n = 24\,j + 13$ we test whether $H_{(n-1)/4} \equiv 0 \bmod$. The list of pseudoprimes $n < 2^{32}$ is given in [FXT: file data/pseudo-13mod24.txt].

The many entries in both lists, together with the fact that the entries are often SPP to several small bases might seem discouraging. The reason is twofold: Firstly, strong pseudoprimes are unevenly distributed among the residue classes modulo 12, see [82]. Secondly, we have been sloppy with the Lucas type tests. One actually should use different recurrences dependent on what are quadratic residues modulo n. This is discussed in [15] where a combined test is suggested for which until today no composite passing the test has been found.

The various concepts of pseudoprimality are unified under the name *Frobenius pseudoprimes* in [53] which is highly recommended for further studies.

Further reading

A excellent introduction into topics related to prime numbers and methods of factorization is [HR]. Some of the newer factorization algorithms can be found in [Cohen], readable surveys are [75] and [81]. Tables of factorizations of numbers of the form $b^e \pm 1$ are given in [Fact] which also contains much historical information.

Remains to say that a deterministic polynomial-time algorithm for proving primality was published by Agrawal, Kayal and Saxena in August 2002 [1]. While this a major breakthrough in mathematics it does not render the Rabin-Miller test worthless. Practically, 'industrial grade' primes are still produced with it. An excellent introduction into the ideas behind the so-called *AKS algorithm* and its improvements is [54].

21.11 Complex moduli

Recall that with real numbers the equation $x^2 = -1$ has no solution, there is no real square root of minus one. The construction of complex numbers proceeds by taking pairs of real numbers $(a, b) = a + i\,b$ together with (component wise) addition $(a, b) + (c, d) = (a + c, b + d)$ and multiplication defined by $(a, b)\,(c, d) = (ac - bd, ad + bc)$. Indeed the pairs of real numbers together with addition and multiplication as given constitute a field.

We will now rephrase the construction in a way that that shows how to construct an *extension field* (the complex numbers in the example above) from a given *base field* (the real numbers in the example above) in a more general way.

21.11.1 The construction of complex numbers

Equivalently to saying that there is no real square root of minus one we could have said that the polynomial $x^2 + 1$ has no linear factor. The construction of the complex numbers proceeds by taking numbers of the form $a + b\,i$ where i was boldly defined to be a root of the polynomial $x^2 + 1$. Now observe that if we identify $a + b\,i = b\,i + a$ with the polynomial $b\,x + a$ and use polynomial addition and multiplication modulo the polynomial $x^2 + 1$ we get the arithmetic of complex numbers. Addition is component wise, no modular reduction occurs. Multiplication:

$$\begin{aligned}(b\,x + a)\;(d\,x + c) &= (bd)\,x^2 + (ad + bc)\,x + (ac) && (21.49a)\\ &\equiv (ad + bc)\,x + (ac - bd) \qquad (\mathrm{mod}\ x^2 + 1) && (21.49b)\end{aligned}$$

We used the relation $u\,x^2 \equiv -u \pmod{x^2 + 1}$. Identify x with i in the relations and observe that computations with a root (that is not in the base field) of a polynomial is equivalent to computations with polynomials (whose coefficients are in the base field) modulo the polynomial $x^2 + 1$.

When the base field is the real numbers the story comes to an end: Every polynomial of arbitrary degree n with complex coefficients has exactly n complex roots. That is, we cannot use the given construction to extend the field $\mathbb{C}$ of complex numbers as every polynomial $p(x)$ with coefficients in $\mathbb{C}$ is a product of linear factors: $p(x) = (x - r_1)\,(x - r_2)\ldots,(x - r_n)$ with $r_k \in \mathbb{C}$. The field $\mathbb{C}$ is algebraically closed.

If we choose the base field to be $\mathbb{F}_p = GF(p)$, the integers modulo a prime p, and an irreducible polynomial $c(x)$ of degree n whose coefficients are in $\mathbb{F}_p$ we get an extension field $\mathbb{F}_{p^n} = GF(p^n)$, a *finite field* with p^n elements. The special case of the *binary finite fields* $GF(2^n)$ is treated in chapter 23.

21.11.2 Complex finite fields

With primes of the form $p = 4k + 3$ it is possible to construct a field of complex numbers as minus one is a quadratic non-residue and so the polynomial $x^2 + 1$ is irreducible. The field is denoted by $GF(p^2)$.

The rules for complex addition, subtraction and multiplication are the 'usual' ones. The field has p^2 elements of which $R = p^2 - 1$ are invertible. The maximal order equals R, so the inverse of and element a can be computed as $a^{-1} = a^{R-1} = a^{p^2-2}$.

For example, the powers of $a = 1 + x = 1 + i$ modulo $c = x^2 + 1 = 0 = 3 + 3i$ are

```
a= 1+1*i

a^1 =  1+1*i
a^2 =  0+2*i  // (1+x)*(1+x)   = x^2+2*x+1   == 2*x+1 - 1 = 2*x+0 == 0+2*x
a^3 =  1+2*i  // (1+x)*(0+2*x) = 2*x^2+2*x   == 2*x   - 2 = 2*x-2 == 1+2*x
a^4 =  2+0*i  // (1+x)*(1+2*x) = 2*x^2+3*x+1 == 3*x+1 - 2 = 3*x-1 == 2+0*x
a^5 =  2+2*i  //                              ^^                    ^^
a^6 =  0+1*i  //                          mod(x^2+1)            mod(3)
a^7 =  2+1*i == a^(-1)= 2+1*i
a^8 =  1+0*i == one
a^9 =  1+1*i
```

```
R=maxord==8 == Mat([2, 3])
r=ord(a)==8
R/r=1
```

Note that the modular reduction happens with both the polynomial x^2+1 and $p=3$. The polynomial reduction uses $x^2=-1$.

With primes of the form $p=4k+1$ it also possible to construct a field $GF(p^2)$. But we have to use a different polynomial as x^2+1 is reducible modulo p and thereby the multiplication rules is different. For example, with $p=5$ we find that x^2+x+1 is irreducible:

```
? p=5;  m=Mod(1,p)*(1+x+x^2);  polisirreducible(m)
  1
a=Mod(1,p)*(1+3*x)
for(k=1,p^2-1,print("a^",k," = ",component(Mod(a,m)^k,2)))
  a^1 = Mod(3, 5)*x + Mod(1, 5)
  a^2 = Mod(2, 5)*x + Mod(2, 5)
  a^3 = Mod(2, 5)*x + Mod(1, 5)
  [...]
```

The output, slightly beautified:

```
a^1  = 1 + 3*x
a^2  = 2 + 2*x  // (1+3*x)*(1+3*x) = 9*x^2+6*x+1 == 6*x+1 - (9*x+9) =-3*x-8 == 2+2*x
a^3  = 1 + 2*x  // (1+3*x)*(2+2*x) = 6*x^2+8*x+2 == 8*x+2 - (6*x+6) = 2*x-4 == 1+2*x
a^4  = 0 + 4*x  // (1+3*x)*(1+2*x) = 6*x^2+5*x+1 == 5*x+1 - (6*x+6) =  -x-5 == 0+4*x
a^5  = 3 + 2*x  //                                  ^^                    ^^
a^6  = 2 + 0*x  //                          mod(x^2+x+1)                mod(5)
a^7  = 2 + 1*x
a^8  = 4 + 4*x
a^9  = 2 + 4*x
a^10 = 0 + 3*x
a^11 = 1 + 4*x
a^12 = 4 + 0*x
a^13 = 4 + 2*x
a^14 = 3 + 3*x
a^15 = 4 + 3*x
a^16 = 0 + 1*x
a^17 = 2 + 3*x
a^18 = 3 + 0*x
a^19 = 3 + 4*x
a^20 = 1 + 1*x
a^21 = 3 + 1*x
a^22 = 0 + 2*x
a^23 = 4 + 1*x
a^24 = 1 + 0*x
```

We see that $a=1+3\,x$ has the maximal order (24), it is a primitive root. The polynomial reduction uses $x^2=-(x+1)$.

The values of the powers of the primitive root can be used to 'randomly' fill a $p\times p$ array. With $a^k=u+x\,v$ we mark the entry at row v, column u with k:

```
[ 4 11  9 19  8]
[10  1 17 14 15]
[22  3  2  5 13]
[16 20  7 21 23]
[-- 24  6 18 12]
```

The position $0,0$ (lower left) is not visited. Note row zero is the lowest row.

As described, the procedure fills an $p\times p$ array where p is a prime. Working with an irreducible polynomial of degree n we can fill a $p^e\times p^f\times p^g\times\ldots\times p^k$ array if $e+f+g+\ldots+k=n$: For the exponents that are one just choose one polynomial coefficient. For the exponents greater one (say, h) combine h polynomial coefficients $c_0,\ c_1,\ \ldots,\ c_{h-1}$ and use $z_h=c_0+c_1\,p+c_2\,p^2+\ldots c_{h-1}\,p^{h-1}$.

21.11.3 Efficient reduction modulo x^2+x+1, x^2-x-1 and x^2+1

We note that the polynomial $C:=x^2+x+1$ has the roots $\left(-1\pm\sqrt{-3}\right)/2$ so it is irreducible modulo p if -3 is not a square modulo p. The first few such primes p are

```
2 5 11 17 23 29 41 47 53 59 71 83 89 101 107 113 131 137 149 167 173 179 191 197 227 233 239
```

Multiplication modulo C costs only three scalar multiplications:

$$\begin{aligned}(a\,x+b)\,(A\,x+B) &= (a\,A)\,x^2+(a\,B+b\,A)\,x+(a\,A+b\,B) && (21.50a)\\ &\equiv (-a\,A+a\,B+b\,A)\,x+(-a\,A+b\,B) \mod x^2+x+1 && (21.50b)\\ &= ((b-a)\,(A-B)-b\,B)\,x+(-a\,A+b\,B) && (21.50c)\end{aligned}$$

Similarly for the polynomial $D=x^2-x-1$ that has the roots $\left(1\pm\sqrt{5}\right)/2$ so it is irreducible modulo p if 5 is not a square modulo p. The first few such primes are:

```
2 3 7 13 17 23 37 43 47 53 67 73 83 97 103 107 113 127 137 157 163 167 173 193 197 223 227 233
```

Multiplication modulo D again costs only three scalar multiplications:

$$\begin{aligned}(a\,x+b)\,(A\,x+B) &= (a\,A)\,x^2+(a\,B+b\,A)\,x+(a\,A+b\,B) && (21.51a)\\ &\equiv (a\,A+a\,B+b\,A)\,x+(a\,A+b\,B) \mod x^2-x-1 && (21.51b)\\ &= ((a+b)\,(A+B)-b\,B)\,x+(a\,A+b\,B) && (21.51c)\end{aligned}$$

For primes of the form $4\,k+3$ (-1 is not a square) one can use the reduction polynomial x^2+1 and

$$(a\,x+b)\,(A\,x+B) \equiv ((a+b)\,(A+B)-a\,A-b\,B)\,x+(a\,A-b\,B) \mod x^2+1 \quad (21.52)$$

The primes for which none of the given polynomials is irreducible are those where all of -3, 5, and -1 are squares. The first few are

```
61 109 181 229 241 349 409 421 541 601 661 709 769 829 1009 ...
```

For these primes one can try whether one of the polynomials $x^2+e\,x+f$ for $e,f\in\{0,\pm1,\pm2\}$ is irreducible.

21.11.4 An algorithm for primitive 2^j-th roots

For primes with the lowest k bits set ($p\equiv(2^k-1) \pmod{2^k}$) the largest power of two dividing the maximal order in $GF(p^2)$ equals $N=2^{k+1}$: $p=j\,2^k-1$ with j odd, so $p+1=j\,2^k$ and $p-1=j\,2^k-2=2\,(j\,2^{k-1}-1)$, thereby $p^2-1=2^{k+1}\,[j\,(j\,2^k-1)]$ where the term in square brackets is odd.

An algorithm for the construction of primitive 2^j-th roots in $GF(p^2)$ for $j=2,3,\ldots,a$ where 2^a is the largest power of two dividing p^2-1 is given in [GaoNB]:

Let $u_2:=0$ and for $j>2$ define

$$u_j := \begin{cases}\left((u_{j-1}+1)/2\right)^{(p+1)/4} & \text{if} \quad j<a\\ \left((u_{j-1}-1)/2\right)^{(p+1)/4} & \text{if} \quad j=a\end{cases} \quad (21.53)$$

and (for $j=2,3,\ldots,a$)

$$v_j := \begin{cases}\left(+1-u_j^2\right)^{(p+1)/4} & \text{if} \quad j<a\\ \left(-1-u_j^2\right)^{(p+1)/4} & \text{if} \quad j=a\end{cases} \quad (21.54)$$

where all operations are modulo p. Then $u_j+i\,v_j$ is a primitive 2^j-th root of unity in $GF(p^2)$.

For example, with $p=127$ we get

```
j:       u_j  v_j
2:    ord(0 + i*1) = 4
3:    ord(8 + i*8) = 8
4:    ord(103 + i*21) = 16
5:    ord(68 + i*87) = 32
6:    ord(15 + i*41) = 64
7:    ord(32 + i*82) = 128
8:    ord(98 + i*38) = 256
```

For Mersenne primes $p = 2^e - 1$ one has $p^2 - 1 = (p+1)(p-1) = 2^e(2^e - 2) = 2^{e+1}(2^{e-1} - 1) =: 2^a k$ where k is odd. The highest power of two for which a primitive root exists is $a = 2^{e+1}$ which checks with our example where $p = 127 = 2^7 - 1$.

21.11.5 Primitive 2^j-th roots with Mersenne primes

For Mersenne primes $p = 2^e - 1$ a primitive root of order 2^{e+1} can be constructed more directly: First compute $\sqrt{-3} = 3^{(p+1)/4} = 3^{2^{e-2}}$ by squaring $e-2$ times. Then compute $1/\sqrt{2} = 2^{(e-1)/2}$ which does not require modular reduction. Now

$$z \;:=\; \frac{1}{\sqrt{2}}\left(1+\sqrt{3}\right) \;=\; \frac{1}{\sqrt{2}}\left(1+i\sqrt{-3}\right) \tag{21.55}$$

is an element of order 2^{e+1}. We have $z^2 = 2 + \sqrt{3} = 2 + i\sqrt{-3} = H_1 + i\sqrt{-3}\,U_1$ and, more general

$$z^{2^k} \;=\; H_{2^{k-1}} + i\sqrt{-3}\,U_{2^{k-1}} \qquad (1 \le k) \tag{21.56}$$

Figure 21.17 shows the values of the successive 2^k-th powers of z in $GF(p^2)$ where $p = 2^{17} - 1$.

```
p =  131071  = 2^17-1
r3 =   43811     r3^2 =  -3

k:    h + i*w =    z^(2^k)            =   h + i*r3*  u
0:    h + i*w =    +256 + i* -56490   =   h + i*r3*    +256    ord(.) = 2^18
1:    h + i*w =      +2 + i* +43811   =   h + i*r3*      +1    ord(.) = 2^17
2:    h + i*w =      +7 + i* +44173   =   h + i*r3*      +4    ord(.) = 2^16
3:    h + i*w =     +97 + i* -36933   =   h + i*r3*     +56    ord(.) = 2^15
4:    h + i*w =  +18817 + i* +43903   =   h + i*r3*  +10864    ord(.) = 2^14
5:    h + i*w =  -17636 + i* -35524   =   h + i*r3*  +45327    ord(.) = 2^13
6:    h + i*w =   -5975 + i* -36232   =   h + i*r3*  +30114    ord(.) = 2^12
7:    h + i*w =  -32446 + i* +44887   =   h + i*r3*  +58666    ord(.) = 2^11
8:    h + i*w =  -38713 + i* -16371   =   h + i*r3*   +3123    ord(.) = 2^10
9:    h + i*w =  +61109 + i* -46595   =   h + i*r3*  +24597    ord(.) = 2^9
10:   h + i*w =  +63110 + i* +25098   =   h + i*r3*  -48310    ord(.) = 2^8
11:   h + i*w =  +35245 + i* +14561   =   h + i*r3*   -3138    ord(.) = 2^7
12:   h + i*w =  -30756 + i* -12111   =   h + i*r3*  +50228    ord(.) = 2^6
13:   h + i*w =  -15743 + i* -35732   =   h + i*r3*  -19124    ord(.) = 2^5
14:   h + i*w =  -26425 + i* -55712   =   h + i*r3*   -1910    ord(.) = 2^4
15:   h + i*w =    -256 + i*   +256   =   h + i*r3*  +18830    ord(.) = 2^3
16:   h + i*w =       0 + i*     -1   =   h + i*r3*  +58294    ord(.) = 2^2
17:   h + i*w =      -1 + i*      0   =   h + i*r3*       0    ord(.) = 2^1
18:   h + i*w =      +1 + i*      0   =   h + i*r3*       0    ord(.) = 2^0
```

Figure 21.17: Elements of order 2^j in $GF(p^2)$ where $p = 2^{17} - 1$ is a Mersenne prime.

The sequences $H = 2, 7, 97, 18817, \ldots$ and $U = 1, 4, 56, 10864, \ldots$ are those which appear in the Lucas-Lehmer test. The order of z^{2^k} is 2^{e+1-k}.

We have $H_{2j} = 2H_j^2 - 1$ and $H_j^2 - 3U_j^2 = 1$, thereby

$$H_{2j} \;=\; H_j^2 + 3U_j^2 \tag{21.57a}$$
$$U_{2j} \;=\; 2H_jU_j \tag{21.57b}$$

These are the index doubling formulas for the convergents of the continued fraction of $\sqrt{3}$.

21.11.6 Cosine and sine in $GF(p^2)$

Let z be an element of order n in $GF(p^2)$, we would like to identify z with $\exp(2\,i\,\pi/n)$ and determine the values equivalent to $\cos(2\,\pi/n)$ and $\sin(2\,\pi/n)$ with complex numbers over $\mathbb{R}$. One can set

$$\cos\frac{2\,\pi}{n} \equiv \frac{z^2+1}{2\,z} \tag{21.58}$$

$$i\,\sin\frac{2\,\pi}{n} \equiv \frac{z^2-1}{2\,z} \tag{21.59}$$

For the choice of sin and cos both relations $\exp(x) = \cos(x) + i\,\sin(x)$ and $\sin(x)^2 + \cos(x)^2 = 1$ should hold. The first check is trivial: $\frac{z^2+1}{2\,z} + \frac{z^2-1}{2\,z} = z$. The second is also easy if we write i for some element that is the square root of -1: $(\frac{z^2+1}{2\,z})^2 + (\frac{z^2-1}{2\,z\,i})^2 = \frac{(z^2+1)^2-(z^2-1)^2}{4\,z^2} = 1$.

For the 2^j-th roots in $GF(127^2)$ we get

```
0:    u+i*v=1 + i*0
      cos=Mod(1, 127)      i*sin=0
1:    u+i*v=126 + i*0
      cos=Mod(126, 127)    i*sin=0
2:    u+i*v=0 + i*1
      cos=0                i*sin=Mod(1, 127)*i
3:    u+i*v=8 + i*8
      cos=Mod(8, 127)      i*sin=Mod(8, 127)*i
4:    u+i*v=103 + i*21
      cos=Mod(103, 127)    i*sin=Mod(21, 127)*i
5:    u+i*v=68 + i*87
      cos=Mod(68, 127)     i*sin=Mod(87, 127)*i
6:    u+i*v=15 + i*41
      cos=Mod(15, 127  )   i*sin=Mod(41, 127)*i
7:    u+i*v=32 + i*82
      cos=Mod(32, 127)     i*sin=Mod(82, 127)*i
8:    u+i*v=98 + i*38
      cos=Mod(38, 127)*i   i*sin=Mod(98, 127)
```

Note how the i swaps side with the element of highest order 2^a.

The simple construction allows to mechanically convert FFT implementations with explicit sin / cos-values into the corresponding NTTs.

The idea of expressing primitive roots as cosines and sines was taken from [apfloat].

21.11.7 Cosine and sine in $GF(p)$

What about primes of the form $p = 4k + 1$ that are used anyway for NTTs? The same construction works. The polynomial $x^2 + 1$ is reducible modulo $p = 4k + 1$, equivalently, -1 is a quadratic residue so its square root lies in $GF(p)$. We could say: i is real modulo p if p is of the form $4k + 1$.

In the implementation [FXT: `class mod` in mod/mod.h] the cosine and sine values are computed from the primitive roots of order 2^j. The demo [FXT: demo mod/modsincos-demo.cc] generates the list of 2^j-th roots and inverse roots shown in figure 21.18 on the next page.

Again we can translate a given FFT implementation in a mechanical way.

An element modulo a prime $p = k \cdot 2^t + 1$ whose order equals 2^t can be found by the following algorithm even if the factorization of k is not known:
Choose random a where $1 < a < p-1$ and compute $s = a^k$, if $-1 = s^{2^{t-1}}$ then return s, else try another a.

The algorithm will terminate when the first element a is encountered whose order has the factor 2^t. Demonstration in pari/gp:

```
el2(k, t)=
{
    local(p, s);
    p=k*2^t+1;
    for(a=2, p-2,
```

```
modulus= 257 == 0x101
modulus is cyclic
modulus is prime
bits(modulus)= 8.0056245  == 9 - 0.99437545
phi(modulus)= 256 == 0x100 == 2^8
maxorder= 256 == 0x100
maxordelem= 3 == 0x3
max2pow= 8   (max FFT length = 2**8 == 256)
root2pow(max2pow)=3   root2pow(-max2pow)=86
sqrt(-1) =: i = 241

 8:    z=   3  = ( 173   +  87)  =  ( 173   + 107*i)
 7:    z=   9  = ( 233   +  33)  =  ( 233   +  14*i)
 6:    z=  81  = ( 123   + 215)  =  ( 123   +  99*i)
 5:    z= 136  = ( 188   + 205)  =  ( 188   + 196*i)
 4:    z= 249  = (  12   + 237)  =  (  12   + 194*i)
 3:    z=  64  = (  30   +  34)  =  (  30   +  30*i)
 2:    z= 241  = (   0   + 241)  =  (   0   +   1*i)
 1:    z= 256  = ( 256   +   0)  =  ( 256   +   0*i)
 0:    z=   1  = (   1   +   0)  =  (   1   +   0*i)
-1:    z= 256  = ( 256   +   0)  =  ( 256   +   0*i)
-2:    z=  16  = (   0   +  16)  =  (   0   + 256*i)
-3:    z= 253  = (  30   + 223)  =  (  30   + 227*i)
-4:    z=  32  = (  12   +  20)  =  (  12   +  63*i)
-5:    z= 240  = ( 188   +  52)  =  ( 188   +  61*i)
-6:    z= 165  = ( 123   +  42)  =  ( 123   + 158*i)
-7:    z= 200  = ( 233   + 224)  =  ( 233   + 243*i)
-8:    z=  86  = ( 173   + 170)  =  ( 173   + 150*i)
```

Figure 21.18: Roots of order 2^j modulo $p = 257 = 2^8 + 1$.

```
        s=Mod(a,p)^k;
        if( Mod(-1,p)==s^(2^(t-1)), return( s ); );
    );
}
```

With $p = 314151729239163 \cdot 2^{26} + 1$ the algorithm terminates after testing $a = 5$ (of order $(p-1)/3$) and returning $s = 18583781386455525528042$ whose order is indeed 2^{26}.

In general, if $p = u \cdot f + 1$, $\gcd(f, u) = 1$ and $f = \prod p_i^{e_i}$ is fully factored then an element of order f can be determined by testing random values a:

1. Take a random a and set $s = a^u$.
2. If $s^{f/p_i} \neq 1$ for all prime factors p_i of f then return s which is an element of order f, otherwise go to step 1.

21.11.8 Decomposing $p = 4\,k + 1$ as sum of two squares

The square root i of -1 can be used to find the representation of a prime $p = 4\,k + 1$ as a sum of two squares, $p = u^2 + v^2$, as follows:

1. Determine i where $i^2 = -1$ modulo p. If $i \geq p/2$ then set $i = p - i$.
2. Compute the continued fraction of p/i, it has the form $[a_0, a_1, \ldots, a_n, a_n, \ldots, a_1, a_0]$.
3. Compute the numerators of the $(n-1)$-st and the n-th convergent, P_{n-1} and P_n. Return $u = P_{n-1}$ and $v = P_n$.

Assume that $p = k \cdot 2^t + 1$ where $t \geq 2$. Use an element of order 2^t to find a square root of -1:

```
imag4k1(k, t)=
{ /* determine x so that x^2=-1 modulo p=k*2^t+1 */
    local(s);
    s = el2(k, t);
    s = s^(2^(t-2));
```

```
    return( s );
}
```

Then

```
sumofsquares(k, t)=
{ /* return [u,v] so that u^2+v^2==p =k*2^t+1 */
    local(i, s, p, cf, q, u, v);
    i = component( imag4k1(k, t), 2);
    p = k*2^t+1;
    if ( i>=p/2, i = p-i );
    cf = contfrac(p/i);
    cf = vector(length(cf)/2, j, cf[j]);
    q = contfracpnqn(cf);
    u = q[1, 1];  v = q[1, 2];
    return( [u, v] );
}
```

An example, for $p = 2281$ we get

```
   i = 1571  \\ square root of -1
   i =  710  \\ choose smaller square root
   cf = [3, 4, 1, 2, 2, 1, 4, 3]  \\ == contfrac(2281/710)
   cf = [3, 4, 1, 2]  \\ first half on contfrac

  q = contfracpnqn(cf) =
    [45 16]  \\ == [P_4, P_3]
    [14 5]   \\ == [Q_4, Q_3] (unused)

    u=45;  v=16;  \\ u^2 + v^2 = 2025 + 256 = 2281
```

The brute force way to determine u and v with $n = u^2 + v^2$ is to check (for $v = 1, 2, \ldots, \lfloor\sqrt{n}\rfloor$) whether $n - v^2$ is a perfect square. If so, return $u = \sqrt{n - v^2}$ and v:

```
sumofsquares_naive(k, t)=
{ /* return [u,v] so that u^2+v^2==p =k*2^t+1 */
    local(n, w);
    n = k*2^t+1;
    for (v=1, floor(sqrt(n)),  \\ search until n-v^2 is a square
        w = n-v^2;
        if ( issquare(w), return( [round(sqrt(w)), v] ) );
    );
    return ( 0 );  \\ not the sum of two squares
}
```

The routine needs at most $\lfloor\sqrt{n}\rfloor$ steps which renders it rather useless for n large: With the prime $n = 314151729239163 \cdot 2^{26} + 1 \approx 2 \cdot 10^{22}$ we have $n = u^2 + v^2$ where $u = 132599472793$ and $v = 59158646772$ and the routine would need v steps to find the solution. The continued fraction based routine finds the solution immediately.

A space saving version

An algorithm that avoids storing the continued fraction comes from the observation that u and v appear in the calculation of $\gcd(p, i)$. With

```
gcd_print(p, i)=
{
    local( t, s );
    if ( p<i,  t=p; p=i; i=t; );
    s = floor( sqrt(p) );
    while ( i,
        print ("  ", p, "  ", i);
        t = p % i;  p = i;  i = t;
    );
}
```

we get (for $p = 2281$ and $i = 710$):

```
     u    v
  2281  710
   710  151
   151  106
   106   45
    45   16 <--=
```

```
  16   13
  13    3
   3    1
```

The searched pair is the first where $u^2 < p$. The resulting routine is

```
sumofsquares_gcd(k, t)=
{ /* return [u,v] so that u^2+v^2==p =k*2^t+1 */
    local(s, p, i, w);
    i = component( imag4k1(k, t), 2);
    p = k*2^t+1;
    if ( i>=p/2, i = p-i );
    w = floor( sqrt(p) );  \\ need great enough precision
    while ( i,
        if ( p<=w, return( [p,i] ) );
        t = p % i;  p = i;  i = t;
    );
    return( [0,0] ); \\ failure (presicion loss in sqrt() ?)
}
```

21.12 Solving $x^2 - d\,y^2 = \pm 1$

Simple continued fractions can be used to find integer solutions of the equations

$$x^2 - d\,y^2 = +1 \tag{21.60a}$$
$$x^2 - d\,y^2 = -1 \tag{21.60b}$$

Equation 21.60a is usually called the *Pell equation*. The name *Bhaskara equation* (or *Brahmagupta-Bhaskara equation*, used in [NTC]) seems more appropriate as Brahmagupta (ca. 600 AD) and Bhaskara (ca. 1100 AD) were the first to study and solve this equation.

The convergents P_k/Q_k of the continued fraction of $\sqrt{d}$ are close to $\sqrt{d}$: $(P_k/Q_k)^2 \approx d$. If we define $e_k := P_k^2 - d\,Q_k^2$ then solutions of 21.60a correspond to $e_k = +1$, solutions of 21.60b to $e_k = -1$.

As an example we set $d = 53$. The continued fraction of $\sqrt{53}$ is

$$\mathrm{CF}(\sqrt{53}) = [7,\ 3,1,1,3,14,\ 3,1,1,3,14,\ 3,1,1,3,14,\ \ldots] \tag{21.61a}$$
$$= [7,\ \overline{3,1,1,3,14}] \tag{21.61b}$$

We observe that the sequence is periodic after the initial term and the last term of the period is twice the initial term. Moreover, disregarding the term 14, the terms in the period form a palindrome. These properties actually hold for all simple continued fractions of square roots $\sqrt{d}$ with d not a perfect square, for the proofs the reader is referred to either [Olds] or [NTC].

The table shown in figure 21.19 on the facing page gives the first convergents P_k/Q_k together with $e_k := P_k^2 - d\,Q_k^2$ ($d = 53$). The entry for $k = 4$ corresponds to the smallest solution (x, y) of $x^2 - 53\,y^2 = -1$: $182^2 - 53 \cdot 25^2 = -1$. Entry $k = 9$ corresponds to $66249^2 - 53 \cdot 9100^2 = +1$, the smallest nontrivial solution to $x^2 - 53\,y^2 = +1$ (the trivial solution is $(P_{-1}, Q_{-1}) = (1, 0)$).

Let $(x, y) = (P_m, Q_m)$ be a solution of relation 21.60b ($e_m = -1$) then $(x, y) := (P_p, Q_p)$ where

$$P_p := P_m^2 + d\,Q_m^2 \tag{21.62a}$$
$$Q_p := 2\,P_m\,Q_m \tag{21.62b}$$

is a solution of 21.60a ($e_p = +1$). If (P_m, Q_m) is the smallest solution with $e := e_m = -1$ then (P_p, Q_p) is the smallest solution with $e_p = e^2 = +1$. If (P_m, Q_m) is the smallest solution with $e = +1$ then (P_p, Q_p) is the second smallest solution with $e_p = +1$.

With our example: $(P_m, Q_m) = (P_4, Q_4) = (182, 25)$ and

$$P_p := 182^2 + 53 \cdot 25^2 = 66249 = P_9 \tag{21.63a}$$
$$Q_p := 2 \cdot 182 \cdot 25 = 9100 = Q_9 \tag{21.63b}$$

k	a_k	P_k	Q_k	$e_k := P_k^2 - d\,Q_k^2$
-1	-	0	1	+1
0	7	7	1	-4
1	3	22	3	+7
2	1	29	4	-7
3	1	51	7	+4
4	3	182	25	-1
5	14	2599	357	+4
6	3	7979	1096	-7
7	1	10578	1453	+7
8	1	18557	2549	-4
9	3	66249	9100	+1
10	14	946043	129949	-4
11	3	2904378	398947	+7
12	1	3850421	528896	-7
13	1	6754799	927843	+4
14	3	24114818	3312425	-1
15	14	344362251	47301793	+4

Figure 21.19: The first convergents P_k/Q_k of the continued fraction of $\sqrt{53}$ together with quantity $e_k := P_k^2 - d\,Q_k^2$.

If the period length of the continued fractions equals l then $m = l - 1$ and $p = 2\,l - 1$. Alternative expressions for P_p are

$$P_p \;=\; 2\,P_m^2 - e \;=\; 2\,d\,Q_m^2 + e \tag{21.64}$$

This can be seen as follows: $P_p + 0 = P_p + e - e = (P_m^2 + d\,Q_m) + (P_m^2 - d\,Q_m) - e$ and (second relation) $P_m^2 = d\,Q_m^1 + e$.

Further (P_t, Q_t) where

$$P_t \;:=\; P_m\,\left(P_m^2 + 3\,d\,Q_m^2\right) \tag{21.65a}$$
$$Q_t \;:=\; Q_m\,\left(3\,P_m^2 + d\,Q_m^2\right) \tag{21.65b}$$

is the second smallest solution with $e_t = e^3 = -1$. With our example $P_t = 24114818$ and $Q_t = 3312425$ (and $t = 14 = 3\,l - 1$). Alternative forms for P_t and Q_t are

$$P_t \;:=\; P_m\,\left(4\,P_m^2 - 3\,e\right) \tag{21.66a}$$
$$Q_t \;:=\; Q_m\,\left(4\,P_m^2 - e\right) \;=\; Q_m\,\left(4\,d\,Q_m^2 + 3\,e\right) \tag{21.66b}$$

Note that relations 21.62a and 21.62b are just the numerator and denominator of the second order iteration for $\sqrt{d}$ (relation 15.20a on page 402). Relations 21.65a and 21.65b correspond to the third order iteration (relation 15.20b on page 402).

With $d = 19$ we get the continued fraction

$$\mathrm{CF}(\sqrt{19}) \;=\; [4,\; 2,1,3,1,2,8,\; 2,1,3,1,2,8,\; 2,1,3,1,2,8,\; \ldots] \tag{21.67a}$$
$$\;=\; [4,\; \overline{2,1,3,1,2,8}] \tag{21.67b}$$

with period length $l = 6$. The corresponding table (figure 21.20 on the next page) contains solutions with $e_k = +1$ but none with $e_k = -1$:

Let e correspond the minimal nontrivial solution of $x^2 - d\,y^2 = \pm 1$, if $e = +1$ then no solution for $x^2 - d\,y^2 = -1$ exists. Nontrivial solutions with $e = +1$ always exist, solutions with $e = -1$ only exist

k	a_k	P_k	Q_k	$e_k := P_k^2 - d\,Q_k^2$
-1	-	0	1	+1
0	4	4	1	-3
1	2	9	2	+5
2	1	13	3	-2
3	3	48	11	+5
4	1	61	14	-3
5	2	170	39	+1
6	8	1421	326	-3
7	2	3012	691	+5
8	1	4433	1017	-2
9	3	16311	3742	+5
10	1	20744	4759	-3
11	2	57799	13260	+1
12	8	483136	110839	-3

Figure 21.20: The first convergents P_k/Q_k of the continued fraction of $\sqrt{19}$ together with quantity $e_k := P_k^2 - d\,Q_k^2$.

when the period length l of the continued fraction of $\sqrt{d}$ is odd. The period length l is always even for primes of the form $p = 4\,k+1$ and never for numbers of the form $n = 4\,k+3$ and $4\,k$. If any factor f_i of d is of the form $f_i = 4\,k+3$ then no solution with $e = -1$ exists, because this would imply $x^2 \equiv -1 \bmod f_i$ but -1 is never a quadratic residue modulo $f_i = 4\,k+3$ by relation 21.36 on page 536.

However, all prime factors being $f_i \equiv 1 \bmod 4$ does not guarantee that $e = -1$, the smallest examples are $205 = 5 \cdot 41$, $221 = 13 \cdot 17$, $305 = 5 \cdot 61$, and $377 = 13 \cdot 29$.

21.13 Multigrades *

Let a, $a+1$ be any two successive numbers, then $a^0 = (a+1)^0$ (wow!). Let a, $a+1$, $a+2$, $a+3$ any four successive numbers, then $a^1 + (a+3)^1 = (a+1)^1 + (a+2)^1$ (still not really exciting). The relations, as 'prototypes':

$$0^0 = 1^0 \tag{21.68a}$$

$$0^1 + 3^1 = 1^1 + 2^1 \tag{21.68b}$$

We also have

$$0^2 + 3^2 + 5^2 + 6^2 = 1^2 + 2^2 + 4^2 + 7^2 \tag{21.68c}$$

and

$$0^3 + 3^3 + 5^3 + 6^3 + 9^3 + 10^3 + 12^3 + 15^3 = 1^3 + 2^3 + 4^3 + 7^3 + 8^3 + 11^3 + 13^3 + 14^3 \tag{21.68d}$$

$$\begin{aligned} 0^4 + 3^4 + 5^4 + 6^4 + 9^4 + 10^4 + 12^4 + 15^4 + 17^4 + 18^4 + 20^4 + 23^4 + 24^4 + 27^4 + 29^4 + 30^4 = \\ = 1^4 + 2^4 + 4^4 + 7^4 + 8^4 + 11^4 + 13^4 + 14^4 + 16^4 + 19^4 + 21^4 + 22^4 + 25^4 + 26^4 + 28^4 + 31^4 \end{aligned} \tag{21.68e}$$

In general, for a fixed exponent $k > 0$, let $s = 2^{k+1}$ and partition the set $\{0, 1, 2, 3, \ldots, s-1\}$ into two sets S_0, S_1 so that S_0 contains all elements j $(0 \le j < s)$ with parity zero, S_1 the elements with parity one. Then for $d \in \mathbb{C}$,

$$\sum_{a \in S_0} (a+d)^k = \sum_{b \in S_1} (b+d)^k \tag{21.69}$$

An example, for $k = 2$ we have $S_0 = \{0, 3, 5, 6\}$, $S_1 = \{1, 2, 4, 7\}$ and so

$$(0+d)^2 + (3+d)^2 + (5+d)^2 + (6+d)^2 \quad = \quad (1+d)^2 + (2+d)^2 + (4+d)^2 + (7+d)^2 \quad (21.70)$$

Compare to relation 21.68c. The prototypes for the exponent k are also valid for all positive integer exponents less than k and the exponent zero.

The equations given are special solutions of the so-called *multigrades* or *Prouhet-Tarry-Escott* problem: Find two sets A and B such that for a given k

$$\sum_{a \in A} a^j \quad = \quad \sum_{b \in B} b^j \qquad j = 0, 1, 2, \ldots, k \quad (21.71)$$

Both sets have the same number n of elements. This is a (k, n)-multigrade. We found solutions of $(k, 2^k)$-multigrades with the special property that the union of both sets contains 2^k successive numbers. The shift invariance holds again, it is a property of all multigrade solutions.

Multiplying the symbolic entries in a prototype gives a valid prototype which again is shift invariant. This means that, for example, relation 21.68c can be interpreted as

$$\begin{aligned} &(0\,m+d)^k + (3\,m+d)^k + (5\,m+d)^k + (6\,m+d)^k \; = \\ &= \; (1\,m+d)^k + (2\,m+d)^k + (4\,m+d)^k + (7\,m+d)^k \end{aligned} \quad (21.72)$$

for $k \in \{0, 1, 2\}$, $d \in \mathbb{C}$ and $m \in \mathbb{C}$.

Adding and subtraction shifted versions of the prototypes gives more prototypes. We subtract from relation 21.68c the shifted by one version with swapped sides:

$$\begin{aligned} +\left(0^2 + 3^2 + 5^2 + 6^2 \right. \quad &= \quad \left. 1^2 + 2^2 + 4^2 + 7^2\right) \\ -\left(2^2 + 3^2 + 5^2 + 8^2 \right. \quad &= \quad \left. 1^2 + 4^2 + 6^2 + 7^2\right) \end{aligned}$$

Giving $2 \cdot 2^2 + 8^2 = 0^2 + 2 \cdot 6^2$. We can divide all symbolic entries by two and get:

$$2 \cdot 1^2 + 4^2 \quad = \quad 0^2 + 2 \cdot 3^2 \quad (21.73)$$

which really is

$$2 \cdot (1\,m+d)^k + (4\,m+d)^k \quad = \quad (0\,m+d)^k + 2 \cdot (3\,m+d)^k \quad (21.74)$$

for $k \in \{0, 1, 2\}$, $d \in \mathbb{C}$ and $m \in \mathbb{C}$.

21.14 Properties of the convergents of $\sqrt{2}$ *

We give some number theoretical properties of the convergents of the continued fraction of $\sqrt{2}$.

Define $P_0 := 1$, $P_1 := 1$, $P_n := 2\,P_{n-1} + P_{n-2}$, $Q_0 := 0$, $Q_1 := 1$ and $Q_n := 2\,Q_{n-1} + Q_{n-2}$. Then P_{n+1}/Q_{n+1} is the n-th convergent of $\sqrt{2}$:

n:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P_n:	1	3	7	17	41	99	239	577	1393	3363	8119	19601	47321	114243	275807
Q_n:	1	2	5	12	29	70	169	408	985	2378	5741	13860	33461	80782	195025

Pythagorean triples

Pythagorean triples are solutions of the Diophantine equation $a^2 + b^2 = c^2$. It is well-known that all primitive (a, b, c coprime) solutions can be obtained by choosing coprime x and y and setting $a = 2\,x\,y$, $b = x^2 - y^2$ and $c = x^2 + y^2$.

There are Pythagorean triples where a and b differ by one:

```
 n :  a^2 + b^2 = c^2
 1 :  0^2 + 1^2 = 1^2
 2 :  3^2 + 4^2 = 5^2
 3 :  20^2 + 21^2 = 29^2
 4 :  119^2 + 120^2 = 169^2
 5 :  696^2 + 697^2 = 985^2
 6 :  4059^2 + 4060^2 = 5741^2
 7 :  23660^2 + 23661^2 = 33461^2
 8 :  137903^2 + 137904^2 = 195025^2
 9 :  803760^2 + 803761^2 = 1136689^2
10 :  4684659^2 + 4684660^2 = 6625109^2
11 :  27304196^2 + 27304197^2 = 38613965^2
12 :  159140519^2 + 159140520^2 = 225058681^2
13 :  927538920^2 + 927538921^2 = 1311738121^2
14 :  5406093003^2 + 5406093004^2 = 7645370045^2
15 :  31509019100^2 + 31509019101^2 = 44560482149^2
```

Note that (equation 2) $3^2 + 2^4 = 5^2$ and (equation 4) $119^2 + 120^2 = 13^4$. The list can be obtained using $a_n = (P_{2\,n-1} - 1)/2$, $b_n = a_n + 1$ and $c_n = Q_{2\,n-1}$:

```
P(n)=( ([1,1]*[0,1;1,2]^n)[1] );
Q(n)=( ([0,1]*[0,1;1,2]^n)[1] );
pyth1(k)=
{ /* return [a,b,c] where a^2+b^2=c^2 and b=a+1 */
    local(p,q);
    p = (P(2*k-1)-1)/2;
    q = Q(2*k-1);
    return( [p, p+1, q] );
}
```

Or with the equivalent

```
pyth2(k)=
{ /* return [a,b,c] where a^2+b^2=c^2 and b=a+1 */
    local(x,y);
    x = P(k);  y=Q(k);
    return( [x^2-y^2, 2*x*y, x^2+y^2] );
}
```

Solutions of $x^4 \pm 4 \cdot z^4 = c^2 + 1$ and $x^4 - y^4 = c^2\,d$

If set $a_n = (P_{2\,n} - 1)/2$, $b_n = a_n + 1$ and $c_n = Q_{2\,n}$ then we get 'almost' Pythagorean triples:

```
    n :  a^2 + b^2 = c^2 + 1
    1 :  1^2 + 2^2 = 2^2 + 1
    2 :  8^2 + 9^2 = 12^2 + 1          // b=3^2  a=2*2^2
    3 :  49^2 + 50^2 = 70^2 + 1        // a=7^2  b=2*7^2
    4 :  288^2 + 289^2 = 408^2 + 1     // b=17^2 a=2*12^2
    5 :  1681^2 + 1682^2 = 2378^2 + 1  // a=41^2 b=2*41^2
```

As the comments suggest, all equations are of the form $x^4 + 4 \cdot z^4 = c^2 + 1$. We obtain solutions via the simple routine:

```
deq44(k)=
{ /* return [x,y,c] where x^4+4*z^4=c^2+1 */
    local(x,z,c);
    x = P(k);  z = Q(k);  c=2*x*z;
    return( [x, z, c] );
}
```

We get

```
 n :  x^4 + 4*z^4 = c^2 + 1
 1 :  1^4 + 4*1^4 = 2^2 + 1
 2 :  3^4 + 4*2^4 = 12^2 + 1
 3 :  7^4 + 4*5^4 = 70^2 + 1
 4 :  17^4 + 4*12^4 = 408^2 + 1
 5 :  41^4 + 4*29^4 = 2378^2 + 1
 6 :  99^4 + 4*70^4 = 13860^2 + 1
 7 :  239^4 + 4*169^4 = 80782^2 + 1
 8 :  577^4 + 4*408^4 = 470832^2 + 1
 9 :  1393^4 + 4*985^4 = 2744210^2 + 1
10 :  3363^4 + 4*2378^4 = 15994428^2 + 1
11 :  8119^4 + 4*5741^4 = 93222358^2 + 1
12 :  19601^4 + 4*13860^4 = 543339720^2 + 1
13 :  47321^4 + 4*33461^4 = 3166815962^2 + 1
14 :  114243^4 + 4*80782^4 = 18457556052^2 + 1
15 :  275807^4 + 4*195025^4 = 107578520350^2 + 1
```

Note that equation 7 is $239^4 + 4 \cdot 13^8 = 80782^2 + 1$. There are no solutions to the equation $x^4 + 4\,z^4 = c^2$ [RecrTN] so this is as close as one can get. Also, $x^4 - 4\,z^4 = c^2$ has no solutions but solutions of $x^4 - 4\,z^4 = c^2 + 1$ are obtained via

```
deq44m(k)=
{ /* return [x,y,c] where x^4-4*z^4=c^2+1 */
    local(x,z,c);
    x = P(2*k);  z = Q(2*k);  c=2*z;
    return( [x, z, c] );
}
```

The first few solutions are

```
     n :  x^4 - 4*z^4 = c^2 + 1
     1 :  3^4 - 4*2^4 = 4^2 + 1
     2 :  17^4 - 4*12^4 = 24^2 + 1
     3 :  99^4 - 4*70^4 = 140^2 + 1
     4 :  577^4 - 4*408^4 = 816^2 + 1
     5 :  3363^4 - 4*2378^4 = 4756^2 + 1
     6 :  19601^4 - 4*13860^4 = 27720^2 + 1
     7 :  114243^4 - 4*80782^4 = 161564^2 + 1
     8 :  665857^4 - 4*470832^4 = 941664^2 + 1
     9 :  3880899^4 - 4*2744210^4 = 5488420^2 + 1
    10 :  22619537^4 - 4*15994428^4 = 31988856^2 + 1
    11 :  131836323^4 - 4*93222358^4 = 186444716^2 + 1
    12 :  768398401^4 - 4*543339720^4 = 1086679440^2 + 1
    13 :  4478554083^4 - 4*3166815962^4 = 6333631924^2 + 1
    14 :  26102926097^4 - 4*18457556052^4 = 36915112104^2 + 1
    15 :  152139002499^4 - 4*107578520350^4 = 215157040700^2 + 1
```

Triangular square numbers

A triangular number is a number of the form $n\,(n+1)/2$. There are triangular numbers that are also squares. Let z_n be the n-th triangular number that is a square. Then

$$z_n \;=\; (P_n\,Q_n)^2 \;=\; 1,\, 36,\, 1225,\, 41616,\, 1413721,\, \ldots \tag{21.75}$$

```
? P(n)=(([1,1]*[0,1;1,2]^n)[1]);
? Q(n)=(([0,1]*[0,1;1,2]^n)[1]);
? zn(n)=(P(k)*Q(k))^2
? for(k=1,10,print(k," ",zn(k)));
   1 1
   2 36
   3 1225
   4 41616
   5 1413721
   6 48024900
   7 1631432881
   8 55420693056
   9 1882672131025
  10 63955431761796
```

A recurrence for z_n is given by $z_0 = 1$, $z_1 = 36$, $z_n = 34\,z_{n-1} - z_{n-2} + 2$, a closed form expression is $z_n = \frac{1}{32}\left((3+\sqrt{8})^{2n} + (3-\sqrt{8})^{2n} - 2\right)$. A simpler recurrence gives the square roots of z_n: Define $w_0 = 1$, $w_1 = 6$ and $w_n = 6\,w_{n-1} - w_{n-2}$. Then $w_n^2 = z_n$, the w_n-th square number equals z_n. The numbers w_n can be obtained as:

```
? wn(n)=return(([1,6]*[0,-1;1,6]^n)[1]);
? for(k=0,10,print1(" ",wn(k)))
 1 6 35 204 1189 6930 40391 235416 1372105 7997214 46611179
? for(k=0,10,print1(" ",wn(k)^2))
 1 36 1225 41616 1413721 48024900 1631432881 55420693056 1882672131025
```

Let $s_0 = 1$, $s_1 = 8$, $s_2 = 49$ and $s_n = 7\,s_{n-1} - 7\,s_{n-2} + s_{n-3}$. Then $s_n\,(s_n+1)/2 = z_n$, the s_n-th triangular number equals z_n. The s_n can be obtained as:

```
? sn(n)=return(([1,8,49]*[0,0,1;1,0,-7;0,1,7]^n)[1]);
? for(k=0,10,print1(" ",sn(k)))
 1 8 49 288 1681 9800 57121 332928 1940449 11309768 65918161
? tria(n)=n*(n+1)/2;
? for(k=0,10,print1(" ",tria(sn(k))))
 1 36 1225 41616 1413721 48024900 1631432881 55420693056 1882672131025
```

Solutions of the equations $s^3 + d^2 = w^4$

The sum of cubes up to n^3 is the square of the n-th triangular number:

$$\sum_{k=1}^{n} k^3 \quad = \quad \left(\frac{n\,(n+1)}{2} \right)^2 \tag{21.76}$$

So the difference of the squares of two consecutive triangular numbers is a cube:

$$\left(\frac{n\,(n+1)}{2} \right)^2 - \left(\frac{(n-1)\,n}{2} \right)^2 \quad = \quad n^3 \tag{21.77}$$

We write $T(n)$ for the n-th triangular number. Now, we have seen that $T(s_n)$, the s_n-th triangular number, is a square: $T(s_n) = w_n^2$. Squaring gives $T(s_n)^2 = w_n^4$. Subtracting $d_n := T(s_n - 1)$ from both sides gives $s_n^3 + d_n^2 = w_n^4$.

That is, the triples $[s_n,\, d_n,\, w_n]$ are a solutions of the Diophantine equation

$$s^3 + d^2 \quad = \quad w^4 \tag{21.78}$$

which is a somewhat surprising observation.

For the practical computation of d_n, we have $d_n = w_n^2 - s_n = z_n - s_n$.

```
deq(n)=
{ /* Return [s,d,w] so that  s^3 + d^2 = w^4 */
    local(s, t, w, d);
    s = sn(n);
    w = wn(n);
    d = w^2-s;  \\ == zn(n)-s;
    return( [s, d, w] );
}
```

The first few solutions of equation 21.78 of the given form are:

```
     n: s^3 + d^2 = w^4
     0: 1^3 + 0^2 = 1^4
     1: 8^3 + 28^2 = 6^4
     2: 49^3 + 1176^2 = 35^4
     3: 288^3 + 41328^2 = 204^4
     4: 1681^3 + 1412040^2 = 1189^4
     5: 9800^3 + 48015100^2 = 6930^4
     6: 57121^3 + 1631375760^2 = 40391^4
     7: 332928^3 + 55420360128^2 = 235416^4
     8: 1940449^3 + 1882670190576^2 = 1372105^4
     9: 11309768^3 + 63955420452028^2 = 7997214^4
    10: 65918161^3 + 2172601941851880^2 = 46611179^4
    11: 384199200^3 + 73804512448220400^2 = 271669860^4
    12: 2239277041^3 + 2507180832055219320^2 = 1583407981^4
    13: 13051463048^3 + 851703438401289936282 = 9228778026^4
    14: 76069501249^3 + 2893284510097771529376^2 = 53789260175^4
    15: 443365544448^3 + 98286503001614049040128^2 = 313506783024^4
```

The equations with odd index are of the form $8\,u^6 + d^2 = w^4$. The equations with even index are of the form $u^6 + d^2 = w^4$, corresponding to Pythagorean triples giving triangles with hypothenuse a square and one side a third power. For example (equation 2): $(7^3)^2 + (1176)^2 = (35^2)^2$ which is $7^2 + 24^2 = 5^4$ multiplied by 7^4.

The triangular square numbers were determined by Euler. That triangular square numbers yield solutions to $s^3 + d^2 = w^4$ was observed 1912 by H.B.Mathieu.

We note that there exist solutions that are not of the given form. A simple search reveals some of them, we use the routine

```
N=200;
{ for (s=1, N,  s3 = s^3;
    for (w=1, N,  w4 = w^4;
        t=w4-s3;
        if ( issquare(t) && (t!=0),
            d=sqrtint(t);
```

```
            g=gcd(s,gcd(d,w));
            print(s, "^3 + ", d, "^2 = ", w, "^4   [", g, "]");
        );
 ); ); }
```

It finds all solutions where $0 < s \leq 200$ and $0 < w \leq 200$:

```
/*  s^3 + d^2 = w^4    [gcd(s,d,w)] */
    8^3 + 28^2 = 6^4   [2]
    18^3 + 27^2 = 9^4   [9]
    23^3 + 6083^2 = 78^4   [1]
    36^3 + 63^2 = 15^4   [3]
    49^3 + 1176^2 = 35^4   [7]
    108^3 + 648^2 = 36^4   [36]
    108^3 + 3807^2 = 63^4   [9]
    126^3 + 2925^2 = 57^4   [3]
    128^3 + 1792^2 = 48^4   [16]
    135^3 + 4941^2 = 72^4   [9]
    136^3 + 4785^2 = 71^4   [1]
    143^3 + 433^2 = 42^4   [1]
```

Regarding triangular numbers, we note a nice generalization of the relation

$$\left(\sum_{k=1}^{n} k\right)^2 \;=\; \sum_{k=1}^{n} k^3 \tag{21.79}$$

which is a consequence of

$$\sum_{k=1}^{n} k \;=\; \frac{n\,(n+1)}{2} \;=\; T(n) \tag{21.80a}$$

$$\sum_{k=1}^{n} k^3 \;=\; \left(\frac{n\,(n+1)}{2}\right)^2 \;=\; T(n)^2 \tag{21.80b}$$

Indeed,

$$\left(\sum_{d\backslash N} \nu(d)\right)^2 \;=\; \sum_{d\backslash N} \nu(d)^3 \tag{21.81}$$

where $\nu(d)$ counts the number of divisors of d. Relation 21.79 is the special case where $N = p^{n-1}$. The divisors of N are 1, p, p^2, …, p^{n-1} whose numbers of divisors in turn are 1, 2, 3, …, n.

We close with a relation that is taken from the marvelous book of A. H. Beiler [RecrTN]:

$$3\left(\sum_{k=1}^{n} T(k)\right)^3 \;=\; \sum_{k=1}^{n} T(k)^3 + 2\sum_{k=1}^{n} T(k)^4 \tag{21.82}$$

Chapter 22

Binary polynomials

This introduces chapter binary polynomials and the arithmetic operations on them. We further describe tests for irreducibility and primitivity. Finally, a method for the factorization of binary polynomials is given. Many of the algorithms shown can easily be implemented in hardware. The arithmetic operations of binary polynomials are the underlying methods for computations in binary finite fields which are treated in chapter 23. Another important application, the linear feedback shift registers, are described in chapter 6

A polynomial with coefficients in the field $GF(2) = \mathbb{Z}/2\mathbb{Z}$ (that is, 'coefficients modulo 2') is called a *binary polynomial*. The operations proceed as for usual polynomials except that the coefficients have to be reduced modulo two.

To represent a binary polynomial in a binary computer one uses words where the bits are set corresponding to the positions where the polynomial coefficients are one. We stick to the convention that the constant term goes to the least significant bit. It turns out that the arithmetic operations can be implemented quite easily in an efficient manner.

22.1 The basic arithmetical operations

Addition of binary polynomials is the XOR operation. Subtraction is the very same operation.

Multiplication of a binary polynomial by its independent variable x is simply a shift to the left.

The following routines are given in [FXT: file bpol/bitpolarith.h].

Multiplication and squaring

Multiplication of two polynomials A and B is identical to the usual (binary algorithm for) multiplication, except that no carry occurs:

```
inline ulong bitpol_mult(ulong a, ulong b)
// Return  A * B
{
    ulong t = 0;
    while ( b )
    {
        if ( b & 1 )  t ^= a;
        b >>= 1;
        a <<= 1;
    }
    return  t;
}
```

As for integer multiplication with the C-type `unsigned long`, the result will silently overflow if $deg(A) + deg(B)$ is equal to or greater than the word length (`BITS_PER_LONG`). If the operation `t^=a;` was replaced with `t+=a;` the ordinary (integer) product would be returned [FXT: demo gf2n/bitpolmult-demo.cc]:

```
        1..11.111 * 11.1.1
                          product as bitpol    ordinary product
        1..11.111   1    t= .....1..11.111   c= ......1..11.111
                    0
       1..11.111..  1    t= ...1.1111.1.11   c= ....11....1..11
                    0
     1..11.111....  1    t= .1.11.1..11.11   c= ..11..11.....11
   1..11.111.....   1    t= 1..1....1111.11  c= 1.......11...11
```

When a binary polynomial $p = \sum_{k=0}^{d} a_k\, x^k$ is squared, the result equals $p^2 = \sum_{k=0}^{d} a_k\, x^{2k}$:

```
           1..11.111 * 1..11.111
           1..11.111   1    t= ........1..11.111
          1..11.111.   1    t= .......11.1.11..1
         1..11.111..   1    t= ......1111....1.1
                       0
       1..11.111....   1    t= ....1.1..1111.1.1
      1..11.111.....   1    t= ...11..1....1.1.1
                       0
                       0
  1..11.111.........   1    t= 1.....1.1...1.1.1
t= 1.....1.1...1.1.1
```

Thereby one can, instead of using `bitpol_mult(a, a)`, square by using the relation

$$\sum_{k=0}^{d} c_k\, x^k \quad = \quad \sum_{k=0}^{d} c_k\, x^{2k} \tag{22.1}$$

Which can be verified by repeated application of $(a+b)^2 = a^2 + 2\,a\,b + b^2 = a^2 + b^2$. So we just have to move the bits from position k to position $2k$:

```
inline ulong bitpol_square(ulong a)
// Return A * A
{
    ulong t = 0,  m = 1UL;
    while ( a )
    {
        if ( a&1 )  t ^= m;
        m <<= 2;
        a >>= 1;
    }
    return t;  // == bitpol_mult(a, a);
}
```

This version will unlikely give a speedup, but the equivalent function `bit_zip()` (see section 1.15 on page 32) can be a win if the degree of a is not too small.

Optimization of the squaring and multiplication routines

The routines for multiplication and squaring can be optimized by partially unrolling which avoids branches. As given, the function is compiled to:

```
  0: 31 c9                xor    %ecx,%ecx
  2: 48 85 ff             test   %rdi,%rdi
  5: ba 01 00 00 00       mov    $0x1,%edx
  a: 74 1b                je     27 <_Z13bitpol_squarem+0x27>  /* a!=0 ? */

 10: 48 89 c8             mov    %rcx,%rax  /* loop start */
 13: 48 31 d0             xor    %rdx,%rax
 16: 40 f6 c7 01          test   $0x1,%dil
 1a: 48 0f 45 c8          cmovne %rax,%rcx  /* if ( a&1 ) ...
 1e: 48 c1 e2 02          shl    $0x2,%rdx
 22: 48 d1 ef             shr    %rdi
 25: 75 e9                jne    10 <_Z13bitpol_squarem+0x10>  /* a!=0 ? */

 27: 48 89 c8             mov    %rcx,%rax
 2a: c3                   retq
```

The `if()`-statement does not cause a branch so we unroll the contents of the loop 4-fold. Further, we move the `while()` statement to the end the loop to avoid the initial branch:

```
inline ulong bitpol_square(ulong a)
{
    ulong t = 0,  m = 1UL;
    do
    {
        if ( a&1 )  t ^= m;
        m <<= 2;  a >>= 1;
        if ( a&1 )  t ^= m;
        m <<= 2;  a >>= 1;
        if ( a&1 )  t ^= m;
        m <<= 2;  a >>= 1;
        if ( a&1 )  t ^= m;
        m <<= 2;  a >>= 1;
    }
    while ( a );
    return t;
}
```

Now we obtain much better machine code:

```
   0: 31 c9                 xor    %ecx,%ecx
   2: ba 01 00 00 00        mov    $0x1,%edx
   7: 48 89 c8              mov    %rcx,%rax /* loop start */
   a: 48 31 d0              xor    %rdx,%rax
   d: 40 f6 c7 01           test   $0x1,%dil
  11: 48 0f 45 c8           cmovne %rax,%rcx /* if () ... */
  15: 48 c1 e2 02           shl    $0x2,%rdx
  19: 48 d1 ef              shr    %rdi
  1c: 48 89 c8              mov    %rcx,%rax
  1f: 48 31 d0              xor    %rdx,%rax
  22: 40 f6 c7 01           test   $0x1,%dil
  26: 48 0f 45 c8           cmovne %rax,%rcx /* if () ... */
  2a: 48 c1 e2 02           shl    $0x2,%rdx
  2e: 48 d1 ef              shr    %rdi
  31: 48 89 c8              mov    %rcx,%rax
  34: 48 31 d0              xor    %rdx,%rax
  37: 40 f6 c7 01           test   $0x1,%dil
  3b: 48 0f 45 c8           cmovne %rax,%rcx /* if () ... */
  3f: 48 c1 e2 02           shl    $0x2,%rdx
  43: 48 d1 ef              shr    %rdi
  46: 48 89 c8              mov    %rcx,%rax
  49: 48 31 d0              xor    %rdx,%rax
  4c: 40 f6 c7 01           test   $0x1,%dil
  50: 48 0f 45 c8           cmovne %rax,%rcx /* if () ... */
  54: 48 c1 e2 02           shl    $0x2,%rdx
  58: 48 d1 ef              shr    %rdi
  5b: 75 aa                 jne    7 <_Z13bitpol_squarem+0x7> /* a!=0 ? */
  5d: 48 89 c8              mov    %rcx,%rax
  60: c3                    retq
```

The multiplication algorithm is optimized in the same way.

Exponentiation

With a multiplication (and squaring) function at hand, it is straightforward (see section 14.6 on page 395) to implement the algorithm for binary exponentiation:

```
inline ulong bitpol_power(ulong a, ulong e)
// Return A ** e
{
    if ( 0==e )  return 1;
    ulong s = a;
    while ( 0==(e&1) )
    {
        s = bitpol_square(s);
        e >>= 1;
    }
    a = s;
    while ( 0!=(e>>=1) )
    {
        s = bitpol_square(s);
        if ( e & 1 )  a = bitpol_mult(a, s);
    }
```

```
    return  a;
}
```

Note that overflow will occur even for moderate exponents.

Quotient and remainder

The remainder a modulo b division can be computed by initializing $A = a$ and subtracting $B = x^j \cdot b$ with $\deg(B) = \deg(A)$ from A at each step. The computation is finished as soon as $\deg b > \deg A$. As C-code:

```
inline ulong bitpol_rem(ulong a, ulong b)
// Return  R = A % B = A - (A/B)*B
// Must have: B!=0
{
    const ulong db = highest_bit_idx(b);
    ulong da;
    while ( db <= (da=highest_bit_idx(a)) )
    {
        if ( 0==a )  break;  // needed because highest_bit_idx(0)==highest_bit_idx(1)
        a ^= (b<<(da-db));
    }
    return  a;
}
```

The function `highest_bit_idx()` is given in section 1.6 on page 15. The following version may be superior if the degree of a is small or if no fast version of the function `highest_bit_idx()` is available:

```
    while ( b <= a )
    {
        ulong t = b;
        while ( (a^t) > t )  t <<= 1;
        // =^= while ( highest_bit(a) > highest_bit(t) )  t <<= 1;
        a ^= t;
    }
    return  a;
```

The quotient of two polynomials is computed by a function that does the computes the remainder and additionally keeps track of the quotient:

```
inline void bitpol_divrem(ulong a, ulong b, ulong &q, ulong &r)
// Set R, Q so that  A == Q * B + R.
// Must have b!=0.
{
    const ulong db = highest_bit_idx(b);
    q = 0;  // quotient
    ulong da;
    while ( db <= (da=highest_bit_idx(a)) )
    {
        if ( 0==a )  break;  // needed because highest_bit_idx(0)==highest_bit_idx(1)
        a ^= (b<<(da-db));
        q ^= (1UL<<(da-db));
    }
    r = a;
}
```

The division routine does the same computation but discards the remainder:

```
inline ulong bitpol_div(ulong a, ulong b)
// Return  Q = A / B
// Must have B!=0.
{
    const ulong db = highest_bit_idx(b);
    ulong q = 0;  // quotient
    ulong da;
    while ( db <= (da=highest_bit_idx(a)) )
    {
        if ( 0==a )  break;  // needed because highest_bit_idx(0)==highest_bit_idx(1)
        a ^= (b<<(da-db));
        q ^= (1UL<<(da-db));
    }
    return  q;
}
```

Greatest common divisor (GCD)

The polynomial greatest common divisor (GCD) can be computed with the Euclidean algorithm [FXT: file bpol/bitpolgcd.h]:

```
inline ulong bitpol_gcd(ulong a, ulong b)
// Return  polynomial gcd(A, B)
{
    if ( a<b )  { ulong t=a; a=b; b=t; }
    // here: b<=a
    while ( 0!=b )
    {
        ulong c = bitpol_rem(a, b);
        a = b;
        b = c;
    }
    return  a;
}
```

The binary GCD algorithm can be implemented as follows:

```
inline ulong bitpol_binary_gcd(ulong a, ulong b)
{
    if ( a < b )  swap2(a, b);
    if ( b==0 )  return a;

    ulong k = 0;
    while ( !((a|b)&1) )  // both divisible by x
    {
        k++;
        a >>= 1;
        b >>= 1;
    }
    while ( !(a&1) )  a >>= 1;
    while ( !(b&1) )  b >>= 1;

    while ( a!=b )
    {
        if ( a < b )  { ulong t=a; a=b; b=t; };  // swap if deg(A)<deg(B)
        ulong t = (a^b) >> 1;
        while ( !(t&1) )  t >>= 1;
        a = t;
    }

    return  a << k;
}
```

With a fast bit-scan instruction we can optimize the function:

```
inline ulong bitpol_binary_gcd(ulong a, ulong b)
// Return polynomial gcd(A, B)
{
    { // one (or both) of a, b zero?
        ulong ta = a&b,  to = a|b;
        if ( ta==to )  return to;
    }

    ulong ka = lowest_bit_idx(a);
    a >>= ka;
    ulong kb = lowest_bit_idx(b);
    b >>= kb;
    ulong k = ( ka<kb ? ka : kb );

    while ( a!=b )
    {
        if ( a < b )  { ulong t=a; a=b; b=t; }  // swap if deg(A)<deg(B)
        ulong t = (a^b) >> 1;
        a = (t >> lowest_bit_idx(t));
    }
    return  a << k;
}
```

Note that the comment

```
        if ( a < b )  { ulong t=a; a=b; b=t; };  // swap if deg(A)<deg(B)
```

is not strictly correct as the swap can also happen with $\deg(a) = \deg(b)$ but that does no harm.

Extended GCD and computation of the inverse

The algorithm for the computation of extended GCD (EGCD) is taken from [DEK2] [FXT: file bpol/bitpolgcd.h]:

```
inline ulong bitpol_egcd(ulong u, ulong v, ulong &iu, ulong &iv)
// Return u3 and set u1,v1 so that  gcd(u,v) == u3 == u*u1 + v*u2
{
    ulong u1 = 1,  u2 = 0;
    ulong v1 = 0,  v3 = v;
    ulong u3 = u,  v2 = 1;
    while ( v3!=0 )
    {
        ulong q = bitpol_div(u3, v3);  // == u3 / v3;

        ulong t1 = u1 ^ bitpol_mult(v1, q);  // == u1 - v1 * q;
        u1 = v1;  v1 = t1;

        ulong t3 = u3 ^ bitpol_mult(v3, q);  // == u3 - v3 * q;
        u3 = v3;  v3 = t3;

        ulong t2 = u2 ^ bitpol_mult(v2, q);  // == u2 - v2 * q;
        u2 = v2;  v2 = t2;
    }

    iu = u1;  iv = u2;
    return u3;
}
```

The routine can be optimized using `bitpol_divrem()`: Remove the lines

```
        ulong q = bitpol_div(u3, v3);  // == u3 / v3;
    [-snip-]
        ulong t3 = u3 ^ bitpol_mult(v3, q);  // == u3 - v3 * q;
```

and insert at the beginning of the body of the loop:

```
        ulong q, t3;
        bitpol_divrem(u3, v3, q, t3);
```

The routine computes the GCD g and two additional quantities i_u and i_v so that

$$g \;=\; u \cdot i_u + v \cdot i_v \tag{22.2}$$

When $g = 1$ we have

$$1 \;\equiv\; u \cdot i_u \bmod v \tag{22.3}$$

That is, i_u is the inverse of u modulo v. Thereby [FXT: file bpol/bitpolmodarith.h]:

```
inline ulong bitpolmod_inverse(ulong a, ulong c)
// Returns the inverse of A modulo C if it exists, else zero.
// Must have deg(A) < deg(C)
{
    ulong i, t;  // t unused
    ulong g = bitpol_egcd(a, c, i, t);
    if ( g!=1 )  i = 0;
    return i;
}
```

22.2 Multiplication for polynomials of high degree

We used the straightforward multiplication scheme whose asymptotic cost is $\sim N^2$ for polynomials of degree N. This is fine when working with polynomials of small degree. For the multiplication of two polynomials U and V both of (high, even) degree N write $U = U_0 + x^{N/2}\,U_1$, $V = V_0 + x^{N/2}\,V_1$ and use the scheme

$$U\,V \;=\; (1 + x^{N/2})\,U_0\,V_0 + x^{N/2}\,(U_1 - U_0)\,(V_0 - V_1) + (x^{N/2} + x^N)\,U_1\,V_1 \tag{22.4}$$

recursively. The resulting scheme is the *Karatsuba multiplication for polynomials* relation 14.4 on page 390 interpreted for polynomials (set $x^{N/2} = B$). Recursive application of the scheme leads to the asymptotic cost $\sim N^{\log_2(3)} \approx N^{1.585}$. When working with polynomials of high degree the implementation of the Karatsuba scheme is a must.

A generalization of the Karatsuba scheme is given in [106] (see also [107]), it does not lead asymptotically better schemes than $\sim N^{\log_2(3)}$ but has a simple structure and avoids all multiplications by constants[1]. We give the scheme for degree-2 (3-term) polynomials, recursive application for 3^n-term polynomials should be straightforward. Let

$$A = a_2\,x^2 + a_1\,x + a_0 \tag{22.5a}$$
$$B = b_2\,x^2 + b_1\,x + b_0 \tag{22.5b}$$
$$C = A\,B = c_4\,x^4 + c_3\,x^3 + c_2\,x^2 + c_1\,x + c_0 \tag{22.5c}$$

We define

$$d_{0,0} = a_0\,b_0 \tag{22.6a}$$
$$d_{1,1} = a_1\,b_1 \tag{22.6b}$$
$$d_{2,2} = a_2\,b_2 \tag{22.6c}$$
$$d_{0,1} = (a_0 + a_1)\,(b_0 + b_1) \tag{22.6d}$$
$$d_{0,2} = (a_0 + a_2)\,(b_0 + b_2) \tag{22.6e}$$
$$d_{1,2} = (a_1 + a_2)\,(b_1 + b_2) \tag{22.6f}$$

Then

$$c_0 = d_{0,0} \tag{22.7}$$
$$c_1 = d_{0,1} - d_{0,0} - d_{1,1} \tag{22.8}$$
$$c_2 = d_{0,2} - d_{0,0} - d_{2,2} + d_{1,1} \tag{22.9}$$
$$c_3 = d_{1,2} - d_{1,1} - d_{2,2} \tag{22.10}$$
$$c_4 = d_{2,2} \tag{22.11}$$

The scheme involves 6 multiplications and 13 additions. Recursive application leads to the asymptotic cost $\sim N^{\log_3(6)} \approx 1.6309$ which is slightly worse than for the 2-term scheme. However, applying this scheme first for a polynomial with $N = 3 \cdot 2^n$ terms and then using the Karatsuba scheme recursively can be advantageous.

We generalize the method for n-term polynomials and denote the scheme by KA-n. The 2-term scheme KA-2 is the Karatsuba algorithm. With

$$A = \sum_{k=0}^{n-1} a_k\,x^k \tag{22.12a}$$
$$B = \sum_{k=0}^{n-1} b_k\,x^k \tag{22.12b}$$
$$C = A\,B =: \sum_{k=0}^{2n-2} c_k\,x^k \tag{22.12c}$$

define

$$d_{s,s} := a_s\,b_s \qquad \text{for } s = 0,\,1,\,\ldots,\,n-1 \tag{22.13a}$$
$$d_{s,t} := (a_s + b_s)\,(a_t + b_t) \quad \text{for } s + t = i,\ t > s \geq 0,\ 1 \leq i \leq 2\,n - 3 \tag{22.13b}$$
$$c_i^* = \sum_{\substack{s+t=i \\ t>s\geq 0}} d_{s,t} - \sum_{\substack{s+t=i \\ n-1\geq t>s\geq 0}} (d_{s,s} + d_{t,t}) \tag{22.13c}$$

[1] The asymptotically better Toom-Cook method involves multiplications by constants.

Then

$$c_0 = d_{0,0} \tag{22.14a}$$
$$c_{2\,n-2} = d_{n-1,n-1} \tag{22.14b}$$

and for $0 < i < 2\,n-2$:

$$c_i = \begin{cases} c_i^* & \text{if} \quad i \text{ odd} \\ c_i^* + d_{i/2,i/2} & \text{else} \end{cases} \tag{22.14c}$$

The Karatsuba scheme is obtained for $n = 2$.

We give pari/gp code whose output is the KA-n algorithm for given n. We need to create symbols '`ak`' (for a_n), '`bk`', and so on:

```
fa(k)=eval(Str("a" k))
fb(k)=eval(Str("b" k))
fc(k)=eval(Str("c" k))
fd(k,j)=eval(Str("d" k "" j))
```

For example, we can create a symbolic polynomial of degree 3:

```
? sum(k=0,3, fa(k) * x^k)
  a3*x^3 + a2*x^2 + a1*x + a0
```

The next routine generates the definitions of all $d_{s,t}$. It returns the number of multiplications involved:

```
D(n)=
{
    local(mct);
    mct = 0; \\ count multiplications
    for (i=0, n-1,  mct+=1; print(fd(i,i), " = ", fa(i), " * ", fb(i) ) );
    for (t=1, n-1,
        for (s=0, t-1,
            mct += 1;
            print(fd(s,t), " = (", fa(s)+fa(t), ") * (", fb(s)+fb(t), ")" ) ;
        );
    );
    return(mct);
}
```

For $n = 3$ the routine would print

```
d00 = a0 * b0
d11 = a1 * b1
d22 = a2 * b2
d01 = (a0 + a1) * (b0 + b1)
d02 = (a0 + a2) * (b0 + b2)
d12 = (a1 + a2) * (b1 + b2)
```

The following routine prints c_i, the coefficient of the product, in terms of several $d_{s,t}$. It returns the additions involved:

```
C(i, n)=
{
    local(N, s, act);
    act = -1; \\ count additons
    print1(fc(i), " = ");
    for (s=0, i-1,
        t = i - s;
        if ( (t>s) && (t<n),
             act += 3;
             print1(" + ", fd(s,t));
             print1(" - ", fd(s,s));
             print1(" - ", fd(t,t));
        );
    );
    if ( 0==i%2,  act+=1; print1(" + ", fd(i/2,i/2)) );
    print();
    return( act );
}
```

It has to be called for all i where $0 \le i \le 2n-2$. The algorithm is generated as follows:

```
KA(n)=
{
    local(mct, act);
    act = 0; \\ count additons
    mct = 0; \\ count multiplications
    mct = D(n);  \\ generate definitions for the d_{s,t}
    \\ generate rules for computation of c_i in terms of d_{s,t}:
    for (i=0, 2*n-2,  act+=C(i,n) );
    act += n*(n-1); \\ additions when setting up d(i,j) for i!=j
    return( [mct, act] );
}
```

With $n = 3$ the output is

```
c0 =  + d00
c1 =  + d01 - d00 - d11
c2 =  + d02 - d00 - d22 + d11
c3 =  + d12 - d11 - d22
c4 =  + d22
```

```
    d00 = a0 * b0
    d11 = a1 * b1
    d22 = a2 * b2
    d33 = a3 * b3
    d44 = a4 * b4
    d01 = (a0 + a1) * (b0 + b1)
    d02 = (a0 + a2) * (b0 + b2)
    d12 = (a1 + a2) * (b1 + b2)
    d03 = (a0 + a3) * (b0 + b3)
    d13 = (a1 + a3) * (b1 + b3)
    d23 = (a2 + a3) * (b2 + b3)
    d04 = (a0 + a4) * (b0 + b4)
    d14 = (a1 + a4) * (b1 + b4)
    d24 = (a2 + a4) * (b2 + b4)
    d34 = (a3 + a4) * (b3 + b4)

    c0 =  + d00
    c1 =  + d01 - d00 - d11
    c2 =  + d02 - d00 - d22 + d11
    c3 =  + d03 - d00 - d33 + d12 - d11 - d22
    c4 =  + d04 - d00 - d44 + d13 - d11 - d33 + d22
    c5 =  + d14 - d11 - d44 + d23 - d22 - d33
    c6 =  + d24 - d22 - d44 + d33
    c7 =  + d34 - d33 - d44
    c8 =  + d44
```

Figure 22.1: Code for the algorithm KA-5.

Now we generate the definitions for the KA-5 algorithm:

```
n=5 /* n terms, degree=n-1 */
default(echo, 0);
KA(n);
```

We obtain the algorithm KA-5 shown in figure 22.1. The format is valid pari/gp input, so we add a few lines that print code to check the algorithm:

```
print("A=",sum(k=0,n-1, fa(k) * x^k))
print("B=",sum(k=0,n-1, fb(k) * x^k))
print("/* direct computation of the product: */")
print("C=A*B")
print("/* Karatsuba computation of the product: */")
print("K=",sum(k=0,2*n-2, fc(k) * x^k))
print("qq=K-C")
print("print( if(0==qq, \"OK.\", \" **** OUCH!\") )")
```

This gives for $n = 5$:

```
    A=a4*x^4 + a3*x^3 + a2*x^2 + a1*x + a0
    B=b4*x^4 + b3*x^3 + b2*x^2 + b1*x + b0
    /* direct computation of the product: */
    C=A*B
    /* Karatsuba computation of the product: */
    K=c8*x^8 + c7*x^7 + c6*x^6 + c5*x^5 + c4*x^4 + c3*x^3 + c2*x^2 + c1*x + c0
    qq=K-C
    print( if(0==qq, "OK.", " **** OUCH!") )
```

We can feed the output into another pari/gp session to verify the algorithm. We use the option '`-f`' that prevents colorization of the output which would confuse the verification process:

```
gp -f  < karatsuba-n.gp  |  gp
```

We obtain (shortened and comments added):

```
/* definitions of d(s,t): */
b0*a0
b1*a1
b2*a2
b3*a3
b4*a4
(b0 + b1)*a0 + (a1*b0 + b1*a1)
(b0 + b2)*a0 + (a2*b0 + b2*a2)
[--snip--]
(b3 + b4)*a3 + (a4*b3 + b4*a4)

/* the c_i in terms of d(s,t), evaluated: */
b0*a0
b1*a0 + a1*b0
b2*a0 + (a2*b0 + b1*a1)
b3*a0 + (a3*b0 + (b2*a1 + a2*b1))
b4*a0 + (a4*b0 + (b3*a1 + (a3*b1 + b2*a2)))
b4*a1 + (a4*b1 + (b3*a2 + a3*b2))
b4*a2 + (a4*b2 + b3*a3)
b4*a3 + a4*b3
b4*a4

/* polynomials: */
a4*x^4 + a3*x^3 + a2*x^2 + a1*x + a0
b4*x^4 + b3*x^3 + b2*x^2 + b1*x + b0

/* direct computation of product: */
b4*a4*x^8 + (b4*a3 + a4*b3)*x^7 + (b4*a2 + (a4*b2 + b3*a3))*x^6 + [...]

/* Karatsuba computation of product: */
b4*a4*x^8 + (b4*a3 + a4*b3)*x^7 + (b4*a2 + (a4*b2 + b3*a3))*x^6 + [...]

/* difference: */
0

OK.  /* we are fine! */
```

The number of multiplications with algorithm KA-n is $(n^2+n)/2$ which is suboptimal except for $n = 2$. However, recursive application can be worthwhile. One should start with the biggest prime factors as the number of additions is then minimized. The number of multiplications does not depend on the order of recursion (see [106] which also tabulates the number of additions and multiplications for $n \leq 128$).

With n just below a highly composite number one should add (zero-valued) 'dummy' terms and recursively use KA-n algorithms for small n. For example, with polynomials of degree 63 recursion with KA-2 (and $n = 64$) will beat the scheme "KA-7, then KA-3".

One could write code generators that create expanded versions of the recursions for n the product of small primes. For example, the expanded code for "KA-2, then KA-2" will be more efficient than KA-4. When the cost of multiplication is much higher than for addition (as for binary polynomial multiplication on general purpose CPUs) substantial savings can be expected.

Binary polynomials of very high degree

For polynomials of very high degree FFT-based algorithms can be used. The conceptionally most simple method is to use integer multiplication without the carry phase (which is polynomial multiplication!). Our multiplication example can be recomputed using decimal digits. The carry phase of the integer multiplication is replaced by a reduction modulo 2:

```
 100110111 * 110101
== 11022223331211  // integer multiplication
== 11000001111011  // parity of digits
```

The scheme will work for polynomials of degree less than nine only. When using an FFT multiplication scheme (see section 14.3 on page 391) we can multiply polynomials up to degree N as long as the integer values 0, 1, 2 ... $N + 1$ can be distinguished after computing the FFT. This is hardly a limitation at all: With the C-type `float` (24 bit mantissa) polynomials up to degree one million can be multiplied

assuming at least 20 bits are correct after the FFT. With type `double` (53-bit mantissa) there is no practical limit.

22.3 Arithmetic modulo a polynomial

Here we consider arithmetic of binary polynomials modulo a binary polynomial. Addition and subtraction are again the XOR operation. No modular reduction is required.

The functions shown in this section are given in [FXT: file bpol/bitpolmodarith.h].

Multiplication and squaring

Multiplication of a polynomial A by x modulo (a polynomial) C is achieved by shifting left and subtracting C if the coefficient shifted out is one:

```
static inline ulong bitpolmod_times_x(ulong a, ulong c, ulong h)
// Return  (A * x) mod C
// where A and C represent polynomials over Z/2Z:
//  W = pol(w) =: \sum_k{ [bit_k(w)] * x^k}
//
// h needs to be a mask with one bit set:
//  h == highest_bit(c) >> 1  == 1UL << (degree(C)-1)
{
    ulong s = a & h;
    a <<= 1;
    if ( s )  a ^= c;
    return  a;
}
```

In order to avoid the repeated computation of the highest set bit we introduced the auxiliary variable h that has to be initialized as described in the comment. Section 1.6 on page 15 gives algorithms for the function `highest_bit()`. Note that h needs to be computed only if the degree of the modulus C changes, which is usually only once for a series of calculations. Using h allows to get the correct result even if the degree of C equals the number of bits in a word in which case C does not fit into a word.

Multiplication of two polynomials a and b modulo C can be achieved by adding a reduction step to the binary multiplication routine:

```
inline ulong bitpolmod_mult(ulong a, ulong b, ulong c, ulong h)
// Return  (A * B) mod C
{
    ulong t = 0;
    while ( b )
    {
        if ( b & 1 )  t ^= a;
        b >>= 1;

        ulong s = a & h;
        a <<= 1;
        if ( s )  a ^= c;
    }
    return  t;
}
```

Optimization of the squaring and multiplication routines

Squaring a can be achieved as the multiplication $a \cdot a$. If many squaring have to be done with a fixed modulus then a optimization using a precomputed table of the residues x^{2k} mod C shown in section 23.1 on page 605 can be useful. Squaring of the polynomial $\sum_{k=0}^{d} a_k\, x^k$ can be achieved via the computation the sum $\sum_{k=0}^{d} a_k\, x^{2k}$ modulo C. When using the function

```
static inline ulong bitpolmod_times_x2(ulong a, ulong c, ulong h)
// Return  (A * x * x) mod C
{
```

```
    { ulong s=a&h; a<<=1; if (s) a^=c; }
    { ulong s=a&h; a<<=1; if (s) a^=c; }
    return  a;
}
```

The squaring function, with a the 4-fold unrolled loop, is

```
static inline ulong bitpolmod_square(ulong a, ulong c, ulong h)
// Return A*A mod C
{
    ulong t = 0, s = 1;
    do
    {
        if (a&1) t^=s; a>>=1; s=bitpolmod_times_x2(s, c, h);
        if (a&1) t^=s; a>>=1; s=bitpolmod_times_x2(s, c, h);
        if (a&1) t^=s; a>>=1; s=bitpolmod_times_x2(s, c, h);
        if (a&1) t^=s; a>>=1; s=bitpolmod_times_x2(s, c, h);
    }
    while ( a );
    return t;
}
```

Whether the unrolled code is used can be specified via the line

```
#define  MULT_UNROLL  // define to unroll loops 4-fold
```

The optimization used for the multiplication routine is also unrolling as decried in section 22.1 on page 570:

```
static inline ulong bitpolmod_mult(ulong a, ulong b, ulong c, ulong h)
{
    ulong t = 0;
    do
    {
        { if(b&1) t^=a; b>>=1; ulong s=a&h; a<<=1; if(s) a^=c; }
        { if(b&1) t^=a; b>>=1; ulong s=a&h; a<<=1; if(s) a^=c; }
        { if(b&1) t^=a; b>>=1; ulong s=a&h; a<<=1; if(s) a^=c; }
        { if(b&1) t^=a; b>>=1; ulong s=a&h; a<<=1; if(s) a^=c; }
    }
    while ( b );
    return  t;
}
```

It turns out that squaring via multiplication is slightly faster than via the described sum computation.

Exponentiation

Now modular exponentiation is identical to `bitpol_power()` except for the call to the modulo-multiplication functions:

```
inline ulong bitpolmod_power(ulong a, ulong e, ulong c, ulong h)
// Return (A ** e)  mod C
{
    if ( 0==e )  return 1;
    ulong s = a;
    while ( 0==(e&1) )
    {
        s = bitpolmod_square(s, c, h);
        e >>= 1;
    }
    a = s;
    while ( 0!=(e>>=1) )
    {
        s = bitpolmod_square(s, c, h);
        if ( e & 1 )  a = bitpolmod_mult(a, s, c, h);
    }
    return  a;
}
```

Division by x

Division by x is possible if the modulus has a nonzero constant term (that is, $\gcd(C, x) = 1$):

```
static inline ulong bitpolmod_div_x(ulong a, ulong c, ulong h)
// Return  (A / x) mod C
// C must have nonzero constant term: (c&1)==1
{
    ulong s = a & 1;
    a >>= 1;
    if ( s )
    {
        a ^= (c>>1);
        a |= h;  // so it also works for  n == BITS_PER_LONG
    }
    return  a;
}
```

If we do not insist on correct results for the case that the degree of C equals the number of bits in a word, we could simply use the following two-liner:

```
    if ( a & 1 )  a ^= c;
    a >> 1;
```

The operation needs only about two cycles (rate greater than 1000 million per second).

Inverse and division

The modular inverse of a binary polynomial can be computed via the extended GCD as shown in section 22.1 on page 574. Modular division is obtained by multiplication with the inverse:

```
inline ulong bitpolmod_divide(ulong a, ulong b, ulong c, ulong h)
// Return a/b modulo c.
// Must have: gcd(b,c)==1
{
    ulong i = bitpolmod_inverse(b, c);
    a = bitpolmod_mult(a, i, c, h);
    return a;
}
```

When the modulus is an irreducible polynomial (see section 22.4) then the inverse can also be computed via powering:

```
inline ulong bitpolmod_inverse(ulong a, ulong c, ulong h)
// Return (A ** -1)  mod C
// Must have: C irreducible.
{
    ulong r1 = (h<<1) - 2;  // max order minus one
    ulong i =  bitpolmod_power(a, r1, c, h);
    return  i;
}
```

22.4 Irreducible and primitive polynomials

A polynomial is called *irreducible* if it has no non-trivial factors (trivial factors are the constant polynomial '1' and the polynomial itself). A polynomial that has a non-trivial factorization is called *reducible*. A polynomial with zero constant term is always reducible because it has the factor x, except if the polynomial equals x. A binary polynomial that is irreducible has at least one term of odd degree and an odd number number of terms in total.

The irreducible polynomials are the 'primes' among the polynomials.

Whether a given polynomial can be factorized does depend on its coefficient field: $p(x) = x^2 + 1$ does not factor as an 'ordinary' polynomial (coefficients in $\mathbb{R}$ or $\mathbb{Z}$) while the factorization over $\mathbb{C}$ is $(x^2 + 1) = (x + i)(x - i)$. As a binary polynomial, the factorization is $(x^2 + 1) = (x + 1)^2$.

Let C be irreducible. Then the sequence $p_k = x^k \bmod (C)$, $k = 1, 2, \ldots$ is periodic and the (smallest) period m of the sequence is the order of x modulo C. We call m the *period* of the polynomial C. For a binary polynomial of degree n the maximal period equals $2^n - 1$. If the period is maximal then all nonzero polynomials of degree $n - 1$ occur in the sequence p.

For the period m of C we have $x^m = 1 \bmod C$, so $x^m - 1 = 0 \bmod C$. That is, C divides $x^m - 1$ but no polynomial $x^k - 1$ with $k < m$.

A polynomial is called *primitive* if its period is maximal. Then the powers of x generate all nonzero binary polynomials of degree $n - 1$. The polynomial x is a generator ('primitive root') modulo C. Primitivity implies irreducibility, the converse is not true.

The situation is somewhat parallel to the operations modulo an integer:

- Among those integers m that are prime some have the primitive root 2: The sequence 2^k for $k = 1, 2, \ldots, m - 1$ contains all nonzero numbers modulo m (see chapter 13 on page 379).
- Among those polynomials C that are irreducible some are primitive: The sequence x^k for $k = 1, 2, \ldots, 2^n - 1$ contains all nonzero polynomials modulo C.

Working modulo prime m the inverse of a number a can be obtained as $a^{-1} = a^{m-2}$ ($m-1$ is the maximal order of an element in $\mathbb{Z}/m\mathbb{Z}$). With an irreducible polynomial C of degree n the inverse modulo C of a polynomial A can be obtained as $A^{-1} = A^{2^n-2}$. Using the routines introduced so far:

```
r1 = (h<<1)-2;  // = max_order - 1 = 2^degree(C) - 2
i = bitpolmod_power(a, r1, c, h);
// here:  1==bitpolmod_mult(a, i, c, h);
```

Note that there is another notion of the term 'primitive', that of a polynomial for which the greatest common divisor of all coefficients is one.

The *weight* of a binary polynomial is the sum of its coefficients.

22.4.1 Roots of primitive polynomials have maximal order

A different characterization of primitivity is a s follows. Suppose you want to do computations with linear combinations $A = \sum_{k=0}^{n-1} a_k \, \alpha^k$ (where $a_k \in GF(2)$) of the powers of an (unknown!) root α of an irreducible polynomial $C = x^n + \sum_{k=0}^{n-1} c_k \, x^k$. Represent the linear combinations as binary vectors. Addition is component-wise, that is, XOR.

When multiplying A with the root α we obtain a term α^n which we want to get rid of. But we have

$$\alpha^n \quad = \quad \sum_{k=0}^{n-1} c_k \, \alpha^k \tag{22.15}$$

as α is a root of the polynomial C. Therefore we can use exactly the same modular reduction as with polynomial computation modulo C. The same is true for the multiplication of two linear combinations (of the powers of the same root α).

We see that of a polynomial is primitive if and only if all of its roots have maximal order.

22.4.2 Testing for irreducibility

Irreducibility tests for binary polynomials use the fact that the polynomial $x^{2^k} + x$ has all irreducible polynomials of degree k as a factor. For example,

$$\begin{aligned} x^{2^5} + x &= x^{32} + x \\ &= x \cdot (x+1) \cdot \\ &\quad \cdot \left(x^5 + x^2 + 1\right) \cdot \left(x^5 + x^3 + 1\right) \cdot \left(x^5 + x^3 + x^2 + x + 1\right) \cdot \\ &\quad \cdot \left(x^5 + x^4 + x^2 + x + 1\right) \cdot \left(x^5 + x^4 + x^3 + x + 1\right) \cdot \left(x^5 + x^4 + x^3 + x^2 + 1\right) \end{aligned} \tag{22.16}$$

A bit more can be said about the polynomial $x^{2^k}+x$ used in the tests: Let

$$z = x^{2^d}+x \tag{22.17a}$$

$$s = 1+\sum_{k=0}^{d-1} x^{2^k} \tag{22.17b}$$

$$t = s-1 = z/s \tag{22.17c}$$

Then s has all degree-d irreducible polynomials as factors where the coefficient of x^{d-1} is one and the factorization of t consists of polynomials where that coefficient is zero. As an example consider the case $d=7$. We use the notation $[a,b,c,\ldots,h] := x^a+x^b+x^c+\ldots+x^h$:

```
z = [128,1]  = s * t

s = [64,32,16,8,4,2,1,0] =          t = [64,32,16,8,4,2,1] =
    [1,0] *                             [1] *
    [7,6,0] *                           [7,1,0] *
    [7,6,3,1,0] *                       [7,3,0] *
    [7,6,4,1,0] *                       [7,3,2,1,0] *
    [7,6,4,2,0] *                       [7,4,0] *
    [7,6,5,2,0] *                       [7,4,3,2,0] *
    [7,6,5,3,2,1,0] *                   [7,5,2,1,0] *
    [7,6,5,4,0] *                       [7,5,3,1,0] *
    [7,6,5,4,2,1,0] *                   [7,5,4,3,0] *
    [7,6,5,4,3,2,0]                     [7,5,4,3,2,1,0]
```

The Ben-Or test for irreducibility

In order to check a degree-d polynomial C one can compute $u_k = x^{2^k}$ (modulo C) for each $k<d$ by successive squarings and test whether $\gcd(C, u_k+x) \neq 1$ for all k. But as a factor of degree f implies another one of degree $d-f$ it suffices to do the first $\lfloor d/2 \rfloor$ of the tests. The implied algorithm is called the *Ben-Or irreducibility test*. A C++ implementation is given in [FXT: file bpol/bitpolirred.h]:

```
inline ulong bitpol_irreducible_q(ulong c, ulong h)
// Return zero if C is reducible
// else (i.e. C is irreducible) return value != 0
// C must not be zero.
// h needs to be a mask with one bit set:
//  h == highest_bit(c) >> 1  == 1UL << (degree(C)-1)
{
    if ( c<2 )  return 0; // constant polynomial is reducible
    if ( 0==(1&c) )  return (c==2 ? 1 : 0); // x is a factor

    // if ( 0==(c & 0xaa..a ) )  return 0; // at least one odd degree term
    // if ( 0==parity(c) )  return 0; // need odd number of nonzero coeff.
    // if ( 0!=bitpol_test_squarefree(c) )  return 0; // must be square free
    ulong d = c;
    ulong u = 2;  // =^= x
    while ( 0 != (d>>=2) )  // floor( degree/2 ) times
    {
        // Square r-times for coefficients of c in GF(2^r).
        // We have r==1
        u = bitpolmod_square(u, c, h);
        ulong upx = u ^ 2;  // =^= u+x
        ulong g = bitpol_gcd(upx, c);
        if ( 1!=g )  return  0;   // reducible
    }
    return  1;  // irreducible
}
```

Commented out at the beginning are a few tests to check for trivial necessary conditions for irreducibility. For the test `bitpol_test_squarefree` (for a square factor) see section 22.8.2 on page 601.

Rabin's test for irreducibility

Rabin's algorithm to check irreducibility (given in [85]) uses less GCD computations than the Ben-Or test. A binary polynomial C of degree d is irreducible only if

$$\gcd\left(x^{2^d} - x \bmod C,\, C\right) \quad = \quad 0 \tag{22.18}$$

and, for all prime divisors p_i of d,

$$\gcd\left(x^{2^{d/p_i}} - x \bmod C,\, C\right) \quad = \quad 1 \tag{22.19}$$

Thereby the number of GCD computations equals the number of prime divisors of d plus one.

When the prime divisors are processed in decreasing order the successive exponents are increasing and the power of x can be updated via squarings (the idea can be found in [44]). Thereby the total number of squarings equals d which is minimal. A pari/gp implementation:

```
polirred2_rabin(c)=
{
    local(d, f, np, p, e, X, m, g, ns);
    d = poldegree(c);
    c *= Mod(1,2);
    if ( c==’x,  return(1) );  \\  ’x’ is irreducible
    if ( 0==polcoeff(c,0),  return(0));  \\ reducible
    if ( d<1,  return(0) ); \\ ’0’ and ’1’ are not irreducible
    if ( 1!=gcd(c, deriv(c,’x)),  return(0));  \\ reducible

    f = factor(d);
    np = matsize(f)[1];  \\ number of prime divisors
    ns = 0;  \\ numbers of squarings so far
    m = Mod( Mod(1,2) * ’x, c);

    forstep (k=np, 1, -1,  \\ biggest prime factor first, smallest exponent first
        p = f[k,1];  \\ prime divisor
        e = d/p;
        m = m^(2^(e-ns));
        \\ here: m == Mod(’x,c)^(2^e);
        ns = e;
        g = gcd(component(m,2)-’x, c);
        if ( 1!=g, return(0) );  \\ reducible
    );

    m = m^(2^(d-ns));
    if ( 0!=m-’x,  return(0) );  \\ reducible

    return( 1 );  \\ irreducible
}
```

Rabin's test will be faster than the Ben-Or test is the polynomial is irreducible. When the polynomial is reducible and has small factors (as often the case with 'random polynomials') than the Ben-Or test will terminate faster. A comparison of the tests is made in [44].

Transformations that preserve irreducibility

When a polynomial is irreducible then the composition with $x+1$ is also irreducible. Similar, the reversed word corresponds to another irreducible polynomial. Reversion also preserves primitivity, composition with $x+1$ does not in general: The most simple example is the primitive polynomial $p(x) = x^4 + x^3 + 1$ where $p(x+1) = x^4 + x^3 + x^2 + x + 1$ has the order 5. The order of x modulo $p(x)$ equals the order of $x+1$ modulo $p(x+1)$.

Composition with $x+1$ can be achieved by:

```
inline ulong bitpol_compose_xp1(ulong c)
// Return C(x+1)
// self-inverse
{
    ulong z = 1;
```

```
    ulong r = 0;
    while ( c )
    {
        if ( c & 1 )  r ^= z;
        c >>= 1;
        z ^= (z<<1);
    }
    return  r;
}
```

A version that avoids any branches and finishes in time $\log_2(b)$ (where b = bits per word) is

```
inline ulong bitpol_compose_xp1(ulong c)
// Return C(x+1)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
    while ( s )
    {
        c ^= ( (c&m) >> s );
        s >>= 1;
        m ^= (m>>s);
    }
    return  c;
}
```

Which is exactly the `blue_code()` from section 1.19 on page 41.

Computation of the reciprocal polynomial:

```
inline ulong bitpol_recip(ulong c)
// Return x^deg(C) * C(1/x)  (the reciprocal polynomial)
{
    ulong t = 0;
    while ( c )
    {
        t <<= 1;
        t |= (c & 1);
        c >>= 1;
    }
    return  t;
}
```

Alternatively, one can use the bit-reversion routines given in section 1.14 on page 28.

In general the sequence of successive 'compose' and 'reverse' operations leads to 6 different polynomials:

```
C= [11, 10, 4, 3, 0]
   [11, 10, 4, 3, 0]  -- recip (C=bitpol_recip(C)) -->
   [11,  8, 7, 1, 0]   -- compose (C=bitpol_compose_xp1(C)) -->
   [11, 10, 9, 7, 6, 5, 4, 1, 0]  -- recip -->
   [11, 10, 7, 6, 5, 4, 2, 1, 0]  -- compose -->
   [11,  9, 7, 2, 0]  -- recip -->
   [11,  9, 4, 2, 0]  -- compose -->
   [11, 10, 4, 3, 0]  == initial value
```

A list of all binary irreducible polynomials of degrees $2 \le d \le 11$ is given in [FXT: file data/all-irredpoly.txt]. Sequence A058943 in [OEIS] lists the first few evaluated (over $\mathbb{Z}$) at $x = 10$:

```
  10, 11,
  111,
  1011, 1101,
  10011, 11001, 11111,
  100101, 101001, 101111, 110111, 111011, 111101,
  1000011, 1001001, 1010111, 1011011, 1100001, 1100111, 1101101, 1110011, 1110101,
  10000011, 10001001, 10001111, 10010001, ...
```

Read these as binary numbers and write them down in decimal to obtain sequence A014580 in [OEIS]:

```
  2, 3,
  7,
  11, 13,
  19, 25, 31,
  37, 41, 47, 55, 59, 61,
  67, 73, 87, 91, 97, 103, 109, 115, 117,
  131, 137, 143, 145, 157, 167, 171, 185, 191, 193, 203, 211, 213, 229, 239, 241, 247, 253,
  283, 285, 299, 301, 313, 319, 333, 351, 355, 357, 361, 369, 375, ...
```

22.4.3 Irreducible polynomials of special forms *

In this section several tables of irreducible polynomials of special forms are given. For the computation the built-in irreducibility routine `polisirreducible` of the pari/gp package (see [Pari]) was used.

Alternating polynomials

The 'alternating' polynomial $1+\sum_{k=0}^{d} x^{2\,k+1} = 1 + x + x^3 + x^5 \ldots + x^{2d+1}$ can be irreducible only if d is odd:

```
d:   (irred. poly.)
1:    x^3 + x + 1
3:    x^7 + x^5 + x^3 + x + 1
5:    x^11 + x^9 + x^7 + x^5 + x^3 + x + 1
```

The list up to $d = 1000$ (sequence A107220 of [OEIS])

```
1, 3, 5, 7, 9, 13, 23, 27, 31, 37, 63, 69, 117, 119, 173, 219, 223,
247, 307, 363, 383, 495, 695, 987,
```

can be obtained (within about 10 minutes) via

```
for(d=1,1000, p=(1+sum(t=0,d,x^(2*t+1))); if(polisirreducible(Mod(1,2)*p),print1(d,", ")))
```

Trinomials of the form $1 + \mathbf{x^d} + \mathbf{x^{k\,d}}$

k=2: The polynomial $p = 1 + x^d + x^{2d}$ is irreducible whenever d is a power of three:

```
  1:  x^2 +  x +  1
  3:  x^6 +  x^3 +  1
  9:  x^18 +  x^9 +  1
 27:  x^54 +  x^27 +  1
 81:  x^162 +  x^81 +  1
243:  x^486 +  x^243 +  1
 ...
```

k=3: Similarly, $p = 1 + x^d + x^{3d}$ is irreducible whenever d is a power of seven:

```
  1:  x^3 +  x +  1
  7:  x^21 +  x^7 +  1
 49:  x^147 +  x^49 +  1
343:  x^1029 +  x^343 +  1
```

k=4: The test for the irreducibility of $p = 1 + x^d + x^{4d}$ can be done via

```
for(d=1,N,(p=(1+x^d+x^(4*d))); if(polisirreducible(Mod(1,2)*p),print(d,":  ",p)))
```

One gets

```
  1:  x^4 + x + 1
  3:  x^12 + x^3 + 1
  5:  x^20 + x^5 + 1
  9:  x^36 + x^9 + 1
 15:  x^60 + x^15 + 1
 25:  x^100 + x^25 + 1
 27:  x^108 + x^27 + 1
 45:  x^180 + x^45 + 1
 75:  x^300 + x^75 + 1
 81:  x^324 + x^81 + 1
125:  x^500 + x^125 + 1
135:  x^540 + x^135 + 1
225:  x^900 + x^225 + 1
243:  x^972 + x^243 + 1
375:  x^1500 + x^375 + 1
405:  x^1620 + x^405 + 1
 ...
```

The list of reversed polynomials is

```
  1:  x^4 + x^3 + 1
  3:  x^12 + x^9 + 1
  5:  x^20 + x^15 + 1
  9:  x^36 + x^27 + 1
 15:  x^60 + x^45 + 1
 25:  x^100 + x^75 + 1
 27:  x^108 + x^81 + 1
 45:  x^180 + x^135 + 1
 75:  x^300 + x^225 + 1
 81:  x^324 + x^243 + 1
125:  x^500 + x^375 + 1
135:  x^540 + x^405 + 1
225:  x^900 + x^675 + 1
243:  x^972 + x^729 + 1
 ...
```

Observe that d always factors as $d = 3^i\,5^j$, $i, j \in \mathbb{N}$, thereby the list can be reproduced and enhanced using

```
fordiv((3*5)^7,i,s=(i);print1(" ",4*s))
```

```
  4, 12, 20, 36, 60, 100, 108, 180, 300, 324, 500, 540, 900, 972,
  1500, 1620, 2500, 2700, 2916, 4500, 4860, 7500, 8100, 8748, 12500, ...
```

Similar regularities can be observed for related forms, see [23].

Trinomials of the form $1 + \mathbf{x^k} + \mathbf{x^d}$

k=1: The polynomials of the form $1 + x + x^d$ are irreducible for the following $2 \le d \le 34353$:

```
    2, 3, 4, 6, 7, 9, 15, 22, 28, 30, 46, 60, 63,
    127, 153, 172, 303, 471, 532, 865, 900,
    1366, 2380, 3310, 4495, 6321, 7447,
    10198, 11425, 21846, 24369, 27286, 28713, 32767, 34353
```

This is sequence A002475 of [OEIS].

k=2: $1 + x^2 + x^d$ is irreducible for the following $3 \le d \le 57341$ (sequence A057460):

```
    3, 5, 11, 21, 29, 35, 93, 123, 333, 845, 4125,
    10437, 10469, 14211, 20307, 34115, 47283, 50621, 57341
```

k=3: $p = 1 + x^3 + x^d$ is irreducible for the following $4 \le d \le 1000$ (sequence A057461):

```
    4, 5, 6, 7, 10, 12, 17, 18, 20, 25, 28, 31, 41, 52, 66,
    130, 151, 180, 196, 503, 650, 761, 986
```

k=4: $p = 1 + x^4 + x^d$ is irreducible for the following $5 \le d \le 1000$ (sequence A057463):

```
    7, 9, 15, 39, 57, 81, 105
```

k=5: $p = 1 + x^5 + x^d$ is irreducible for only few $6 \le d \le 1000$ (sequence A057474):

```
    6, 9, 12, 14, 17, 20, 23, 44, 47, 63, 84,
    129, 236, 278, 279, 297, 300, 647, 726, 737,
```

Low-block and full polynomials

Irreducible polynomials of the form $x^d + \sum_{k=0}^{q} x^q$ where $q < d$. Note that the reversed polynomials are also irreducible. The list [FXT: file data/all-lowblock-irredpoly.txt] contains all those polynomials for $d \le 400$. A subset of the list are those polynomials where $q = d - 1$, that is, all coefficients are one: The 'full' polynomial $p = \sum_{k=0}^{s-1} x^k = 1 + x + x^2 + \ldots + x^{s-1}$ can be irreducible only if $s = d + 1$ is prime. The first examples of irreducible polynomials of this form are

```
  s:   (irred. poly.)
  2:   x + 1
  3:   x^2 + x + 1
  5:   x^4 + x^3 + x^2 + x + 1
 11:   x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
```

Note that, with the exception of $x^2 + x + 1$, none of them is primitive. In fact the order of x equals $d + 1$, which is immediate when printing the powers of x (example using $s = 5$):

```
    k     x^k
    0     ...1
    1     ..1.
    2     .1..
    3     1...
    4     1111
    5     ...1  == x^5 == 1
```

The full polynomials $x^{s-1} + \ldots + 1$ are irreducible for the following $s < 400$:

```
    2, 3, 5, 11, 13, 19, 29, 37, 53, 59, 61, 67, 83, 101, 107, 131, 139, 149,
    163, 173, 179, 181, 197, 211, 227, 269, 293, 317, 347, 349, 373, 379, 389, ...
```

It can be generated with the pari/gp one-liner:

```
1111111111111111111111111111 == c
11.........................          deg= 1  |   2^1 ==  2
1..1.......................          deg= 3  |   2^2 ==  4
1......1...................          deg= 7  |   2^3 ==  8
1..............1...........          deg=15  |   2^4 == 16
1.1........................          deg= 2  |   2^5 ==  3
1....1.....................          deg= 5  |   2^6 ==  6
1..........1...............          deg=11  |   2^7 == 12
1......................1.....          deg=23  |   2^8 == 24
1.................1.........          deg=18  |   2^9 == 19
1.......1..................          deg= 8  |  2^10 ==  9
1.................1.........          deg=17  |  2^11 == 18
1.....1....................          deg= 6  |  2^12 ==  7
1...........1..............          deg=13  |  2^13 == 14
1.........................1.          deg=27  |  2^14 == 28
[end Ben-Or test]
1.........................1..          deg=26  |  2^15 == 27
1........................1....          deg=24  |  2^16 == 25
1...................1........          deg=20  |  2^17 == 21
1...........1..............          deg=12  |  2^18 == 13
1........................1...          deg=25  |  2^19 == 26
1......................1.....          deg=22  |  2^20 == 23
1..............1...........          deg=16  |  2^21 == 17
1...1......................          deg= 4  |  2^22 ==  5
1........1.................          deg= 9  |  2^23 == 10
1..................1........          deg=19  |  2^24 == 20
1........1.................          deg=10  |  2^25 == 11
1...................1......          deg=21  |  2^26 == 22
1............1.............          deg=14  |  2^27 == 15
..........................          deg= 0  |  2^28 ==  1
```

Figure 22.2: Polynomials $b_k := x^{2^k-1} + 1 \bmod c$ where $c = \sum_{j=0}^{s-1} x^j$ and $s = 29$. The right columns give the degree of b_k and the value of $2^k \bmod s$.

```
11111111111111111111 == c
11....................          deg= 1  |   2^1 ==  2
1..1..................          deg= 3  |   2^2 ==  4
1......1..............          deg= 7  |   2^3 ==  8
1..............1......          deg=15  |   2^4 == 16
1.......1.............          deg= 8  |   2^5 ==  9
1................1.....          deg=17  |   2^6 == 18
1...........1.........          deg=12  |   2^7 == 13
1.1...................          deg= 2  |   2^8 ==  3
1....1................          deg= 5  |   2^9 ==  6
1..........1..........          deg=11  |  2^10 == 12
......................          deg= 0  |  2^11 ==  1
[end Ben-Or test]
11....................          deg= 1  |  2^12 ==  2
1..1..................          deg= 3  |  2^13 ==  4
1......1..............          deg= 7  |  2^14 ==  8
[-snip-]
```

Figure 22.3: Polynomials $b_k := x^{2^k-1} + 1 \bmod c$ where $c = \sum_{j=0}^{s-1} x^j$ and $s = 23$. The right columns give the degree of b_k and the value of $2^k \bmod s$.

```
forprime(s=2,400,p=sum(q=0,s-1,x^q);if(polisirreducible(Mod(1,2)*p),print1(", ",s)))
```

Compare this list to the list of primes with primitive root 2 (figure 6.6 on page 225). The lists are identical for $s > 2$. In fact, the number of factors f of the full polynomial of degree $d = s - 1$ with coefficients in $GF(p)$ for prime s can be computed as follows:

$$f = \frac{s-1}{\mathrm{ord}_0(p)} \quad \text{where} \tag{22.20}$$

$$\mathrm{ord}_0(p) := \begin{cases} \text{the order of} \quad (p \bmod s) & \text{for } p \neq s \\ 1 & \text{for } p = s \end{cases}$$

The relation implies that $1 + x + \ldots + x^{s-1}$ over $GF(p)$ is irreducible if and only if p is a primitive root modulo s. Further all factors are of the same degree $(s-1)/f = \mathrm{ord}_0(p)$.

It is instructive to look at a slightly modified version of the Ben-Or test. Instead of using $x^{2^k} + x$ we use $x^{2^k-1} + 1$ in step k. For $s = 29$ we get the polynomials shown in figure 22.2 on the facing page. The polynomials $x^{2^k-1} + 1 \bmod p$ have exactly two nonzero coefficients, the constant coefficient and the highest coefficient at position $2^k - 1 \bmod s$. If 2 is not a primitive root modulo s then the zero polynomial occurs for $k \leq (s-1)/2$. For $s = 23$ this happens at step 11, as shown in figure 22.3 on the preceding page.

22.4.4 Testing for primitivity

Checking a degree-d binary polynomial for primitivity by directly using the definition costs proportional 2^n operations which is prohibitive except for tiny d.

Improvements come from the multiplicative structure of the problem: It suffices to check whether $x^k \equiv 1 \bmod C$ for all divisors of the maximal order that are greater than n. However, this soon gets impractical quickly as n grows. Already with $n = 144$ one has maximal order $m = 2^{144} - 1$ (which has 262144 divisors of which 262112 have to be tested) so the computation gets expensive. Note that the implied algorithm needs the factorization of $m = 2^n - 1$.

A much better solution is a modification of the algorithm to determine the oder in a finite field given on page 533. The implementation given here uses the pari/gp language:

```
polorder(p) =
/* Order of x modulo p (p irreducible over GF(2)) */
{
  local(g, g1, te, tp, tf, tx);
  g = 'x;
  p *= Mod(1,2);
  te = nn_;
  for(i=1, np_,
    tf = vf_[i];  tp = vp_[i];  tx = vx_[i];
    te = te / tf;
    g1 = Mod(g, p)^te;
    while ( 1!=g1,
      g1 = g1^tp;
      te = te * tp;
    );
  );
  return( te );
}
```

The function uses the following global variables that must be set up before call:

```
nn_ = 0;  /* max order  =  2^n-1 */
np_ = 0;  /* number of primes in factorization */
vp_ = []; /* vector of primes */
vf_ = []; /* vector of factors (prime powers) */
vx_ = []; /* vector of exponents */
```

As given, the algorithm will do n_p exponentiations modulo p where n_p is the number of different primes in the factorization in m. For $n = 144$ one has $n_p = 17$.

A shortcut that makes the algorithm terminate as soon as the computed order drops below maximum is

```
polmaxorder_q(p) =
/* Whether order of x modulo p is maximal  (p irreducible over GF(2)) */
/* Early-out variant */
{
  local(g, g1, te, tp, tf, tx, ct);
  g = 'x;
  p *= Mod(1,2);
  te = nn_;
  for(i=1, np_,
    tf = vf_[i];  tp = vp_[i];  tx = vx_[i];
    te = te / tf;
    g1 = Mod(g, p)^te;
    ct = 0;
    while ( 1!=g1,
      g1 = g1^tp;
      te = te * tp;
      ct = ct + 1;
    );
    if ( ct<tx,  return(0) );
  );
  return(1);
}
```

Range	time	# irred. tests	tests/sec
90 .. 99:	<1 sec	258	>300
190 .. 199:	6 sec	588	100
290 .. 299:	10 sec	474	58
390 .. 399:	30 sec	810	27

Determination of irreducible polynomials.

Range	time	# irred. tests	# prim. tests
90 .. 99:	4 sec	313	13
190 .. 199:	40 sec	1371	25
290 .. 299:	90 sec	864	14
390 .. 399:	260 sec	1529	14

Determination of primitive polynomials.

Figure 22.4: Timings and number of tests involved in the determination of 10 low-bit irreducible (top) and primitive (bottom) binary polynomials in selected ranges of their degrees. The small number of tests for primitivity indicates that an irreducible polynomial is likely also primitive.

Using `polmaxorder_q()` and pari's built-in `polisirreducible()` the search for the lowest-bit primitive polynomials up to degree $n = 100$ is a matter of about 10 seconds. Extending the list up to $n = 200$ takes 3 minutes. The computation of all polynomials up to degree $n = 400$ takes less than on hour. Figure 22.4 shows approximate timings with the determination of irreducible and primitive polynomials.

Again, the algorithm depends an precomputed factorizations. The table [FXT: file data/mersenne-factorizations-400.txt] taken from [Fact] was used in order to save computation time.

For prime $m = 2^n - 1$ (that is, m is a Mersenne prime) irreducibility suffices for primality: The one-liner

```
x=127; for(z=1,x-1,if(polisirreducible(Mod(1,2)+t^z+t^x),print1(" ",z)))
```

finds all primitive trinomials for exponents of Mersenne primes in no time. For example, within 70 seconds one obtains:

```
  89:  38 51
 127:  1 7 15 30 63 64 97 112 120 126
 521:  32 48 158 168 353 363 473 489
 607:  105 147 273 334 460 502
```

22.4.5 Primitive polynomials of special forms *

In this section several tables of primitive polynomials of special forms are given. For the computation the pari/gp routines given so far were used. The abbreviation 'PP' is used for 'primitive polynomial' in what follows.

Primitive minimum-weight polynomials

The data in [FXT: file data/minweight-primpoly.txt] lists minimal-weight PPs where in addition the coefficients are as close to the low end as possible. The first entries are (column-wise formatting for readability):

```
2,1,0           17,3,0          32,7,6,2,0      48,9,7,4,0
3,1,0           18,7,0          33,13,0         49,9,0
4,1,0           19,5,2,1,0      34,8,4,3,0      50,4,3,2,0
5,2,0           20,3,0          35,2,0          51,6,3,1,0
6,1,0           21,2,0          36,11,0         52,3,0
7,1,0           22,1,0          37,6,4,1,0      53,6,2,1,0
8,4,3,2,0       23,5,0          39,4,0          54,8,6,3,0
9,4,0           24,4,3,1,0      40,5,4,3,0      55,24,0
10,3,0          25,3,0          41,3,0          56,7,4,2,0
11,2,0          26,6,2,1,0      42,7,4,3,0      57,7,0
12,6,4,1,0      27,5,2,1,0      43,6,4,3,0      58,19,0
13,4,3,1,0      28,3,0          44,6,5,2,0      59,7,4,2,0
14,5,3,1,0      29,2,0          45,4,3,1,0      60,1,0
15,1,0          30,6,4,1,0      46,8,7,6,0      61,5,2,1,0
16,5,3,2,0      31,3,0          47,5,0          62,6,5,3,0
```

A list of minimal-weight PPs that fit into a machine word is given in [FXT: file bpol/minweightprimpoly.cc].

Primitive low-bit polynomials

Choosing those PPs where the highest nonzero coefficient is as low as possible one gets the list in [FXT: file data/lowbit-primpoly.txt]. It starts as

```
2,1,0           10,3,0          18,5,2,1,0      26,6,2,1,0
3,1,0           11,2,0          19,5,2,1,0      27,5,2,1,0
4,1,0           12,6,4,1,0      20,3,0          28,3,0
5,2,0           13,4,3,1,0      21,2,0          29,2,0
6,1,0           14,5,3,1,0      22,1,0          30,6,4,1,0
7,1,0           15,1,0          23,5,0          31,3,0
8,4,3,2,0       16,5,3,2,0      24,4,3,1,0      32,7,5,3,2,1,0
9,4,0           17,3,0          25,3,0          33,6,4,1,0
```

The corresponding extract for small degrees is given in [FXT: file bpol/lowbitprimpoly.cc]. The index (position) of the highest highest nonzero coefficient grows slowly with n. For $n \leq 400$ the entries where that index is bigger than in all prior lines are:

```
2,1,0
5,2,0
8,4,3,2,0
12,6,4,1,0
32,7,5,3,2,1,0
46,8,5,3,2,1,0
104,9,8,6,5,4,3,2,0
108,10,9,7,6,5,4,3,2,1,0
360,12,9,8,6,4,3,1,0
```

Thereby one can store the list in compact binary format even if it extends to high degrees.

All primitive low-bit polynomials for certain degrees

A list of all PPs for degree $n = 256$ with the second-highest order ≤ 15 is given in [FXT: file data/lowbit256-primpoly.txt]. It starts as

```
256,10,5,2,0
256,10,8,5,4,1,0
256,10,9,8,7,4,2,1,0
256,11,8,4,3,2,0
256,11,8,6,4,3,0
256,11,10,9,4,2,0
256,11,10,9,7,4,0
256,12,7,5,4,2,0
256,12,8,7,6,3,0
```

Similar tables for degrees 127, 128 and 521 can be found under the obvious names.

All primitive polynomials for low degrees

```
2,1,0

3,1,0
3,2,0

4,1,0
4,3,0

5,2,0
5,3,0
5,3,2,1,0
5,4,2,1,0
5,4,3,1,0
5,4,3,2,0

6,1,0
6,4,3,1,0
6,5,0
6,5,2,1,0
6,5,3,2,0
6,5,4,1,0

7,1,0
7,3,0
7,3,2,1,0
7,4,0
7,4,3,2,0
7,5,2,1,0
7,5,3,1,0
7,5,4,3,0
7,5,4,3,2,1,0
7,6,0
7,6,3,1,0
7,6,4,1,0
7,6,4,2,0
7,6,5,2,0
7,6,5,3,2,1,0
7,6,5,4,0
7,6,5,4,2,1,0
7,6,5,4,3,2,0

8,4,3,2,0
8,5,3,1,0
8,5,3,2,0
8,6,3,2,0
8,6,4,3,2,1,0
8,6,5,1,0
8,6,5,2,0
8,6,5,3,0
8,6,5,4,0
8,7,2,1,0
8,7,3,2,0
8,7,5,3,0
8,7,6,1,0
8,7,6,3,2,1,0
8,7,6,5,2,1,0
8,7,6,5,4,2,0
```

Figure 22.5: All binary primitive polynomials up to degree 8.

For degrees $n \leq 8$ the complete list of PPs is shown in figure 22.5. The file [FXT: file data/all-primpoly.txt] contains a list which extends to degree 11. The total number of PPs of degree n is $P_n = \varphi(2^n - 1)/n$, see section 22.5 on the facing page.

Primitive trinomials

For many degrees there exist PPs with just three nonzero coefficients, these are called *trinomials*.

A list all primitive trinomials is given in [FXT: file data/all-trinomial-primpoly.txt]. The lists starts like (column-wise for readability):

```
2,1,0    5,2,0    7,1,0    10,3,0    15,1,0    17,3,0    18,7,0    22,1,0
3,1,0    5,3,0    7,3,0    10,7,0    15,4,0    17,5,0    18,11,0   22,21,0
3,2,0    6,1,0    7,4,0    11,2,0    15,7,0    17,6,0    20,3,0    23,5,0
4,1,0    6,5,0    7,6,0    11,9,0    15,8,0    17,11,0   20,17,0   23,9,0
4,3,0             9,4,0              15,11,0   17,12,0   21,2,0    23,14,0
                  9,5,0              15,14,0   17,14,0   21,19,0   23,18,0
```

A more compact form of the list can is given in [FXT: file data/all-trinomial-primpoly-short.txt]:

```
 2:  1
 3:  1 2
 4:  1 3
 5:  2 3
 6:  1 5
 7:  1 3 4 6
 9:  4 5
10:  3 7
11:  2 9
15:  1 4 7 8 11 14
```

A line starts with the entry for the degree followed by all possible positions of the middle coefficient.

Primitive trinomials of the most simple form $x^n + x + 1$ exist for $n \leq 400$ and $n \in \{2, 3, 4, 6, 7, 15, 22, 60, 63, 127, 153\}$. In fact these numbers are the sequence A073639 of [OEIS], where one finds in addition 471, 532, 865, 900, 1366 with the next candidate being 4495.

Primitive pentanomials

PPs with exactly five nonzero coefficients are given in [FXT: file data/pentanomial-primpoly.txt]. No primitive pentanomial exists for degrees $n < 5$ but for all higher degrees one seems to exist but this has not been proven so far. Entries of the special form $x^n+x^3+x^2+x+1$ are there for $n \in \{5, 7, 17, 25, 31, 41, 151\}$.

Primitive low-block polynomials

A low-block polynomial has the special form $x^n + \sum_{j=0}^{k} x^j$. Such PPs exist for 218 degrees $n \le 400$. These are especially easy to store in an array (saving the index of the second highest nonzero coefficient in array element n). Moreover, simplified and possible more efficient routines based on the special forms might be possible. A complete list of all low-block PPs with degree $n \le 400$ is given in [FXT: file data/all-lowblock-primpoly.txt]. The list starts like:

```
2,1
3,1
4,1
5,3
6,1
7,1
7,3  // == x^7 + x^3 + x^2 + x + 1
7,5
10,7
```

Among the low-block PPs are a few where just one bit (the coefficient after the leading coefficient) is not set. For $n \le 400$ this is for the following degrees:

```
3, 5, 7, 13, 15, 23, 37, 47, 85, 127, 183, 365, 383
```

The PPs listed in [FXT: file data/lowblock-primpoly.txt] have the smallest possible block of set bits.

Random primitive polynomials

A list PPs that were generated by choosing from random polynomials those that were primitive is given in [FXT: file data/rand-primpoly.txt].

```
2,1,0
3,2,0
4,3,0
5,3,2,1,0
6,5,0
7,6,3,1,0
8,6,5,3,0
9,5,3,2,0
10,8,4,3,0
11,10,8,7,6,4,2,1,0
12,11,10,9,4,3,0
[--snip--]
31,29,23,21,19,17,15,12,10,8,7,5,4,2,0
32,31,29,28,26,24,23,18,10,9,8,7,6,5,4,1,0
33,31,29,28,27,25,20,19,17,13,11,10,8,6,5,1,0
34,33,31,27,25,24,22,18,16,15,14,12,11,10,9,8,7,6,4,2,0
[--snip--]
43,40,39,36,33,30,29,27,26,22,21,17,16,15,13,12,11,9,8,5,3,1,0
```

Random PP tend to have a weight that is approximately equal to half of the degree.

22.5 The number of irreducible and primitive polynomials

The number of irreducible binary polynomials of degree n is

$$I_n \;=\; \frac{1}{n}\sum_{d\backslash n}\mu(d)\,2^{n/d} \;=\; \frac{1}{n}\sum_{d\backslash n}\mu(n/d)\,2^{d} \tag{22.21}$$

The function $\mu(.)$ is defined by relation 22.30 on page 597. The expression is identical to the formula for the number of Lyndon words (relation 5.20 on page 214). If n is prime then $I_n = \frac{2^n-2}{n}$. For $n \leq 40$ we get (sequence A001037 of [OEIS]):

n	I_n	n	I_n	n	I_n	n	I_n
1	2	11	186	21	99858	31	69273666
2	1	12	335	22	190557	32	134215680
3	2	13	630	23	364722	33	260300986
4	3	14	1161	24	698870	34	505286415
5	6	15	2182	25	1342176	35	981706806
6	9	16	4080	26	2580795	36	1908866960
7	18	17	7710	27	4971008	37	3714566310
8	30	18	14532	28	9586395	38	7233615333
9	56	19	27594	29	18512790	39	14096302710
10	99	20	52377	30	35790267	40	27487764474

The list of all irreducible polynomials up to degree 11 is given in [FXT: file data/all-irredpoly.txt].

The number of primitive binary polynomials of degree n is

$$P_n \quad = \quad \frac{\varphi(2^n-1)}{n} \tag{22.22}$$

If n is the exponent of a Mersenne prime we have $P_n = \frac{2^n-2}{n}$. The values of P_n for $n \leq 40$ are:

n	P_n	n	P_n	n	P_n	n	P_n
1	1	11	176	21	84672	31	69273666
2	1	12	144	22	120032	32	67108864
3	2	13	630	23	356960	33	211016256
4	2	14	756	24	276480	34	336849900
5	6	15	1800	25	1296000	35	929275200
6	6	16	2048	26	1719900	36	725594112
7	18	17	7710	27	4202496	37	3697909056
8	16	18	7776	28	4741632	38	4822382628
9	48	19	27594	29	18407808	39	11928047040
10	60	20	24000	30	17820000	40	11842560000

The difference $D_n := I_n - P_n$ is the number of irreducible non-primitive polynomials:

n	D_n	n	D_n	n	D_n	n	D_n
1	1	11	10	21	15186	31	0
2	0	12	191	22	70525	32	67106816
3	0	13	0	23	7762	33	49284730
4	1	14	405	24	422390	34	168436515
5	0	15	382	25	46176	35	52431606
6	3	16	2032	26	860895	36	1183272848
7	0	17	0	27	768512	37	16657254
8	14	18	6756	28	4844763	38	2411232705
9	8	19	0	29	104982	39	2168255670
10	39	20	28377	30	17970267	40	15645204474

D_n equals zero whenever n is the exponent of a Mersenne prime. The complete list of these polynomials up to degree 12 inclusive is given in [FXT: file data/all-nonprim-irredpoly.txt].

22.6 Generating irreducible polynomials from necklaces

That the number of length-n Lyndon words (see section 5.22.2 on page 214) is equal to the number of degree-n irreducible polynomials is not a coincidence. Indeed, [35] gives an algorithm that, given a

primitive polynomial, generates an irreducible polynomial from a Lyndon word: Let b be a Lyndon word, c be a primitive polynomial of degree n and a an element of maximal order modulo c. Set $e = a^b$ and compute $p(x)$, a polynomial over $GF(2^n)$, as:

$$p(x) \quad := \quad \left(x - e\right)\left(x - e^2\right)\left(x - e^4\right)\left(x - e^8\right)\cdots\left(x - e^{2^{n-1}}\right) \tag{22.23}$$

Then all coefficients of $p(x)$ are either zero or one and the polynomial is irreducible over $GF(2)$.

An implementation in C++ is given in [FXT: `class necklace2bitpol` in bpol/necklace2bitpol.h]:

```
class necklace2bitpol
{
public:
    ulong p_[BITS_PER_LONG+1];  // polynomial over GF(2**n_)
    ulong n_;  // degree of c_
    ulong c_;  // modulus (irreducible polynomial)
    ulong h_;  // mask used for computation
    ulong a_;  // generator modulo c
    ulong e_;  // a^b
public:
    necklace2bitpol(ulong n, ulong c=0, ulong a=0)
        : n_(n), c_(c), a_(a)
    {
        if ( 0==c )  c_ = lowbit_primpoly[n];
        if ( 0==a )  a_ = 2UL;  // 'x'
        h_ = (highest_bit(c_) >> 1);
    }

    ~necklace2bitpol()  { ; }

    ulong poly(ulong b)
    {
        for (ulong k=0; k<=n_; ++k)  p_[k] = 0;
        p_[0] = 1;
        ulong e = bitpolmod_power(a_, b, c_, h_);
        e_ = e;  // for reference

        for (ulong d=1; d<=n_; ++d)
        {
            for (ulong k=d; k; --k)  p_[k] = p_[k-1];
            p_[0] = 0;
            for (ulong k=0; k<d; ++k)
            {
                p_[k] ^= bitpolmod_mult( p_[k+1], e, c_, h_);
            }

            e = bitpolmod_square(e, c_, h_);
        }

        ulong p2 = 0;
        for (ulong j=0; j<=n_; ++j)  p2 |= (p_[j] << j);
        return p2;
    }
};
```

Here is a list of all polynomials that are generated with $c = x^4 + x^3 + 1$, $a = x$ and not restricting b in any way:

```
      b             e         p
    ...1 4 m    ..1.    .11..1 P
    ..1. 4      .1..    .11..1 P
    ..1i 4 m    .1..    .1111
    .1.. 4      .1..1   .11..1 P
    .1.1 2 m    .1.11   .1.1.1 reducible
    .11. 4      .1111   .11111
    .111 4 m    ..111   .1..11 P
    1... 4      .111.   .11..1 P
    1..1 4      ..1.1   .11111
    1.1. 2      .1.1.   .1.1.1 reducible
    1.11 4      .111.1  .1..11 P
    11.. 4      ...11   .11111
    11.1 4      ..11.   .1..11 P
    111. 4      .11..   .1..11 P
    1111 1 m    ....1   .1...1 reducible
```

This is the output of [FXT: demo gf2n/necklace2irred-demo.cc]. The columns are: b and its cyclic period (an '`m`' appended if the word is the cyclic minimum), e and p where a '`P`' indicates that p is primitive. Observe that cyclic shifts of the same word give identical polynomials p. Further, if the period is not maximal then p is reducible. Restricting our attention to the necklaces b:

```
     b            e         p
....1 4 m    ...1.     .11..1 P
...11 4 m    .1...     .11111
..1.1 2 m    .1.11     .1.1.1 reducible
..111 4 m    ..111     .1...11 P
.1111 1 m    ....1     .1...1 reducible
```

The Lyndon words b give all degree-n irreducible polynomials. The primitive polynomials are exactly those where $\gcd(b,\, 2^n - 1) = 1$.

To generate all irreducible binary polynomials of fixed degree use [FXT: `class all_irredpol` in bpol/allirredpol.h]. The usage is demonstrated with [FXT: demo gf2n/all-irredpoly-demo.cc] which by default prints all irreducible polynomials and indicates by a '`P`' which are primitive:

```
modulus=  12 6 4 1 0
  1.....1.1..11  P
  1.1...1.11.11
  1.....1..11.1
  1.1...11.1.11
  1..111111.1.1
  1.1.1.1.1.1.11
  11.111.1...111  P
  1.11...111111  P
  111.1.1.11.11  P
  1.11.11111111
[--snip--]
  1.1.111111..1
  11.11.1.1.111  P
  1..1..11.11.1
  11..1.1....1  P
 n = 12:    # irreducible =335  # primitive =144
```

For $n = 24$ one gets $I_{24} = 698,870$ irreducible polynomials (among them are $P_{24} = 276,480$ primitive polynomials). The computation takes about 15 seconds.

22.7 Irreducible and cyclotomic polynomials *

The roots (over $\mathbb{C}$) of the polynomial $x^n - 1$ are the n-th roots of unity:

$$x^n - 1 \;=\; \prod_{k=0}^{n-1} \left(x - \exp\left(\frac{2\,i\,\pi\,k}{n} \right) \right) \tag{22.24}$$

The n-th *cyclotomic polynomial* Y_n can be defined as the polynomial whose roots are the primitive n-th roots of unity:

$$Y_n(x) \;:=\; \prod_{\substack{k=0..n-1 \\ \gcd(k,n)=1}} \left(x - \exp\left(\frac{2\,i\,\pi\,k}{n} \right) \right) \tag{22.25}$$

The degree of Y_n equals the number of primitive n-th roots, that is

$$\deg(Y_n) \;=\; \varphi(n) \tag{22.26}$$

The coefficients are integers, for example,

$$Y_{63}(x) \;=\; x^{36} - x^{33} + x^{27} - x^{24} + x^{18} - x^{12} + x^9 - x^3 + 1 \tag{22.27}$$

One has

$$x^n - 1 \;=\; \prod_{d \backslash n} Y_d(x) \tag{22.28}$$

and

$$Y_n(x) \;=\; \prod_{d\backslash n} \left(x^d - 1\right)^{\mu(n/d)} \tag{22.29}$$

where $\mu(.)$ is the *Möbius function*:

$$\mu(n) \;:=\; \begin{cases} 0 & \text{if } n \text{ has a square factor} \\ (-1)^k & \text{if } n \text{ is a product of } k \text{ distinct primes} \\ +1 & \text{if } n = 1 \end{cases} \tag{22.30}$$

The Möbius function is multiplicative

$$\mu(n)\;\mu(m) \;=\; \begin{cases} \mu(n\,m) & \text{if} \quad \gcd(n,m) = 1 \\ 0 & \text{else} \end{cases} \tag{22.31}$$

and satisfies

$$\sum_{d\backslash n} \mu(d) \;=\; \begin{cases} 1 & \text{if} \quad n = 0 \\ 0 & \text{else} \end{cases} \tag{22.32}$$

The pair of relations 22.28 and 22.29 is actually a special case of the (multiplicative) *Möbius inversion principle*:

$$g(n) \;=\; \prod_{d\backslash n} f(d) \quad\Longleftrightarrow\quad f(n) \;=\; \sum_{d\backslash n} g\,(d)^{\mu(n/d)} \tag{22.33}$$

Relation 22.28 implies (considering the polynomial degrees only and using relation 22.26)

$$n \;=\; \sum_{d\backslash n} \varphi(d) \tag{22.34}$$

while relation 22.29 corresponds to

$$\varphi(n) \;=\; \sum_{d\backslash n} d\;\mu(n/d) \tag{22.35}$$

Relations 22.34 and 22.35 are a special case of the additive version of the Möbius inversion principle:

$$g(n) \;=\; \sum_{d\backslash n} f(d) \quad\Longleftrightarrow\quad f(n) \;=\; \sum_{d\backslash n} g(d)\;\mu(n/d) \tag{22.36}$$

More general, if $g(a\,b) = g(a)\,g(b)$ then

$$h(n) \;=\; \sum_{d\backslash n} f(d)\,g(n/d) \quad\Longleftrightarrow\quad f(n) \;=\; \sum_{d\backslash n} g(d)\,h(n/d)\;\mu(n/d) \tag{22.37}$$

The primitive binary polynomials of degree n can be obtained by factoring Y_N over $GF(2)$ where $N = 2^n - 1$. For example, with $n = 6$:

```
? n=6; N=2^n-1; factormod(polcyclo(N),2)
    [Mod(1, 2)*x^6 + Mod(1, 2)*x + Mod(1, 2) 1]
    [Mod(1, 2)*x^6 + Mod(1, 2)*x^4 + Mod(1, 2)*x^3 + Mod(1, 2)*x + Mod(1, 2) 1]
    [Mod(1, 2)*x^6 + Mod(1, 2)*x^5 + Mod(1, 2) 1]
    [Mod(1, 2)*x^6 + Mod(1, 2)*x^5 + Mod(1, 2)*x^2 + Mod(1, 2)*x + Mod(1, 2) 1]
    [Mod(1, 2)*x^6 + Mod(1, 2)*x^5 + Mod(1, 2)*x^3 + Mod(1, 2)*x^2 + Mod(1, 2) 1]
    [Mod(1, 2)*x^6 + Mod(1, 2)*x^5 + Mod(1, 2)*x^4 + Mod(1, 2)*x + Mod(1, 2) 1]
```

We use a routine (`pcfprint(N)`) that prints the N-th cyclotomic polynomial and its factors in symbolic form. With $n = 6$, $N = 2^n - 1 = 63$ we get:

```
? n=6; N=2^n-1;
? pcfprint(N)
    63: [ 36 33 27 24 18 12 9 3 0 ]
     [ 6 1 0 ]
     [ 6 4 3 1 0 ]
     [ 6 5 0 ]
     [ 6 5 2 1 0 ]
     [ 6 5 3 2 0 ]
     [ 6 5 4 1 0 ]
```

The irreducible but non-primitive binary polynomials are factors of cyclotomic polynomials Y_d where $d \backslash N$, $d < N$ and the order of 2 modulo d equals n:

```
? fordiv(N,d,if(n==znorder(Mod(2,d)) && (d<N), pcfprint(d) ));
    9: [ 6 3 0 ]
     [ 6 3 0 ]
    21: [ 12 11 9 8 6 4 3 1 0 ]
     [ 6 4 2 1 0 ]
     [ 6 5 4 2 0 ]
```

The number of factors of Y_d equals $\varphi(d)/n$ so we can count how many degree-n irreducible polynomials correspond to which divisor of $N = 2^n - 1$:

```
     1:  [1:1]    1
     2:  [3:1]    1
     3:  [7:2]    2
     4:  [5:1]  [15:2]    3
     5:  [31:6]    6
     6:  [9:1]  [21:2]  [63:6]    9
     7:  [127:18]    18
     8:  [17:2]  [51:4]  [85:8]  [255:16]    30
     9:  [73:8]  [511:48]    56
    10:  [11:1]  [33:2]  [93:6]  [341:30]  [1023:60]    99
    11:  [23:2]  [89:8]  [2047:176]    186
```

Line 6 tells us that one irreducible polynomial is due to the factor 9, two are due to the factor 21 and the 6 primitive polynomials correspond to $N = 63$ itself which we have verified a moment ago. The list was produced using

```
{ for (n=1, 11,
    print1(n,": ");
    s = 0;
    N = 2^n-1;
    fordiv (N, d,
        if ( n==znorder(Mod(2,d)) ,
            a = eulerphi(d)/n;
            print1(" [",d,":",a,"] ");
            s += a;
        );
    );
    print("   ",s);
 ); }
```

For further information about cyclotomic polynomials see [Intr2FF]. The Möbius inversion principle is nicely explained in [ConcMath].

22.8 Factorization of binary polynomials

This section treats the factorization of binary polynomials. The first part describes how to factorize polynomials that do not contain a square factor. The second part gives algorithms to detect and remove square factors. Finally, an algorithm to factorize arbitrary binary polynomials is given.

22.8.1 Factorization of squarefree polynomials

In order to factorize a polynomial that must not contain a square factor we will use *Berlekamp's Q-matrix algorithm*. The algorithm consists of two main steps: The computation of the nullspace of a matrix and a refinement phase that finds the distinct irreducible factors.

Let c be a binary polynomial of degree d. The Q-matrix is a $d \times d$ matrix whose n-th column can be computed as the binary polynomial $x^{2\,n} \pmod c$. The algorithm will use the nullspace of $Q - \mathrm{id}$.

The routine to compute the Q-matrix is [FXT: `setup_q_matrix()` in bpol/berlekamp.cc]:

```
void
setup_q_matrix(ulong c, ulong d, ulong *ss)
// Compute the Q-matrix for the degree-d polynomial c.
// Used in Berlekamp's factorization algorithm.
{
    ulong h = 1UL << (d-1);
    {
        ulong x2 = 2UL;  // == 'x'
        ulong q = 1UL;
        x2 = bitpolmod_mult(x2, x2, c, h);
        for (ulong k=0;  k<d;  ++k)
        {
            ss[k] = q;
            q = bitpolmod_mult(q, x2, c, h);
        }
        bitmat_transpose(ss, d, ss);
    }
}
```

For the irreducible binary polynomial $c = x^7 + x + 1$ we get

```
     Q=         Q-id=
    1.....     .....
    .1...1.    .1...1.
    ..1..1.    ..11..1
    ...1...    ...1...
    ..1..1.    ..1.11.
    .....1.    .....11
    ....1..1   ....1...
  nullspace=
    1......
```

This is the output of the demo [FXT: demo gf2n/qmatrix-demo.cc]. The vector $n_0 = [1, 0, \ldots, 0]$ lies in the nullspace of $Q - \mathrm{id}$ for every polynomial c.

For $c = x^7 + x^3 + x + 1 = (x + 1)\,(x^2 + x + 1)\,(x^4 + x + 1)$ one obtains:

```
     Q=         Q-id=
    1.....     .....
    ...1.1.    ...1.1.
    .1..1.1    .11.1.1
    ..1...1    ..1...1
    ..1.111    ..1..11
    .....1.    .....11
    ...1.11    ...1.1.
    nullspace=
    1......
    .1..1..
    .1.1.11
```

In fact, the rank[2] r of the nullspace equals the number of distinct irreducible factors of c. That is, for polynomials containing a square factor we do not get the total number of factors. For example, with $c = (1 + x)^7 = x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$ one gets

```
     Q=         Q-id=
    1...1..    .1...1..
    .1...1.    .11...1.
    ..1...1    ..1.1.1
    .....1.    .....1.
    ..1...1    ..1.1.1
    .....1.    .....1.
    ...1...    ...1..1
    nullspace=
    1......
```

The vectors spanning the nullspace must be post-processed in order to obtain the irreducible factors of c. The algorithm can be described as follows: Let F be a set of binary polynomials whose product equals c, the refinement step R_i proceeds as follows:
Let t be the i-th element of the nullspace. For each element $f \in F$ do the following: If the degree of f equals one, keep it in the set, else delete f from the set and add from the set $X = \{\gcd(f, t), \gcd(f, t+1)\}$ those elements whose degrees are greater or equal to one.

[2] The rank of the nullspace equals the minimal number of vectors spanning the nullspace.

One starts with $F = \{c\}$ and does the refinement steps $R_0, R_1, \ldots R_{r-1}$ corresponding to the vectors of the nullspace. Afterwards the set F will contain exactly the distinct irreducible factors of c.

This is implemented in the routine [FXT: `bitpol_refine_factors()` in bpol/berlekamp.cc]:

```
ulong
bitpol_refine_factors(ulong *f, ulong nf, const ulong *nn, ulong r)
// Given the nullspace nn[0,...,r-1] of (Q-id)
// and nf factors f[0,...,nf-1] whose product equals c
// (typically nf=1 and f[0]==c)
// then get all r irreducible factors of c.
{
    ulong ss[r];
    for (ulong j=0; j<r; ++j) // for all elements t in nullspace
    {
        ulong t = nn[j];

        // skip trivial elements in nullspace:
        if ( bitpol_deg(t)==0 )  continue;

        ulong sc = 0;
        for (ulong b=0; b<nf; ++b)  // for all elements bv in set
        {
            ulong bv = f[b];
            ulong db = bitpol_deg(bv);
            if ( db <= 1 )  // bv cannot be reduced
            {
                ss[sc++] = bv;
            }
            else
            {
                for (ulong s=0; s<2; ++s)  // for all elements in GF(2)
                {
                    ulong ti = t ^ s;
                    ulong g = bitpol_gcd(bv, ti);
                    if ( bitpol_deg(g) >= 1 )  ss[sc++] = g;
                }
            }
        }

        nf = sc;
        for (ulong k=0; k<nf; ++k)  f[k] = ss[k];

        if ( nf>=r )  break;  // nf>r would be an error
    }

    return nf;
}
```

As can be seen, one can skip elements corresponding to constant polynomials. Further, as soon as the set F contains r elements, all factors are found and one can terminate the algorithm.

Now Berlekamp's algorithm can be implemented as

```
ulong
bitpol_factor_squarefree(ulong c, ulong *f)
// Fill irreducible factors of squarefree polynomial c into f[]
// Return number of factors.
{
    ulong d = bitpol_deg(c);

    if ( d<=1 )  // trivial cases:  0, 1, x, x+1
    {
        f[0] = c;
        if ( 0==c )  d = 1;  // 0==0^1
        return d;
    }

    ulong ss[d];
    setup_q_matrix(c, d, ss);
    bitmat_add_unit(ss, d);

    ulong nn[d];
    ulong r = bitmat_nullspace(ss, d, nn);

    f[0] = c;
    ulong nf = 1;
    if ( r>1 )  nf = bitpol_refine_factors(f, nf, nn, r);
```

```
    return r;
}
```

The algorithm for the computation of the nullspace was taken from [DEK2], find the implementation in [FXT: file bmat/bitmatnullspace.cc].

Berlekamp's algorithm is given in [Cohen] in a more general form: In order to factorize a polynomial with coefficients in $\mathbb{F}_q$ one has to set up the Q-matrix with columns $x^{q\,i}$ and in the refinement step X has to be the set $X = \{\gcd(f, t+0), \gcd(f, t+1), \ldots, \gcd(f, t+(q-1))\}$. The algorithm is efficient only if q is small.

22.8.2 Extracting the squarefree part of a polynomial

To test whether a polynomial c has a square factor one computes $g = \gcd(c, c')$ where c' is the derivative. If $g \neq 1$ then c has a square factor. The corresponding routine for binary polynomials is given in [FXT: file bpol/bitpolsquarefree.h]:

```
inline ulong bitpol_test_squarefree(ulong c)
// Return  0 if polynomial is squarefree
// else return square factor != 0
{
    ulong d = bitpol_deriv(c);
    if ( 0==d )  return  (1==c ? 0 : c);
    ulong g = bitpol_gcd(c, d);
    return  (1==g ? 0 : g);
}
```

The derivative of a binary polynomial can be computed easily [FXT: file bpol/bitpolderiv.h]:

```
inline ulong bitpol_deriv(ulong c)
// Return derived polynomial
{
#if  BITS_PER_LONG >= 64
    c &= 0xaaaaaaaaaaaaaaaaUL;
#else
    c &= 0xaaaaaaaaUL;
#endif
    return  (c>>1);
}
```

The coefficients at the even powers have to be deleted because derivation multiplies them with an even factor which equals zero modulo two.

If the derivative of a binary polynomial is zero, then it is a perfect square:

```
inline ulong bitpol_pure_square_q(ulong c)
// Return whether polynomial is a pure square != 1
{
    if ( 1UL==c )  return 0;
#if  BITS_PER_LONG >= 64
    c &= 0xaaaaaaaaaaaaaaaaUL;
#else
    c &= 0xaaaaaaaaUL;
#endif
    return  (0==c);
}
```

The routine

```
inline ulong bitpol_test_squarefree(ulong c)
{
    ulong d = bitpol_deriv(c);
    if ( 0==d )  return  (1==c ? 0 : c);
    ulong g = bitpol_gcd(c, d);
    return  (1==g ? 0 : g);
}
```

returns zero if c is squarefree. If c is has a square factor $s \neq 1$ then s is returned. In case a polynomial is a perfect square then its square root can be computed as

```
inline ulong bitpol_pure_sqrt(ulong c)
// Return t = sqrt(c)
// c must be a pure square.
{
    ulong t = 0;
    for (ulong mc=1,mt=1;  mc;  mc<<=2,mt<<=1)
    {
        if ( mc & c )  t |= mt;
    }
    return  t;
}
```

For the factorization algorithm for general polynomials we have to extract the product of all distinct irreducible factors (the squarefree part) from a polynomial. The routine [FXT: `bitpol_sreduce()` in bpol/bitpolsquarefree.cc] returns a polynomial where the even exponents are reduced:

```
ulong
bitpol_sreduce(ulong c)
{
    ulong s = bitpol_test_squarefree(c);
    if ( 0==s )  return c;  // c is squarefree

    ulong f = bitpol_div(c, s);

    do  // here s is a pure square and s>1
    {
        s = bitpol_pure_sqrt(s);
    }
    while ( bitpol_pure_square_q(s) );

    ulong g = bitpol_gcd(s, f);
    s = bitpol_div(s, g);
    f = bitpol_mult(f, s);

    return f;
}
```

With $c = f \cdot s^{k\,2^t}$ (k odd, the factors of f and s not necessarily distinct) the returned polynomial equals $f \cdot s^k$. For example,

$$\begin{aligned}
a^2 &\mapsto a \\
a^4 &\mapsto a \\
a^3 = a\,a^2 &\mapsto a\,a \\
a^5 = a\,a^4 &\mapsto a\,a \\
a\,b^2 &\mapsto a\,b \\
a\,b\,b^2 &\mapsto a\,b\,b = a\,b^2 \mapsto a\,b \\
f \cdot s^{k\,2^t} &\mapsto f \cdot s^k
\end{aligned}$$

To extract the squarefree part of a polynomial call the routine repeatedly until the returned polynomial equals the input:

```
inline ulong bitpol_make_squarefree(ulong c)
{
    ulong z = c, t;
    while ( z!=(t=bitpol_sreduce(z)) )  z = t;
    return z;
}
```

The reduction routine will be called at most $\log_2(n)$ times for a polynomial of degree n: The worst case is a perfect power $p = a^{2^k-1}$ where $2^k - 1 \le n$. Observe that $(2^k - 1) = 1 + 2\,(2^{k-1} - 1)$, so the reduction routine will split p as $p = a\,s^2 \mapsto a\,s$ where $s = a^{2^{k-1}-1}$ is of the same form.

22.8.3 Factorization of arbitrary polynomials

The factorization routine for arbitrary binary polynomials [FXT: `bitpol_factor()` in bpol/bitpolfactor.cc] extracts the squarefree part f of its input c, uses Berlekamp's algorithm to factor f and updates the exponents according to the polynomial $s = c/f$:

```
ulong
bitpol_factor(ulong c, ulong *f, ulong *e)
// Factorize the binary polynomial c:
// c = \prod_{i=0}^{fct-1}{f[i]^e[i]}
// The number of factors (fct) is returned.
{
    ulong d = bitpol_deg(c);
    if ( d<=1 )  // trivial cases:  0, 1, x, x+1
    {
        f[0] = c;
        if ( 0==c )  d = 1;  // 0==0^1
        return d;
    }

    // get squarefree part:
    ulong cf = bitpol_make_squarefree(c);

    // ... and factor it:
    ulong fct = bitpol_factor_squarefree(cf, f);

    // All exponents are one:
    for (ulong j=0; j<fct; ++j)  { e[j] = 1; }

    // Here f[],e[] is a valid factorization of the squarefree part cf

    // Update exponents with square part:
    ulong cs = bitpol_div(c, cf);
    for (ulong j=0; j<fct; ++j)
    {
        if ( 1==cs )  break;
        ulong fj = f[j];
        ulong g = bitpol_gcd(cs, fj);
        while ( 1!=g )
        {
            ++e[j];
            cs = bitpol_div(cs, fj);
            if ( 1==cs )  break;
            g = bitpol_gcd(cs, fj);
        }
    }
    return fct;
}
```

As given the algorithm makes just one call to the routine that computes a nullspace.

$$
\begin{aligned}
x^5 &= (x)^5 \\
x^5+1 &= (x+1)\cdot\left(x^4+x^3+x^2+x+1\right) \\
x^5+x &= (x)\cdot(x+1)^4 \\
x^5+x+1 &= \left(x^3+x^2+1\right)\cdot\left(x^2+x+1\right) \\
x^5+x^2 &= (x)^2\cdot(x+1)\cdot\left(x^2+x+1\right) \\
x^5+x^2+1 &= \left(x^5+x^2+1\right) \\
x^5+x^2+x &= (x)\cdot\left(x^4+x+1\right) \\
x^5+x^2+x+1 &= (x+1)^2\cdot\left(x^3+x+1\right) \\
x^5+x^3 &= (x)^3\cdot(x+1)^2 \\
x^5+x^3+1 &= \left(x^5+x^3+1\right)
\end{aligned}
$$

Figure 22.6: Factorizations of the first few binary polynomials of degree 5.

The demo [FXT: demo gf2n/bitpolfactor-demo.cc] prints the factorizations of the first few binary polynomials of given degree. Part of its output is shown in figure 22.6. Factoring the first million polynomials of degrees 20, 30, 40 and 60 takes about 5, 10, 15 and 30 seconds, respectively.

A variant of the factorization algorithm often given uses the so-called *squarefree factorization* $c = \prod_i a_i^i$ where the a_i are squarefree and pairwise coprime. Given the squarefree factorization one has to call the core routine for each non-trivial a_i.

As noted, the refinement step gets expensive if the coefficients are in a field $GF(q)$ where q is not small because q computations of the polynomial gcd are involved. For an algorithm that is efficient also for large values of q see [Cohen].

Chapter 23

Binary finite fields: $GF(2^n)$

This chapter introduces the binary finite fields $GF(2^n)$. The polynomial representation is used for stating some basic properties. An introduction of the representation by normal bases follows. Certain normal bases are advantageous for hardware implementations of the arithmetic algorithms. Finally, several ways of computing the number of irreducible binary normal polynomials are given.

Binary finite fields are important for applications like error correcting codes and cryptography.

The underlying arithmetical algorithms are given in chapter 22.

23.1 Arithmetic and basic properties

In chapter 13 we have met the finite fields $\mathbb{Z}/p\mathbb{Z} = GF(p)$ for p a prime. The 'GF' stands for *Galois Field*, another symbol often used is $\mathbb{F}_p$. The arithmetic in $GF(p)$ is the simply the arithmetic modulo p.

There are more finite fields: for every prime p there are fields with $Q = p^n$ elements for all $n \geq 1$. All elements in a finite field $GF(p^n)$ can be represented as polynomials modulo an degree-n irreducible polynomial C with coefficients over the field $GF(p)$. The arithmetic to be used is polynomial arithmetic modulo C. As in general there is more than one irreducible polynomial of degree n it might seem that there is more than one field $GF(Q)$ for given $Q = p^n$. There isn't. Using different polynomials as modulus leads to isomorphic representations of the same field. The field $GF(p^n)$ is called an *extension field* of $GF(p)$. The field $GF(p)$ is called the *ground field* of $GF(p^n)$.

When speaking about an element of $GF(Q)$ one can think of a polynomial modulo some fixed irreducible polynomial C (modulus). For example, the product of two elements can be computed as the polynomial product modulo C. For the equivalent construction using the polynomial $x^2 + 1$ with real coefficients that leads to the complex numbers, see section 21.11 on page 553.

The elements zero (the neutral element of addition) and one (neutral element of multiplication) are the constant polynomials with constant zero and one, respectively. This does not depend on the choice of the modulus.

Multiplication with an element of the ground field is called *scalar multiplication*. In this section an element of the ground field is denoted by u. Scalar multiplication corresponds to the multiplication of every coefficient of (the polynomial representing) the field element a by u. We call a function $f(.)$ *linear* if the relation

$$f(u_1 \cdot a + u_2 \cdot b) \quad = \quad u_1 \cdot f(a) + u_2 \cdot f(b) \tag{23.1}$$

holds for u_1 and u_2 from the ground field.

We restrict our attention to $Q = 2^n$, that is, the *binary finite fields* $GF(2^n)$ in what follows as we have seen the algorithms for the underlying arithmetic.

Inverse and square root

The number of elements in $GF(Q)$ equals Q. For any element $a \in GF(Q)$ one has $a^Q = a$ and $a^{Q-1} = 1$. So we can compute the *inverse* of every nonzero element a as

$$a^{-1} \quad = \quad a^{Q-2} \tag{23.2}$$

The number of invertible elements (units) in $GF(Q)$ is clearly $|GF(Q)^*| = Q - 1 = p^n - 1$.

Every element a of $GF(2^n)$ has a unique *square root* s that can easily be computed as

$$s \quad = \quad a^{Q/2} \; = \; a^{2^{n-1}} \tag{23.3}$$

It is computed by squaring the element $n - 1$ times.

Squaring

For $GF(p^n)$ one has

$$(u+v)^p \quad = \quad u^p + v^p \tag{23.4}$$

because the binomial coefficients $\binom{p}{k}$ are divisible by p for $k = 1, 2, \ldots, p-1$. For $GF(2^n)$:

$$(u+v)^2 \quad = \quad u^2 + v^2 \tag{23.5}$$

Squaring (and raising to a power 2^k) is a linear operation in $GF(2^n)$. Thereby, via relation 23.3, we see that taking the square root is also linear. The linearity can be used to accelerate the computation of squares. Write

$$\begin{aligned} \left(u_0 + u_1\, x + u_2\, x^2 + \ldots + u_{n-1}\, x^{n-1}\right)^2 \quad &= \quad u_0 + u_1\, x^2 + u_2\, x^4 + \ldots + u_{n-1}\, x^{2\,(n-1)} \\ &=: \quad u_0\, s_0 + u_1\, s_1 + u_2\, s_2 + \ldots u_{n-1}\, s_{n-1} \end{aligned} \tag{23.6}$$

One has to precompute and store the values $s_i = x^{2\,i} \bmod C$ for $i = 0, 1, 2, \ldots, n-1$. For successive square computations it is only necessary to add (that is, XOR) those s_i corresponding to nonzero u_i. For example, with $n = 13$ and the polynomial modulus $C = x^{13} + x^4 + x^3 + x^1 + 1$ one obtains the table

```
 S0 =  ............1
 S1 =  ...........1..
 S2 =  ..........1...
 S3 =  .........1....
 S4 =  ........1.....
 S5 =  .......1......
 S6 =  ......1.......
 S7 =  .....11.11....
 S8 =  ....11.11.....
 S9 =  ...11.11......
S10 =  ..11.11.......
S11 =  .1.11...11.11
S12 =  .11....1.11.1.
```

The squares $s_0, s_1 \ldots s_{\lfloor (n-1)/2 \rfloor}$ have the trivial form $s_i = x^{2\,i}$ which can be used to further accelerate the computation.

Trace

The *trace* of an element u in $GF(2^n)$ is defined as

$$\mathrm{Tr}(a) \quad := \quad a + a^2 + a^4 + a^8 + \ldots + a^{2^{n-1}} \; = \; \sum_{j=0}^{n-1} a^{2^j} \tag{23.7}$$

The trace of any element is either zero or one. The trace of zero is always zero. The trace of one in $GF(2^n)$ equals one for odd n, else zero, that is, $\mathrm{Tr}(1) = n \bmod 2$. Exactly half of the elements have trace one. The trace of the sum of two elements is the sum of the traces of the elements:

$$\mathrm{Tr}(a+b) \quad = \quad \mathrm{Tr}(a) + \mathrm{Tr}(b) \tag{23.8}$$

The trace of the square of an element is the trace of that element:

$$\mathrm{Tr}(a^2) \quad = \quad \mathrm{Tr}(a) \tag{23.9}$$

With u zero or one (an element of the ground field $GF(2)$) one has

$$\mathrm{Tr}(u \cdot a) \quad = \quad u \cdot \mathrm{Tr}(a) \tag{23.10}$$

Thereby, for u_1 and u_1 from the ground field,

$$\mathrm{Tr}(u_1 \cdot a + u_2 \cdot b) \quad = \quad u_1 \cdot a + u_2 \cdot b \tag{23.11}$$

That is, the Tr(.) function is linear.

A fast algorithm to compute the trace uses a precomputed table $t_i = \mathrm{Tr}(x^i)$ for $i = 0, 1, 2, \ldots, n-1$ and the linearity of the trace

$$\mathrm{Tr}(u) \quad = \quad \sum_{i=0}^{n-1} u_i \, t_i \tag{23.12}$$

where the u_1 are zero or one. Thereby, when the polynomial fits into a single binary word, precompute a 'trace vector' `tv` whose bits are the traces of the powers of x and later compute the trace of an element via [FXT: `gf2n_fast_trace()` in bpol/gf2nfuncs.h]:

```
static inline ulong gf2n_fast_trace(ulong a, ulong tv)
// Fast trace computation of the trace of a
//  using the precalculated table tv.
{ return  parity( a & tv ); }  // scalar product over GF(2)
```

Given the trace vector it is also easy to find elements of trace zero or one by simply taking the lowest unset or set bit of the vector, respectively.

Order and primitive roots

The *order* of an element a is the least positive exponent r so that $a^r = 1$. The maximal order of an element in $GF(2^n)$ equals $2^n - 1 = Q - 1$. An element of maximal order is called a *generator* (or *primitive root*) as its powers 'generate' all nonzero elements in the field. The order of an given element a in $GF(2^n)$ can be computed like

```
function order(a, n)
{
    if a==0 then  return 0  // a not a unit

    h := 2**n - 1  // number of units
    e := h
    {np, p[], k[]} := factorization(h)  // h==product(i=0..np-1, p[i]**k[i])
    for i:=0 to np-1
    {
        f := p[i]**k[i]
        e := e / f
        g1 := a**e  // modulo polynomial
        while g1!=1
        {
            g1 := g1**p[i]  // modulo polynomial
            e := e * p[i]
            p[i] := p[i] - 1
        }
    }
    return e
}
```

The C++ implementation is [FXT: `gf2n_order()` in bpol/gf2norder.cc]. We have seen a very similar algorithm in section 22.4.4 on page 589. Indeed, with just minimal changes we get a pari/gp implementation to compute the order of an element g in $GF(2^n)$:

```
polorder(p, g='x) =
/* Order of g modulo p (p irreducible over GF(2)) */
{
  local(g1, te, tp, tf, tx);
  p *= Mod(1,2);
  te = nn_;
  for(i=1, np_,
    tf = vf_[i];  tp = vp_[i];  tx = vx_[i];
    te = te / tf;
    g1 = Mod(g, p)^te;
    while ( 1!=g1,
      g1 = g1^tp;
      te = te * tp;
    );
  );
  return( te );
}
```

The only difference to the original algorithm is that g can have other values than (the polynomial) x. The function uses the following global variables that must be set up before call:

```
nn_ = 0;  /* max order  =  2^n-1 */
np_ = 0;  /* number of primes in factorization */
vp_ = []; /* vector of primes */
vf_ = []; /* vector of factors (prime powers) */
vx_ = []; /* vector of exponents */
```

To check whether g is a primitive root modulo p use the function

```
polmaxorder_q(p, g='x) =
/* Whether order of g modulo p is maximal  (p irreducible over GF(2)) */
/* Early-out variant */
{
  local(g1, te, tp, tf, tx, ct);
  p *= Mod(1,2);
  te = nn_;
  for(i=1, np_,
    tf = vf_[i];  tp = vp_[i];  tx = vx_[i];
    te = te / tf;
    g1 = Mod(g, p)^te;
    ct = 0;
    while ( 1!=g1,
      g1 = g1^tp;
      te = te * tp;
      ct = ct + 1;
    );
    if ( ct<tx,  return(0) );
  );
  return(1);
}
```

23.2 Implementation

A C++ class that allows to do computations in the fields $GF(2^n)$ for n not greater than `BITS_PER_LONG` is [FXT: `class GF2n` in bpol/gf2n.h]:

```
class GF2n
// class to implement a GF(2**n) type
// and arithmetic operations on it
//
// GF2n::init(n)  _MUST_ be called before usage
{
public:
    ulong v_;
public:
    static ulong n_;  // the 'n' in GF(2**n)
```

```
    static ulong c_;  // polynomial modulus
    static ulong h_;  // auxiliary bitmask for computations
    static ulong mm_; // 2**n - 1 == max order (a Mersenne number)
    static ulong g_;  // a generator (element of maximal order)
    static ulong tv_; // trace vector
    static ulong sqr_tab[BITS_PER_LONG]; // table for fast squaring
    static factorization mfact_; // factorization of max order
    static char* pc_; // chars to print zero and one: e.g.  "01" or ".1"

    static ulong is_normal_;  // whether field polynomial is normal
    // tables for basis conversion:
    static ulong p2n_tab[BITS_PER_LONG]; // polynomial to normal
    static ulong n2p_tab[BITS_PER_LONG]; // normal to polynomial

    static GF2n zero; // zero (neutral element wrt. addition) in GF(2**n)
    static GF2n one;  // one (neutral element wrt. multiplication) in GF(2**n)
    static GF2n tr1e; // an element with trace == 1
public:
    explicit GF2n()  { ; }
    explicit GF2n(const ulong i) : v_(i & mm_)  { ; }
    GF2n(const GF2n &g) : v_(g.v_)  { ; }
    ~GF2n()  { ; }

    // bpol/gf2n.cc:
    static void init(ulong n, ulong c=0, bool normalq=0);
[--snip--]
```

Before doing anything one has to call `GF2n::init(n)` which accepts a second argument that, if supplied, must be an irreducible polynomial of degree n.

```
      n = 4  GF(2^n)
      c = 1..11 (polynomial modulus)
       == 4 1 0  (positions where coefficients are one)
      mm= .1111 (maximal order)
       == 15  =  3 * 5
      h = .1... (aux. bitmask)
      g = ...1. (element of maximal order)
      tv= .1... (traces of x^i)
    tr1e= 1... (element with trace=1)

       k  :   f:=g**k  t:=trace(f)         s:=f*f  q:=sqrt(f)
                                r:=order(f)
     .... :   f= ...1  t= 0  r=    1      s= ...1  q= ...1
     ...1 :   f= ..1.  t= 0  r=   15      s= .1..  q= .1.1
     ..1. :   f= .1..  t= 0  r=   15      s= ..11  q= ..1.
     ..11 :   f= 1...  t= 1  r=    5      s= 11..  q= 1.1.
     .1.. :   f= ..11  t= 0  r=   15      s= .1.1  q= .1..
     .1.1 :   f= .11.  t= 0  r=    3      s= .111  q= .111
     .11. :   f= 11..  t= 1  r=    5      s= 1111  q= 1...
     .111 :   f= 1.11  t= 1  r=   15      s= 1..1  q= 111.
     1... :   f= .1.1  t= 0  r=   15      s= ..1.  q= ..11
     1..1 :   f= 1.1.  t= 1  r=    5      s= 1...  q= 1111
     1.1. :   f= .111  t= 0  r=    3      s= .11.  q= .11.
     1.11 :   f= 111.  t= 1  r=   15      s= 1.11  q= 11.1
     11.. :   f= 1111  t= 1  r=    5      s= 1.1.  q= 11..
     11.1 :   f= 11.1  t= 1  r=   15      s= 111.  q= 1..1
     111. :   f= 1..1  t= 1  r=   15      s= 11.1  q= 1.11
```

Figure 23.1: Powers of the generator x in $GF(2^4)$ when a primitive polynomial is used as a modulus.

A demo of the arithmetics is [FXT: demo gf2n/gf2n-demo.cc], it shows the successive powers of a primitive root. By default computations in $GF(2^4)$ are shown, both for a primitive polynomial modulus (figure 23.1) and for a non-primitive polynomial modulus (figure 23.2 on the next page).

Let a be an invertible element (that is, $a \neq 0$), then a can be written as a power of a generator: $a = g^k$. Then for the order r of a we have

$$r(g^k) \quad = \quad \frac{N}{\gcd(k, N)} \tag{23.13}$$

```
  n = 4  GF(2^n)
  c = 11111 (polynomial modulus) [NON-primitive]
   == 4 3 2 1 0  (positions where coefficients are one)
  mm= .1111 (maximal order)
   == 15  =  3 * 5
  h = .1... (aux. bitmask)
  g = ...11 (element of maximal order)
  tv= .111. (traces of x^i)
 tr1e= ..1. (element with trace=1)

    k  :   f:=g**k  t:=trace(f)        s:=f*f  q:=sqrt(f)
                              r:=order(f)
 .... :   f= ...1  t= 0  r=    1      s= ...1  q= ...1
 ...1 :   f= ..11  t= 1  r=   15      s= .1.1  q= 1..1
 ..1. :   f= .1.1  t= 1  r=   15      s= 111.  q= ..11
 ..11 :   f= 1111  t= 1  r=    5      s= 1...  q= .1..
 .1.. :   f= 111.  t= 1  r=   15      s= 1..1  q= .1.1
 .1.1 :   f= 11.1  t= 0  r=    3      s= 11..  q= 11..
 .11. :   f= 1...  t= 1  r=    5      s= ..1.  q= 1111
 .111 :   f= .111  t= 0  r=   15      s= 1.1.  q= 1.11
 1... :   f= 1..1  t= 1  r=   15      s= ..11  q= 111.
 1..1 :   f= .1..  t= 1  r=    5      s= 1111  q= ..1.
 1.1. :   f= 11..  t= 0  r=    3      s= 11.1  q= 11.1
 1.11 :   f= 1.11  t= 0  r=   15      s= .111  q= .11.
 11.. :   f= ..1.  t= 1  r=    5      s= .1..  q= 1...
 11.1 :   f= .11.  t= 0  r=   15      s= 1.11  q= 1.1.
 111. :   f= 1.1.  t= 0  r=   15      s= .11.  q= .111
```

Figure 23.2: Powers of a generator in $GF(2^4)$ when a non-primitive polynomial is used as a modulus.

where $N = 2^n - 1$. For N a (Mersenne-) prime the order of all invertible elements except 1 is N.

23.3 Minimal polynomials

The *minimal polynomial* of an element a in $GF(2^n)$ is defined as the polynomial of least degree which has a as a root. The minimal polynomial can be computed as the product

$$p_a(x) \quad := \quad \prod_{k=0}^{r-1} \left(x - a^{2^k} \right) \tag{23.14}$$

where r is the smallest positive integer so that $a^{2^r} = a$. The minimal polynomial of any element is irreducible and its degree is a divisor of n.

From the definition it can be seen that the coefficients of the minimal polynomial lie in $GF(2^n)$, however, all of them are either zero or one. Thereby the computation has to be carried out using the arithmetic in $GF(2^n)$ but the final result is a binary polynomial [FXT: `minpoly()` in bpol/minpoly.cc]:

```
ulong
minpoly(GF2n a, ulong &bp)
// Compute the minimal polynomial p(x) of a \in GF(2**n).
// Return the degree of p().
// The polynomial p() is written to bp
{
    GF2n p[BITS_PER_LONG+1];
    ulong n = GF2n::n_;
    for (ulong k=0; k<=n; ++k)  p[k] = 0;
    p[0] = 1;
    ulong d;
    GF2n s = a;
    for (d=1; d<=n; ++d)
    {
        for (ulong k=d; 0!=k; --k)  p[k] = p[k-1];
        p[0] = 0;
        for (ulong k=0; k<d; ++k)  p[k] += p[k+1] * s;
```

```
        s = s.sqr();
        if ( s == a )  break;
    }
    // Here all coefficients are either zero or one,
    //  so we can fill them into a binary word:
    ulong p2 = 0;
    for (ulong j=0; j<=d; ++j)  p2 |= (p[j].v_ << j);
    bp = p2;
    return  d;
}
```

By construction the zeros of the polynomial are $a, a^2, a^4, a^8, \ldots, a^{2^{r-1}}$. Note that $\left(a^{2^{r-1}}\right)^2 = a$, that is, $p_a(.) = p_{a^2}(.) = p_{a^4}(.) = \ldots$. The elements $a^2, a^4, \ldots$ are called the *conjugates* of a.

```
      n = 6  GF(2^n)
      c = 1....11 (polynomial modulus)
       == 6 1 0  (positions where coefficients are one)
      mm= .111111 (maximal order)
       == 63  =  3^2 * 7
      h = .1..... (aux. bitmask)
      g = .....1. (element of maximal order)
      tv= .1..... (traces of x^i)
    tr1e= .1..... (element with trace=1)

       k        :   f:=g**k  r:=order(f)   p:=minpoly(f)  t:=trace(f) d:=deg(p)
       0 = ..... :   f= .....1  r=    1   p=......11     t= 0        d=  1
       1 = .....1 :   f= ....1.  r=   63   p=.1....11     t= 0        d=  6
       2 = ....1. :   f= ...1..  r=   63   p=.1....11     t= 0        d=  6
       3 = ....11 :   f= ..1...  r=   21   p=.1.1.111     t= 0        d=  6 N
       4 = ...1.. :   f= .1....  r=   63   p=.1....11     t= 0        d=  6
       5 = ...1.1 :   f= 1.....  r=   63   p=.11..111     t= 1        d=  6
       6 = ...11. :   f= ....11  r=   21   p=.1.1.111     t= 0        d=  6 N
       7 = ...111 :   f= ...11.  r=    9   p=.1..1..1     t= 0        d=  6 N
       8 = ..1... :   f= ..11..  r=   63   p=.1....11     t= 0        d=  6
       9 = ..1..1 :   f= .11...  r=    7   p=....11.1     t= 0        d=  3
      10 = ..1.1. :   f= 11....  r=   63   p=.11..111     t= 1        d=  6
      11 = ..1.11 :   f= 1...11  r=   63   p=.11.11.1     t= 1        d=  6
      12 = ..11.. :   f= ...1.1  r=   21   p=.1.1.111     t= 0        d=  6 N
      13 = ..11.1 :   f= ..1.1.  r=   63   p=.1.11.11     t= 0        d=  6
      14 = ..111. :   f= .1.1..  r=    9   p=.1..1..1     t= 0        d=  6 N
      15 = ..1111 :   f= 1.1...  r=   21   p=.111.1.1     t= 1        d=  6 N
      16 = .1.... :   f= .1..11  r=   63   p=.1....11     t= 0        d=  6
      17 = .1...1 :   f= 1..11.  r=   63   p=.11..111     t= 1        d=  6
      18 = .1..1. :   f= ..1111  r=    7   p=....11.1     t= 0        d=  3
      19 = .1..11 :   f= .1111.  r=   63   p=.1.11.11     t= 0        d=  6
      20 = .1.1.. :   f= 1111..  r=   63   p=.11..111     t= 1        d=  6
    [--snip--]
      53 = 11.1.1 :   f= 1.1.1.  r=   63   p=.111..11     t= 1        d=  6
      54 = 11.11. :   f= .1.111  r=    7   p=....1.11     t= 0        d=  3
      55 = 11.111 :   f= 1.111.  r=   63   p=.11....1     t= 1        d=  6
      56 = 111... :   f= .11111  r=    9   p=.1..1..1     t= 0        d=  6 N
      57 = 111..1 :   f= 11111.  r=   21   p=.111.1.1     t= 1        d=  6 N
      58 = 111.1. :   f= 111111  r=   63   p=.111..11     t= 1        d=  6
      59 = 111.11 :   f= 1111.1  r=   63   p=.11....1     t= 1        d=  6
      60 = 1111.. :   f= 111..1  r=   21   p=.111.1.1     t= 1        d=  6 N
      61 = 1111.1 :   f= 11...1  r=   63   p=.11....1     t= 1        d=  6
      62 = 11111. :   f= 1....1  r=   63   p=.11....1     t= 1        d=  6
```

Figure 23.3: Minimal polynomials of the powers of a generator in $GF(2^6)$. Polynomials of degree $n = 6$ that are non-primitive are marked with an '`N`'. The trace of an element equals the coefficient of x^{n-1} of its minimal polynomial.

The demo [FXT: demo gf2n/gf2n-minpoly-demo.cc] prints the minimal polynomials for the powers of a primitive element g, see figure 23.3. The polynomials (of maximal degree) that are non-primitive are marked by an '`N`'. The minimal polynomials for g^k are non-primitive or of degree smaller than n whenever $\gcd(k, 2^n - 1) \neq 1$.

Let C be a irreducible polynomial of degree n and a an element so that $r = \mathrm{ord}_C(a)$ is the order of a modulo C. Then the order of x modulo the minimal polynomial of a is also r. Thereby a primitive polynomial can be determined from an irreducible polynomial C and a generator g modulo C by computing the minimal polynomial of g.

With a primitive polynomial (and the generator $g =$'x') the minimal polynomial of an element $f = g^k$ is primitive if k is a Lyndon word and $\gcd(k, n) = 1$. With a fast algorithm for the generation of Lyndon words one can therefore generate all primitive polynomials as shown in section 22.6 on page 594.

23.4 Solving quadratic equations

We want to solve, in $GF(2^n)$,

$$a\,x^2 + b\,x + c \quad = \quad 0 \tag{23.15}$$

The fact that extracting a square root of an arbitrary element in $GF(2^n)$ is easy does not enable us to solve the given equation. The formula $r_{0,1} = \left(-b \pm \sqrt{b^2 - 4ac}\right)/(2a)$ that works fine for real and complex numbers is of no help here: How should we divide by two?

Instead we transform the equation into a special form: Divide by a: $x^2 + (b/a)\,x + (c/a) = 0$, substitute $x = z\,(b/a)$ to get $z^2\,(b/a)^2 + (b/a)^2\,z + (c/a) = 0$ and divide by $(b/a)^2$ to get

$$z^2 + z + C \quad = \quad 0 \quad \text{where} \quad C = \frac{a\,c}{b^2} \tag{23.16}$$

If r_0 is one solution of this equation then $r_1 = r_0 + 1$ is the other one: $z\,(z+1) = C$. The equation doesn't necessarily have a solution at all, the trace of C must be 0.

The function [FXT: `solve_quadratic()` in bpol/solvequadratic.cc] transforms the equation into the reduced form:

```
bool solve_quadratic(GF2n a, GF2n b, GF2n c, GF2n& r0, GF2n& r1)
// Solve  a*x^2 + b*x + c == 0
// Return  whether solutions exist.
// If so, the solutions are written to r0 and r1.
{
    GF2n cc = a*c;
    cc /= (b.sqr());  // cc = (a*c)/(b*b)
    GF2n r;
    bool q = solve_reduced_quadratic(cc, r);
    if ( !q )  return  false;

    GF2n s = b / a;
    r0 = r * s;
    r1 = (r+GF2n::one) * s;

    return  true;
}
```

The function [FXT: `solve_reduced_quadratic()` in bpol/solvequadratic.cc] checks whether the reduced equation has a solution and if so, returns one solution:

```
bool
solve_reduced_quadratic(GF2n c, GF2n& r)
// Solve  z^2 + z == c
// Return  whether solutions exist.
// If so, one solutions is written to r.
// The other solution is r+1.
{
    if ( 1==c.trace() )  return  false;

    GF2n t( GF2n::tr1e );
    GF2n z( GF2n::zero );
    GF2n u( t );
    for (ulong j=1; j<GF2n::n_; ++j)
    {
        GF2n u2 = u.sqr();
        z = z.sqr();  z += u2*c;   // z = z*z + c*u*u
```

```
        u = u2 + t;                // u = u*u + t
    }
    r = z;
    return  true;
}
```

```
k=1:
    u=t^2 + t
    z=c*t^2
  z^2=c^2*t^4
z^2+z=c^2*t^4 + c*t^2

k=2:
    u=t^4 + t^2 + t
    z=(c^2 + c)*t^4 + c*t^2
  z^2=(c^4 + c^2)*t^8 + c^2*t^4
z^2+z=(c^4 + c^2)*t^8 + c*t^4 + c*t^2

k=3:
    u=t^8 + t^4 + t^2 + t
    z=(c^4 + c^2 + c)*t^8 + (c^2 + c)*t^4 + c*t^2
  z^2=(c^8 + c^4 + c^2)*t^16 + (c^4 + c^2)*t^8 + c^2*t^4
z^2+z=(c^8 + c^4 + c^2)*t^16 + c*t^8 + c*t^4 + c*t^2
     =(c^8 + c^4 + c^2)*t    + c*(t^8 + t^4 + t^2)    [using t^16=t]
     =( c + trace(c)  )*t    + c*( t + trace(t) )     [using x^8+x^4+x^2=x+trace(x)]
     =( c + 0  )*t    + c*( t + 1 )                   [using trace(c)=0, trace(t)=1]
     =                         c                     [ z^2+z==c ]
```

Figure 23.4: Solving the reduced quadratic equation $z^2 + z = c$ in $GF(2^4)$.

The algorithm is (for a more general setting) given in [95], the given implementation takes advantage of a precomputed element with trace one. At the end of step $k \geq 1$ we have

$$u_k = \sum_{j=0}^{k} t^{2^k} \tag{23.17a}$$

$$z_k = \sum_{j=0}^{k-1} \left[t^{2^k} \sum_{i=0}^{k-1} c^{2^i} \right] \tag{23.17b}$$

Figure 23.4 shows (for $GF(2^4)$) that this expression is the solution sought.

The program [FXT: demo gf2n/gf2n-solvequadratic-demo.cc] gives the solutions to the reduced equations $x^2 + x = f$ for all f with trace zero. Both for a primitive and a non-primitive modulus, see figure 23.5 on the next page.

Routines for the solution of quadratic equations that do not depend on the class `GF2n` are given in [FXT: file bpol/bitpolmodsolvequadratic.cc].

23.5 Representation by matrices *

With a primitive polynomial modulus $C(x)$ a representation of the elements of $GF(2^n)$ as matrices can be obtained from the powers of a primitive element ('x') in a surprisingly simply way: In the table of powers of a generator

```
n = 4  GF(2^n)
c = 1..11 (polynomial modulus)
  k  :  f:=g**k
  0  :   ...1
  1  :   ..i.
  2  :   .i..
  3  :   i...
  4  :   ..ii
  5  :   .ii.
  6  :   ii..
  7  :   i.ii
  8  :   .i.1
```

```
  n = 4  GF(2^n)
  c = 1..11 (polynomial modulus)
   == 4 1 0  (positions where coefficients are one)
  mm= .1111 (maximal order)
   == 15  =  3 * 5
  h = .1... (aux. bitmask)
  g = ...1. (element of maximal order)
  tv= .1... (traces of x^i)
tr1e= .1... (element with trace=1)

  k   :   f:=g**k  t:=trace(f)  s:=rootof(z^2+z=f)
 ...1 :   f= ...1  t= 0       s=.11.
 ..1. :   f= ..1.  t= 0       s=1.11
 ..11 :   f= .1..  t= 0       s=1..1
 .1.. :   f= 1...  t= 1
 .1.1 :   f= ..11  t= 0       s=11.1
 .11. :   f= .11.  t= 0       s=..1.
 .111 :   f= 11..  t= 1
 1... :   f= 1.11  t= 1
 1..1 :   f= .1.1  t= 0       s=1111
 1.1. :   f= 1.1.  t= 1
 1.11 :   f= .111  t= 0       s=.1..
 11.. :   f= 111.  t= 1
 11.1 :   f= 1111  t= 1
 111. :   f= 11.1  t= 1
 ---- :   f= 1..1  t= 1
```

```
  n = 4  GF(2^n)
  c = 11111 (polynomial modulus) [NON-primitive]
   == 4 3 2 1 0  (positions where coefficients are one)
  mm= .1111 (maximal order)
   == 15  =  3 * 5
  h = .1... (aux. bitmask)
  g = ...11 (element of maximal order)
  tv= .111. (traces of x^i)
tr1e= ...1. (element with trace=1)

  k   :   f:=g**k  t:=trace(f)  s:=rootof(z^2+z=f)
 ...1 :   f= ...1  t= 0       s=11..
 ..1. :   f= ..11  t= 1
 ..11 :   f= .1.1  t= 1
 .1.. :   f= 1111  t= 1
 .1.1 :   f= 111.  t= 1
 .11. :   f= 11.1  t= 0       s=.11.
 .111 :   f= 1...  t= 1
 1... :   f= .111  t= 0       s=1111
 1..1 :   f= 1..1  t= 1
 1.1. :   f= .1..  t= 1
 1.11 :   f= 11..  t= 0       s=1.1.
 11.. :   f= 1.11  t= 0       s=.1.1
 11.1 :   f= ..1.  t= 1
 111. :   f= .11.  t= 0       s=..11
 ---- :   f= 1.1.  t= 0       s=1..1
```

Figure 23.5: Solutions fo the equation $z^2 + z = f$ for all f with trace zero. Both for a primitive (top) and non-primitive (bottom) modulus.

```
 9  :   1.1
10  :   .111
11  :   111.
12  :   1111
13  :   11.1
14  :   1..1
```

take row k through $k+n-1$ to obtain the columns of a matrix M_k. The first few are:

```
M_0 =    M_1 =    M_2 =    M_3 =    M_4 =     M_5 =
1...     ...1     ..1.     .1..     1..1      ..11
.1..     1..1     ..11     .11.     11.1      1.1.
..1.     .1..     1..1     ..11     .11.      11.1
...1     ..1.     .1..     1..1     ..11      .11.
```

Now one has $M_k = M_1^k$ so one can use the matrices to represent the elements of $GF(2^n)$.

The matrix $M := M_1$ is the *companion matrix* of the polynomial modulus g. The companion matrix of a (monic[1]) polynomial $p(x) = x^n - \sum_{i=0}^{n-1} c_i\, x^i$ of degree n is defined as the $n \times n$ matrix

$$M \quad = \quad \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & c_0 \\ 1 & 0 & 0 & \cdots & 0 & c_1 \\ 0 & 1 & 0 & \cdots & 0 & c_2 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & c_{n-2} \\ 0 & 0 & 0 & \cdots & 1 & c_{n-1} \end{bmatrix} \tag{23.18}$$

For polynomials $p(x) = \sum_{i=0}^{n} a_i\, x^i$ set $c_i := -a_i/a_n$.

The *characteristic polynomial* $c(x)$ of a $n \times n$ matrix M is defined as

$$c(x) \quad := \quad \det(x\, E_n - M) \tag{23.19}$$

where E_n is the $n \times n$ unit matrix. The roots of characteristic polynomial are the eigenvalues of the matrix. The characteristic polynomial of the companion matrix of a polynomial $p(x)$ equals $p(x)$. If $p(x)$ is the characteristic polynomial of a matrix M then $p(M) = 0$.

Let $c_k(x)$ be the characteristic polynomial of the matrix $M_k = M^k$ and $p_k(x)$ the minimal polynomial of the element $g^k \in GF(2^n)$. Then

$$c_k(x) \quad = \quad [p_k(x)]^d \quad \text{where} \quad d = n/r \tag{23.20}$$

where r is the smallest positive integer so that $M_k^{2^r} = M_k$ This is a direct consequence of the fact that a matrix is a zero of its own characteristic polynomial.

For example, for the primitive modulus $C(x) = x^4 + x + 1$ (as above) the sequence of characteristic polynomials of the powers of the generator 'x' is

```
  k:    [ p_k(x) ]^d
----------------------
  0:    [    11 ]^4
  1:    [ 11..1 ]^1
  2:    [ 11..1 ]^1
  3:    [ 11111 ]^1
  4:    [ 11..1 ]^1
  5:    [   111 ]^2
  6:    [ 11111 ]^1
  7:    [ 1..11 ]^1
  8:    [ 11..1 ]^1
  9:    [ 11111 ]^1
 10:    [   111 ]^2
 11:    [ 1..11 ]^1
 12:    [ 11111 ]^1
 13:    [ 1..11 ]^1
 14:    [ 1..11 ]^1
```

[1] A monic polynomial has one as leading coefficient.

The trace of the matrix M^k is the d-th power of the *polynomial trace* of the minimal polynomial of g^k. The polynomial trace of $p(x) = x^n - (c_{n-1}\,x^{n-1} + \cdots + c_1\,x + c_0)$ equals c_{n-1} as can be seen from relation 23.18 on the preceding page.

By construction, picking the first column of M_k gives the vector of the coefficients of the polynomial x^k modulo $C(x)$:

$$M^k\,[1,0,0,\ldots,0]^T \quad \equiv \quad x^k \pmod{C(x)} \tag{23.21}$$

Finally, the characteristic polynomial of an element $a \in GF(2^n)$ in polynomial representation can be written as

$$p_a(x) \quad := \quad \prod_{k=0}^{n-1} \left(x - a^{2^k}\right) \tag{23.22}$$

Compare to relation 23.14 on page 610 for minimal polynomials.

23.6 Representation by normal bases

As a vector space over $GF(2)$, we so far used the n basis vectors $x^0, x^1, x^2, x^3, \ldots, x^{n-1}$ to represent an element in $GF(2^n)$. The arithmetic operations were the polynomial operations modulo an irreducible polynomial modulus C.

For certain irreducible polynomials it is possible to use the *normal basis* $x^1, x^2, x^4, x^8, \ldots, x^{2^{n-1}}$ to represent elements of $GF(2^n)$. These polynomials are called *normal polynomials* or *N-polynomials*. To check whether a polynomial C is normal, compute $r_k = x^{2^k} \bmod C$ for $1 \le k \le n$, compute the *null-space*[2] of the matrix M whose k-th row is r_k. If the null-space is empty (that is, $M \cdot v = 0$ implies $v = 0$) then the polynomial is normal.

The normality of a polynomial is equivalent to the fact that its roots are linearly independent (see section 22.4.1 on page 582 for the equivalence of computations modulo a polynomial and computations with linear combinations of its roots).

Addition and subtraction with a normal basis is again a simple XOR. Squaring an element can be achieved by a cyclic shift by one position. Note that $(x^{2^{n-1}})^2 = x^1$. Taking the square root is a cyclic shift in the other direction.

Computation of the trace is particularly easy with normal bases, it equals the parity of the binary word.

Multiplication

Multiplication of two elements is achieved via a *multiplication matrix* M. Given two elements $a, b \in GF(2^n)$ in normal basis representation their product $p = a \cdot b$ is computed as follows: Let a_k be the coefficient (zero or one) of x^{2^k} where $0 \le k \le n-1$. Then, for the first component of the product,

$$p_0 \quad = \quad a^T \cdot M \cdot b \tag{23.23}$$

and in general

$$p_k \quad = \quad \left(a^{-2^{k-1}}\right)^T \cdot M \cdot b^{-2^{k-1}} \tag{23.24}$$

That is, all components of the product are computed like the first but with a and b cyclically shifted.

A routine that checks whether a given polynomial c is normal and, if so, computes the multiplication matrix M, is [FXT: `bitpol_normal_q()` in bpol/bitpolnormal.cc] which proceeds as follows:

[2] The null-space of a matrix M is the subspace spanned by the vectors v_i where $M \cdot v_i = 0$.

- First check whether c is irreducible, and if not, return false.
- Then form a matrix A whose k-th row equals $x^{2^k} \bmod c$. If A is not invertible then (the null-space is not empty and) c is not normal, so return false.
- Now set $D := A \cdot C^T \cdot A^{-1}$ where C is the companion matrix of c.
- Finally, compute the multiplication matrix M where $M_{i,j} := D_{j',i'}$, $i' := -i \bmod n$ and $j' := j - i \bmod n$. Return (true and) the matrix M.

Examples of the intermediate results for two different field polynomials are given in figure 23.6.

```
Normal poly:  c=11111 =^= 4,3,2,1,0

    A=          A^-1=
      .1..        1111
      ..1.        1...
      1111        .1..
      ...1        ...1

    C^T=        D=A*C^T*A^{-1}=
      .1..        .1..
      ..1.        ...1
      ...1        1111
      1111        ..1.

    Multiplication matrix:  M=
      ..1.
      ..11
      11..
      .1.1
```

```
Normal poly:  c=111.11 =^= 5,4,2,1,0

A=            A^-1=
  .1...         11111
  ..1..         1....
  ....1         .1...
  .1.1.         1..1.
  1.111         ..1..

C^T=          D=A*C^T*A^{-1}=
  .1...         .1...
  ..1..         1..1.
  ...1.         ...11
  ....1         .11..
  111.1         ..1.1

Multiplication matrix:  M=
  .1...
  1..1.
  ...11
  .11..
  ..1.1
```

Figure 23.6: Matrices that occur with the computation of the multiplication matrix for the field polynomials $c = 1 + x + x^2 + x^3 + x^4$ (left) and $c = 1 + x + x^2 + x^4 + x^5$ (right).

A C++ function implementing the multiplication algorithm is [FXT: `normal_mult()` in bpol/normalmult.cc]:

```
ulong
normal_mult(ulong a, ulong b, const ulong *M, ulong n)
// Multiply two elements (a and b in GF(2^n))
//  in normal basis representation.
// The multiplication matrix has to be supplied in M.
{
    ulong p = 0;
    for (ulong k=0; k<n; ++k)
    {
        ulong v = bitmat_mult_Mv(M, n, b);
        v = parity( v & a );  // dot product
        p ^= ( v << k );
        a = bit_rotate_right(a, 1, n);
        b = bit_rotate_right(b, 1, n);
    }
    return  p;
}
```

Solving the reduced quadratic equation

The reduced quadratic equation $x^2 + x = f$ has two solutions if $\mathrm{Tr}(f) = 0$. If so, one solution $x = [x_0, x_1, \ldots, x_{n-1}]$ can be obtained as $x_k = \sum_{j=0}^{k} f_k$ where $f = [f_0, f_1, \ldots, f_{n-1}]$. This follows from observing that

$$x^2 + x \quad = \quad [x_0 + x_{n-1},\, x_0 + x_1,\, x_1 + x_2,\, \ldots,\, x_{n-2} + x_{n-1},\, x_{n-1} + x_0] \tag{23.25}$$

equating $x^2 + x = f$ and setting $x_{n-1} = 0$. In C++ this translates to (see page 27) [FXT: file bpol/normalsolvequadratic.cc]

```
ulong
normal_solve_reduced_quadratic(ulong c)
// Solve x^2 + x = c
// Must have:  trace(c)==0
// Return one solution x, the other solution equals 1+x
{
    return  inverse_green_code(c);
}
```

The highest bit of the result is zero exactly if the equation is solvable. The green code is presented in section 1.16.5 on page 37.

In normal basis representation the element one always consists of the all-ones word. Thereby adding one is equivalent to complementing the binary word.

```
arg 1: 5 == n  [The n in GF(2**n)]  default=5
args 2,3,... : [Optionally supply nonzero coefficients of normal poly c]

Normal poly:  c=1.1111
Multiplication matrix:  M=
  ..111
  ...11
  1..1.
  111.1
  11.11

g=....1
  k = bin(k):    f=g**k trace(f) x^2+x==f
  1 = ....1 :    ....1    1
  2 = ...1. :    ...1.    1
  3 = ...11 :    .111.    1
  4 = ..1.. :    ..1..    1
  5 = ..1.1 :    1.111    .       x=.11.1
  6 = ..11. :    111..    1
  7 = ..111 :    .11.1    1
  8 = .1... :    .1...    1
  9 = .1..1 :    111.1    .       x=.1.11
 10 = .1.1. :    .1111    .       x=..1.1
 11 = .1.11 :    ..11.    .       x=...1.
 12 = .11.. :    11..1    1
 13 = .11.1 :    11...    .       x=.1...
 14 = .111. :    11.1.    1
 15 = .1111 :    1.1..    .       x=.11..
 16 = 1.... :    1....    1
 17 = 1...1 :    ..111    1
 18 = 1..1. :    11.11    .       x=.1..1
 19 = 1..11 :    1.11.    1
 20 = 1.1.. :    1111.    .       x=.1.1.
 21 = 1.1.1 :    ...11    .       x=....1
 22 = 1.11. :    .11..    .       x=..1..
 23 = 1.111 :    .1.1.    .       x=..11.
 24 = 11... :    1..11    1
 25 = 11..1 :    .1.11    1
 26 = 11.1. :    1...1    .       x=.1111
 27 = 11.11 :    ..1.1    .       x=...11
 28 = 111.. :    1.1.1    1
 29 = 111.1 :    1..1.    .       x=.111.
 30 = 1111. :    .1..1    .       x=..111
 31 = 11111 :    11111    1
```

Figure 23.7: Powers f of the generator x and their traces in normal basis representation. The rightmost column gives solutions of the equation $x^2 + x = f$ whenever the trace of f equals zero. The (primitive) field polynomial is $1 + x^2 + x^3 + x^4 + x^5$.

The program [FXT: demo gf2n/normalbasis-demo.cc] prints the powers of x in normal basis representation, see figure 23.7. By default a primitive normal polynomial from [FXT: file bpol/highbitnormalprimpoly.cc] is used. If more than one argument is given the arguments should give the nonzero coefficients of the polynomial modulus.

23.7 Conversion between normal and polynomial representation

If the field polynomial C is normal then conversion between the representations in polynomial and normal basis can be achieved as follows: Let Z be the $n \times n$ matrix whose k-th column equals $x^{2^k} \bmod C$ where n is the degree of C. If a is the polynomial representation then the normal representation is $b = Z^{-1} \cdot a$ (both a and b shall be column vectors).

The implementation [FXT: `class GF2n` in bpol/gf2n.h] allows the conversion to the normal representation if the field polynomial is normal. In the initializer [FXT: `GF2n::init()` in bpol/gf2n.cc] the matrix Z (`n2p_tab[]`) and Z^{-1} (`p2n_tab[]`) are computed with the lines

```
    // conversion to and from normal representation:
    for (ulong k=0,s=2;  k<n_;  ++k)
    {
        n2p_tab[k] = s;
        s = bitpolmod_square(s, c_, h_);
    }
    bitmat_transpose(n2p_tab, n_, n2p_tab);
    is_normal_ = bitmat_inverse(n2p_tab, n_, p2n_tab);
```

The last line records whether the field polynomial is normal which is the case exactly if Z is invertible.

The functions [FXT: file bpol/gf2n.cc]

```
ulong // static
GF2n::p2n(ulong f)
{
    return  bitmat_mult_Mv(p2n_tab, n_, f);
}
ulong // static
GF2n::n2p(ulong f)
{
    return  bitmat_mult_Mv(n2p_tab, n_, f);
}
```

allow conversions between the normal and polynomial representations. The method

```
    ulong p2n()  const  { return p2n(v_); }
```

provides a convenient way to obtain the normal representation of a given element.

This is demonstrated in [FXT: demo gf2n/gf2n-normal-demo.cc] where both the polynomial and the normal representation are given, see figure 23.8 on the following page.

If the last argument of the initialization routine of the C++ class `GF2n`, `init(n, c, normalq)`, is set then a (primitive) normal polynomial will be used as field polynomial. A list of primitive normal polynomials is given in [FXT: file bpol/highbitnormalprimpoly.cc].

23.8 Optimal normal bases (ONB)

The number of nonzero terms in the multiplication matrix determine the complexity (operation count) for the multiplication with normal bases. It turns out that for certain values of n there are normal bases of $GF(2^n)$ whose multiplication matrices have at most two nonzero entries in each row (and column). Such bases are called *optimal normal bases* (ONB).

Optimal normal bases are especially interesting for hardware implementations because of both the highly regular structure of the multiplication algorithm and the minimal complexity with ONBs.

```
arg 1: 5 == n  [degree of field]  default=5
args 2,3,... : [Optionally supply nonzero coefficients of field poly c]
  n = 5  GF(2^n)
  c = 1111.1 (polynomial modulus) [normal]
   == 5 4 3 2 0  (positions where coefficients are one)
  mm= .11111 (maximal order)
   == 31 (prime)
  g = ....1. (element of maximal order)
  tv= .11111 (traces of x^i)
tr1e= .....1 (element with trace=1)

P2N=
  11...
  1.11.
  1..11
  1..1.
  1....
N2P=
  ....1
  1...1
  .1.1.
  ...11
  ..11.

   k   :   f:=g**k == (normal)      t:=trace(f)
 ....1 :   f= ...1 == 11111        t= 1
 ...11 :   f= ..1. == ...1.        t= 1
 ..1.1 :   f= .1.. == ..1..        t= 1
 .1..1 :   f= 1... == .1...        t= 1
 1...1 :   f= ...1 == 1111.        t= 0
 ...11 :   f= ..11 == 111..        t= 1
 ..111 :   f= .111 == .11.1        t= 1
 .1..1 :   f= 111. == .1...        t= 1
 1...1 :   f= ..1.1 == 111.1       t= 0
[--snip--]
 11.11 :   f= 1..1. == ..1.1       t= 0
 111.1 :   f= 11..1 == 1.1.1       t= 1
 1111. :   f= .1111 == 1..1.       t= 0
 1111. :   f= 1111. == .1..1       t= 0
```

Figure 23.8: Conversion between normal- and polynomial representation with the (primitive) polynomial $c = 1 + x^2 + x^3 + x^4 + x^5$. The conversion matrices are given as `P2N` and `N2P`.

23.8.1 Type-1 optimal normal bases

A so-called *type-1 optimal normal basis* exists for n when $p := n + 1$ is prime and 2 is a primitive root modulo p. The first such $n \leq 100$ are 2, 4, 10, 12, 18, 28, 36, 52, 58, 60, 66, 82, 100, ... One has always $n \equiv 2$ or $n \equiv 4$ modulo 8. A list of the corresponding primes is given in figure 6.6 on page 225. The field polynomial corresponding to a type-1 ONB is the 'full' polynomial

$$c \quad = \quad 1 + x + x^2 + x^3 + \ldots + x^n \tag{23.26}$$

These polynomials are non-primitive for all $n \neq 2$.

The multiplication matrices are sparse: there is exactly one entry in the first row and column and exactly two in the other rows and columns. That is the multiplication matrices for $GF(n)$ with optimal normal basis have exactly $2n - 1$ nonzero entries. For example, with $n = 10$ one gets the matrix shown (together with the intermediate results) in figure 23.9 on the facing page. The equivalent data for $n = 4$ is shown in figure 23.6 on page 617.

23.8.2 Type-2 optimal normal bases

A *type-2 optimal normal basis* exists for n if $p := 2\,n + 1$ is prime and either

- $n \equiv 1$ or $n \equiv 2$ modulo 4 and the order of 2 modulo p equals $2\,n$.

```
Normal poly:  c=11111111111

A=              A^-1=
  .1........      1111111111
  ..1.......      1.........
  ....1.....      .1........
  ........1.      ........1.
  .....1....      ..1.......
  1111111111      ....1.....
  .........1      .........1
  ........1.      ........1.
  ...1......      ...1......
  ......1...      ......1...

C^T=            D=A*C^T*A^{-1}=     Multiplication matrix:  M=
  .1........      .1........              .....1....
  ..1.......      .......1..              ....11....
  ...1......      ....1.....              ...1...1..
  ....1.....      ......1...              ..1.....1.
  .....1....      .........1              .......1.1
  ......1...      1111111111              1.........
  .......1..      .....1....              .1......1.
  ........1.      ...1......              ..1.1.....
  .........1      ..1.......              ...1..1...
  1111111111      .......1..              ....1....1
```

Figure 23.9: Matrices that occur with the computation of the multiplication matrix for the field polynomial $c = 1 + x + \ldots + x^{10}$.

- $n \equiv 3 \bmod 4$ and the order of 2 modulo p equals n.

A type-2 basis exists for the following $n \leq 100$:

$$2,\ 3,\ 5,\ 6,\ 9,\ 11,\ 14,\ 18,\ 23,\ 26,\ 29,\ 30,\ 33,\ 35,\ 39,\ 41,$$
$$50,\ 51,\ 53,\ 65,\ 69,\ 74,\ 81,\ 83,\ 86,\ 89,\ 90,\ 95,\ 98,\ 99,\ \ldots$$

The corresponding normal polynomial of degree n can be computed via the recursion

$$p_0 := 1 \tag{23.27a}$$
$$p_1 := x + 1 \tag{23.27b}$$
$$p_k := x\,p_{k-1} + p_{k-2} \tag{23.27c}$$

Compare to the recursion that transforms a linear hybrid cellular automaton into a binary polynomial relation 6.5 on page 226: The type-2 ONBs correspond to the most trivial LHCA defined by the rule having a single one at the lowest part of the rule word.

As with type-1 ONBs, the multiplication matrices are sparse. The polynomials and multiplication matrices for $n = 6$ and $n = 9$ are:

```
6,5,4,1,0     9,8,6,5,4,1,0
 1....         .1.......
1...1.        1...1....
..11..        ....1.1..
...1.1        .....1..1
.11...        .11...11.
...1.1        ...1.11..
              ..1.1....
              ..1..1...
              ...1....1
```

The intermediate values with the computation of the multiplication matrix for $n = 5$ is shown in figure 23.6 on page 617.

23.9 Gaussian normal bases

The *type-t Gaussian normal basis* (GNB) generalize the optimal normal basis. The type-1 and type-2 GNBs are the corresponding ONBs. The multiplication matrices for type-t GNBs for $t > 2$ have more nonzero entries than the ONBs.

A type-t GNB exists for n when $p := t\,n + 1$ is prime and $\gcd(n, t\,n/r_2) = 1$ where r_2 is the order of 2 modulo p. Implementation of the test using pari/gp:

```
gauss_test(n, t)=
{ /* test whether a type-t Gaussian normal basis exists for GF(2^n) */
    local( p, r2, g, d );
    p = t*n + 1;
    if ( !isprime(p), return( 0 ) );
    if ( p<=2, return( 0 ) );
    r2 = znorder( Mod(2, p) );
    d = (t*n)/r2;
    g = gcd( d, n);
    return ( if ( 1==g, 1, 0) );
}
```

For n divisible by 8 no GNB exist. If a type-t GNB exists it is unique.

23.9.1 Computation of the multiplication matrix

An algorithm that computes the multiplication matrix for a type-t GNB proceeds as follows (The algorithm uses a vector $F[1, 2, \ldots, p-2]$):

1. Set $p = t\,n + 1$ (this is a prime), and compute an element r of order t modulo p.
2. For $k = 0, 1, \ldots, t-1$ do the following: Set $j = r^k$ and for $i = 0, 1, \ldots, n-1$ set $F[j\,2^i] = i$.
3. Set the multiplication matrix M to zero.
4. For $i = 1, 2, \ldots, p-2$ add one to $M_{F[p-i],F[i+1]}$.
5. If t is odd set $h = n/2$ and do the following: for $i = 0, 1, \ldots, h-1$ increment $M_{i,h+i}$ and $M_{h+i,i}$.

Implementation in pari/gp:

```
gauss_nb(n, t)=
{ /* return multiplier matrix for type-t Gaussian normal basis */
  /* returned matrix is over Z and has to be multiplied by Mod(1,2) */
    local(p, r, F, w, x, nh, m, ir, ic);
    if ( 0==gauss_test(n, t), print("No NB for n=",n,",t=",t); return(0));
    p = t*n + 1;
    r = znprimroot(p);  r = r^(n);  /* r has order t */
    F = vector(p-1);
    w = Mod(1, p);
    for (k=0, t-1,
        j = component(w, 2);
        for (i=0, n-1,
            F[j] = i;
            j+=j;  if (j>=p, j-=p);  /* 2*j mod p */
        );
        w *= r;
    );
    m = matrix(n, n);
    for (i=1, p-2,
        ir = F[p-i];  ic = F[i+1];
        m[ ir+1, ic+1 ] += 1;
    );
    if ( 1==(t%2),
        nh = n/2; /* odd t ==> even n */
        for (i=0, nh-1,
            ir = i;  ic = nh + i;
            ir += 1;  ic += 1;
            m[ir, ic] += 1;
            m[ic, ir] += 1;
        );
    );
    return ( m );
}
```

The implementation computes M with entries in $\mathbb{Z}$ and has to be reduced modulo 2 before usage. For $n = 7$ and $t = 4$ one gets (dots for zeros):

```
  M=          M*Mod(1,2)=
 1..2.       .1...
i.i..ii      i.i..ii
.i.iii.      .i.iii.
..i2.i.      ..i..i.
2.ii..i      .ii..i
.iii.i.      .iii.i.
.i..iii      .i..iii
```

For $n = 12$ and $t = 3$:

```
      M=             M*Mod(1,2)=
...i.1.ii..1.      ...i.1.ii..1.
...i.ii1i....      ...i.ii1i....
..i.i..i2...       ..i.i..i.i...
.ii..i...i..       .ii..i...i..
i.....i.ii.        i.....i.ii.
.i.i...i.1i        .i.i....i.1i
ii..2....i.        ii.i.....i.
.1i.i....i.        .1i.i....i.
..2.....i.1        ...i....i.1
...iii..i...       ...iii..i...
i...i..i...i       i...i..i...i
.....i..i.ii       .....i..i.ii
```

The file [FXT: file data/gauss-normalbasis.txt] lists for each $2 \le n \le 1032$ the smallest ten values of t so that there is a type-t GNB. Note that different values of t do not necessary lead to different multiplication matrices, especially for small values of n. For example, the modulo 2 reduced multiplication matrices for $n = 6$ and the 10 smallest values of t are:

```
 t=2:       t=3:       t=6:       t=10:      t=11:
 1...       ..ii.1     1...       1...       ..ii.i
i...i.      ..ii..     i.ii.i     i.ii.i     ..ii..
..iii.      ii..ii     .i...i     .i...i     ii..ii
..i..i      ii...i     .i.i.i     .i.i.i     ii...i
.ii...      .i.i.i     .i.i.i     .i.i.i     .i.i.i
...i.i      i.i.ii     .ii.ii     .ii.ii     i.i.ii

 t=23:      t=27:      t=30:      t=35:      t=55:
 1.ii.      1.ii.      1...i.     1.ii.      1.ii.
i.iii.      i.iii.     i...i.     i.iii.     i.iii.
.i..i.      .i..i.     ..ii..     .i..i.     .i..i.
ii...i      ii...i     .i..i      ii...i     ii...i
iii..i      iii..i     .ii...     iii..i     iii..i
....ii      ....ii     ...i.i     ....ii     ....ii
```

23.9.2 Determination of the field polynomial

We give two algorithms, the first uses computations with complex numbers, the second uses polynomial modular arithmetic over $GF(2)$.

Algorithm with complex numbers

The normal polynomial corresponding to a type-t Gaussian basis can be computed with the following algorithm.

1. Set $p = t\,n + 1$ and determine r so that the order of r modulo p equals t.
2. For $1 \le k \le n$ compute $w_k = \sum_{j=0}^{t-1} \exp(a_k\,\pi\,i/p)$ where $a_k = 2^k\,r^j$.
3. Let $z(x) = \prod_{k=1}^{n} (x - w_k)$, this is a polynomial with real integer coefficients.
4. Return the polynomial with coefficients reduced modulo 2.

The computation of the polynomial $z(x)$ uses complex (inexact) arithmetics. That the coefficients should be close to real integers can be used as a check.

The following pari/gp routine computes the complex polynomial. In order to keep the arguments for the exponential function small the values a_k are computed modulo $2\,p$, the periodicity of $\exp(.\,\pi\,i/p)$.

```
gauss_zpoly(n, t)=
{ /* return field polynomial for type-t Gaussian normal basis
     as polynomial over the complex numbers */

    local(p, r, wk, tk, rj, a, zp);
```

```
n=4 t=1:  p=5
  a(1)=2  w(1)=(+0.309016994374947 + 0.951056516295153 I)
  a(2)=4  w(2)=(-0.809016994374947 + 0.587785252292473 I)
  a(3)=8  w(3)=(+0.309016994374947 - 0.951056516295153 I)
  a(4)=6  w(4)=(-0.809016994374947 - 0.587785252292473 I)
z(x)=x^4 + x^3 + x^2 + x + 1
p(x)=x^4 + x^3 + x^2 + x + 1

n=4 t=3:  p=13
  a(1)=18  w(1)=(+0.65138781886599 + 0.52241580345640 I)
  a(2)=10  w(2)=(-1.15138781886599 + 1.72542218842200 I)
  a(3)=20  w(3)=(+0.65138781886599 - 0.52241580345640 I)
  a(4)=14  w(4)=(-1.15138781886599 - 1.72542218842200 I)
z(x)=x^4 + x^3 + 2*x^2 - 4*x + 3
p(x)=x^4 + x^3 + 1

n=4 t=7:  p=29
  a(1)=40  w(1)=(+1.09629120178362 - 2.64399848798350 I)
  a(2)=22  w(2)=(-1.59629120178362 - 0.50918758384404 I)
  a(3)=44  w(3)=(+1.09629120178362 + 2.64399848798350 I)
  a(4)=30  w(4)=(-1.59629120178362 + 0.50918758384404 I)
z(x)=x^4 + x^3 + 4*x^2 + 20*x + 23
p(x)=x^4 + x^3 + 1
```

Figure 23.10: Numerical values with the computation of the field polynomial for $n = 4$ and types $t \in \{1, 3, 7\}$. Note that the final result is identical for the types $t = 3$ and $t = 7$.

```
n=11 t=2:  p=23
  a(1)=44   w(1)=(+1.92583457469559 - 5.69206140 E-19 I)
  a(2)=42   w(2)=(+1.70883880909297 - 2.71050543 E-19 I)
  a(3)=38   w(3)=(+0.92013007546230 - 2.98155597 E-19 I)
  a(4)=30   w(4)=(-1.15336064422973 + 2.43945488 E-19 I)
  a(5)=14   w(5)=(-0.66975922434197 + 1.08420217 E-19 I)
  a(6)=28   w(6)=(-1.55142258140884 + 3.25260651 E-19 I)
  a(7)=10   w(7)=(+0.40691202610526 - 6.60685698 E-20 I)
  a(8)=20   w(8)=(-1.83442260301090 + 1.89735380 E-19 I)
  a(9)=40   w(9)=(+1.36510628643730 - 8.13151629 E-20 I)
  a(10)=34  w(10)=(-0.13648482672934 + 3.68459331 E-20 I)
  a(11)=22  w(11)=(-1.98137189207266 + 4.33680868 E-19 I)
z(x)=x^11 + x^10 - 10*x^9 - 9*x^8 + 36*x^7 + 28*x^6 - 56*x^5 \
      - 35*x^4 + 35*x^3 + 15*x^2 - 6*x - 1
p(x)=x^11 + x^10 + x^8 + x^4 + x^3 + x^2 + 1

n=11 t=6:  p=67
  [--snip--]
z(x)=x^11 + x^10 - 30*x^9 - 63*x^8 + 220*x^7 + 698*x^6 - 101*x^5 \
      - 1960*x^4 - 1758*x^3 - 35*x^2 + 243*x - 29
p(x)=x^11 + x^10 + x^8 + x^5 + x^2 + x + 1

n=11 t=8:  p=89
  [--snip--]
z(x)=x^11 + x^10 - 40*x^9 - 19*x^8 + 482*x^7 + 84*x^6 - 2185*x^5 \
      + 102*x^4 + 3152*x^3 - 781*x^2 + 57*x - 1
p(x)=x^11 + x^10 + x^8 + x^5 + x^2 + x + 1

n=11 t=18:  p=199
  [--snip--]
z(x)=x^11 + x^10 - 90*x^9 - 115*x^8 + 2349*x^7 + 943*x^6 - 26327*x^5 \
      + 21284*x^4 + 102168*x^3 - 217794*x^2 + 148930*x - 30647
p(x)=x^11 + x^10 + x^8 + x^7 + x^6 + x^5 + 1
```

Figure 23.11: Numerical values with the computation of the field polynomial for $n = 11$ and types $t \in \{2, 6, 8, 18\}$. The final results are identical for $t = 6$ and $t = 8$.

```
    p = n*t + 1;
    r = znprimroot(p);
    r = r^(n);  /* r has order t */
    r = component(r, 2);
    zp = 1;
    for (k=1, n,
        tk = Mod(2,2*p)^k;
        wk = 0;
        for(j=0, t-1,
            rj = Mod(r,2*p)^j;
            a = rj * tk;
            a = component(a, 2);
            wk += exp(1.0*I*Pi*a/p);
        );
        zp *= (x-wk);
    );
    return ( zp );
}
```

The final step uses pari/gp's function `round()` which rounds all coefficients of its polynomial argument:

```
gauss_poly(n, t)=
{ /* return field polynomial for type-t Gaussian normal basis */
    local(pp, zp);
    zp = gauss_zpoly(n, t);
    pp = round(real(zp));  /* rounds all coefficients */
    pp *= Mod(1,2);  /* coefficients modulo 2 */
    return( pp );
}
```

The results for type-1 bases can be verified using relation 23.26 on page 620, results with type-2 bases with relations 23.27a... 23.27c on page 621. The intermediate values occurring with the computation for $n = 4$ and the types $t \in \{1, 3, 7\}$ are shown in figure 23.10 on the facing page. The values for $n = 11$ and the types $t \in \{2, 6, 8, 18\}$ are shown in figure 23.11 on the preceding page.

Algorithm working in $GF(2)$

The following algorithm is a variation of what is given in [103].

1. Set $p = t\,n + 1$ and determine r so that the order of r modulo p equals t.
2. Set $M(x) = \sum_{k=0}^{p-1} x^k$. All computations are done modulo M.
3. If t equals 1 return M.
4. Set $F_0 = 1$ (modulo M).
5. For $1 \le k \le n$:
   (a) Set $Z_k = \sum_{j=0}^{t-1} x^{a_k}$ (modulo M) where $a_k = 2^k\, r^j \bmod p$.
   (b) Set $F_k = (x + Z_k)\; F_{i-1}$ (modulo M).
6. Return F_n with coefficients reduced modulo 2.

The intermediate quantities in the computation for $n = 4$ and $t = 3$ are shown at the top of figure 23.12 on the following page. The result is a polynomial over the integers identical to the one computed with the algorithm that uses complex numbers. When all polynomials are taken over $GF(2)$ the computation proceeds as shown at the bottom of figure 23.12. Implementation in pari/gp:

```
gauss_poly2(n, t)=
{
    local(p, M, r, F, t2, Z );
    p = t*n + 1;
    r = znprimroot(p)^n;  \\ element of order t mod p
    M = sum(k=0, p-1, 'x^k);  \\ The polynomial modulus
    M *= Mod(1,2);  \\ ... over GF(2)
    F = Mod(1, M);
```

```
  n=4  t=3:  p=13  \\ integer computation

     r= Mod(3, 13)  ord(r)=3 ==  t
     M= x^12 + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1

     ------- k=1
     Z= x^9 + x^3 + x
     F= x^9 + x^3 + 2*x
     ------- k=2
     Z= x^6 + x^5 + x^2
     F= x^11 + x^10 + x^9 + x^8 + 2*x^7 + 2*x^6 + x^5 + x^4 + 2*x^3 + 3*x^2 + x
     ------- k=3
     Z= -x^11 - x^9 - x^8 - x^7 - x^6 - x^5 - x^3 - x^2 - x - 1
     F= -x^11 - x^10 - 2*x^9 + x^7 + 2*x^6 + x^3 - 1
     ------- k=4
     Z= x^11 + x^8 + x^7
     F= x^4 - x^3 + 2*x^2 + 4*x + 3

   ==> x^4 - x^3 + 2*x^2 + 4*x + 3  ==   x^4 + x^3 + 1 (mod 2)
```

```
  n=4  t=3:  p=13  \\ computation over GF(2)

     r= Mod(3, 13)  ord(r)=3 ==  t
     M= x^12 + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1

     ------- k=1
     Z= x^9 + x^3 + x
     F= x^9 + x^3 + 2*x
     ------- k=2
     Z= x^6 + x^5 + x^2
     F= x^11 + x^10 + x^9 + x^8 + x^5 + x^4 + x^2 + x
     ------- k=3
     Z= x^11 + x^9 + x^8 + x^7 + x^6 + x^5 + x^3 + x^2 + x + 1
     F= x^11 + x^10 + x^7 + x^3 + 1
     ------- k=4
     Z= x^11 + x^8 + x^7
     F= x^4 + x^3 + 1

   ==>  x^4 + x^3 + 1
```

Figure 23.12: Computation of the field polynomial for $n = 4$ and $t = 3$ with polynomials over the integers (top) and polynomials over $GF(2)$ (bottom).

```
    for (k=1, n,
        t2 = Mod(2,p)^(k-1);
        Z =  sum(j=0, t-1, Mod('x^(component( t2*r^j, 2)), M) );
        F = ('x+Z)*F;
    );
    F = component(F, 2);
    if ( 0==F,  return( M ) );  \\ for type 1 (t==1)
    return( F );
}
```

While the algorithm avoids inexact arithmetic the polynomial modulus M is of degree $p - 1 = n\,t$ which can be large for large t. In practice the computation with complex numbers is much faster. It finishes in less than 1 second for $n = 620$ and $t = 3$ (and a working precision of 150 decimal digits) while the exact method needs more than 15 minutes.

23.10 The number of binary normal polynomials

The number A_n of degree-n binary normal polynomials up to $n = 40$ is given in figure 23.13 on the next page. A table of the values A_n for $1 \le n \le 130$ and their factorizations is given in [FXT: file data/num-normalpoly.txt]. The sequence A_n is sequence A027362 of [OEIS].

The number B_n of degree-n binary primitive normal polynomials up to $n = 30$ is given in figure 23.14 on the facing page. This is sequence A107222 of [OEIS].

n	A_n	n	A_n	n	A_n	n	A_n
1	1	11	93	21	27783	31	28629151
2	1	12	128	22	95232	32	67108864
3	1	13	315	23	182183	33	97327197
4	2	14	448	24	262144	34	250675200
5	3	15	675	25	629145	35	352149525
6	4	16	2048	26	1290240	36	704643072
7	7	17	3825	27	1835001	37	1857283155
8	16	18	5376	28	3670016	38	3616800768
9	21	19	13797	29	9256395	39	5282242875
10	48	20	24576	30	11059200	40	12884901888

Figure 23.13: The number A_n of degree-n binary normal polynomials up to $n = 40$.

n	B_n	n	B_n	n	B_n
1	1	11	87	21	23579
2	1	12	52	22	59986
3	1	13	315	23	178259
4	1	14	291	24	103680
5	3	15	562	25	607522
6	3	16	1017	26	859849
7	7	17	3825	27	1551227
8	7	18	2870	28	1815045
9	19	19	13797	29	9203747
10	29	20	11255	30	5505966

Figure 23.14: The number B_n of degree-n binary primitive normal polynomials up to $n = 30$.

Brute force computation of A_n and B_n

For small degrees all normal polynomials can be generated by selecting from the irreducible polynomials those that are normal. Using the mechanism that generate all irreducible polynomials via Lyndon words that is described in section 22.6 on page 594, the computation is a matter of minutes for $n < 25$.

The program [FXT: demo gf2n/all-normalpoly-demo.cc] prints all normal polynomials of a given degree n and marks those that are primitive with a '`P`'. The Output for $n = 9$ is

```
 1:  c=11..11..1  P
 2:  c=11...1..11  P
 3:  c=11111..11  P
 4:  c=11.11.1.11  P
 5:  c=111....1.1  P
 6:  c=111...1111  P
 7:  c=11.......1
 8:  c=11.111..11  P
 9:  c=1111.1.1.1  P
10:  c=1111..1.11  P
11:  c=11.1.11.11  P
12:  c=11.1...1..1
13:  c=1111111.11  P
14:  c=1111...111  P
15:  c=11...1.1.1  P
16:  c=11.1...1111  P
17:  c=11...11111  P
18:  c=111.111..1  P
19:  c=1111.11..1  P
20:  c=11..111.11  P
21:  c=11.11.11.1  P
```

We can compute the number of normal (=: A_n) and primitive normal (=: B_n) binary polynomials for a small degrees n using that program. The (simplified) output for $n \le 12$ is:

```
n= 2:    #=   1    #P =   1    #N =   0
n= 3:    #=   1    #P =   1    #N =   0
n= 4:    #=   2    #P =   1    #N =   1
n= 5:    #=   3    #P =   3    #N =   0
n= 6:    #=   4    #P =   3    #N =   1
n= 7:    #=   7    #P =   7    #N =   0
n= 8:    #=  16    #P =   7    #N =   9
n= 9:    #=  21    #P =  19    #N =   2
n=10:    #=  48    #P =  29    #N =  19
```

```
n=11:    #=  93     #P =  87     #N =   6
n=12:    #= 128     #P =  52     #N =  76
```

The table of the values B_n in figure 23.14 was produced with the mentioned program, the computation takes about 90 minutes. As noted in [50], no formula for the number of primitive normal polynomials is presently known.

A_n and cycles in the De Bruijn graph

Quite surprisingly, it turns out that A_n equals the number of cycles in the De Bruijn graph (see sections 7.2.2 on page 237 and 6.3 on page 221). Thereby for n a power of two the number A_n equals the number of binary De Bruijn sequences of length $2\,n$. Apparently no isomorphism between both objects (paths and polynomials) is presently known.

A_n and invertible circulant matrices

Moreover, A_n equals the number of invertible circulant $n \times n$ matrices over $GF(2)$.

```
arg 1: 6 == n  [  n x n - matrices]  default=6
arg 2: 2 == wh  [What to do: 0==>just count  1==>print words
                             2==>also print matrix]  default=2

v0 = 1..... [I]      v0 = 1.11.. [I]
M=                   M=
  1.....               1.11..
  .1....               .1.11.
  ..1...               ..1.11
  ...1..               1..1.1
  ....1.               11..1.
  .....1               .11..1

v0 = 111... [S]      v0 = 11111. [I]      v0 = 11.1.. [I]
M=                   M=                   M=
  111...               11111.               11.1..
  .111..               .11111               .11.1.
  ..111.               1.1111               ..11.1
  ...111               11.111               1..11.
  1...11               111.11               .1..11
  11...1               1111.1               1.1..1

n=6   #invertible=4   #singular=1
```

Figure 23.15: The length-6 Lyndon words of odd weight and the corresponding circulant matrices. Singular matrices are marked with '`[S]`', invertible matrices with '`[I]`'.

This is demonstrated in [FXT: demo gf2n/bitmat-circulant-demo.cc] whose output for $n = 6$ is shown in figure 23.15. The search uses the Lyndon words as periodic words would trivially lead to singular matrices. Further, Lyndon words with an even number of bits can be skipped as the vector $[1, 1, 1, \ldots, 1]$ is in the nullspace of the corresponding matrices.

If the set $\{\alpha, \alpha^2, \alpha^4, \alpha^8, \ldots, \alpha^{2^{n-1}}\}$ is a normal basis of $GF(2^n)$ we say that α generates the normal basis. Considering the rows of a circulant matrix as some element β in a normal basis representation then the following rows are $\beta^2, \beta^4, \beta^8, \ldots, \beta^{2^{n-1}}$ and the matrix is invertible if β generates a normal basis. In fact, if the Lyndon word under consideration is taken as a polynomial $L(x)$ over $GF(2)$ then the corresponding matrix is invertible exactly if $gcd(L(x), x^n - 1) = 1$.

Further, if α generates a normal basis then an element $\beta = \sum_{i=0}^{n-1} a_i\, \alpha^{2^i}$ generates a normal basis exactly if the polynomial $\sum_{i=0}^{n-1} a_i\, x^i$ is relatively prime to $x^n - 1$. Thereby, with a fast algorithm to generate Lyndon words, determine all elements that generate normal bases if one such element is known as follows: Select the Lyndon words with an odd number of ones and test whether $\gcd(x^n - 1, L(x)) = 1$ where $L(x)$ is the polynomial corresponding to the Lyndon word.

If $n = 2^k$ is a power of two, then $x^n - 1 = (x-1)^k$ and all Lyndon words with an odd number of ones are coprime to $x^n - 1$.

A_n and the factorization of $x^n - 1$

The factorization of the polynomial $x^n - 1$ over $GF(2)$ can be used for the computation of A_n. The file [FXT: file data/polfactdeg.txt] supplies the necessary information:

```
# Structure of the factorization of x^n-1 over GF(2):
 1:  [1]  [1*1]
 2:  [2]  [1*1]
 3:  [1]  [1*1 + 1*2]
 4:  [4]  [1*1]
 5:  [1]  [1*1 + 1*4]
 6:  [2]  [1*1 + 1*2]
 7:  [1]  [1*1 + 2*3]
 8:  [8]  [1*1]
 9:  [1]  [1*1 + 1*2 + 1*6]
10:  [2]  [1*1 + 1*4]
11:  [1]  [1*1 + 1*10]
12:  [4]  [1*1 + 1*2]
13:  [1]  [1*1 + 1*12]
14:  [2]  [1*1 + 2*3]
15:  [1]  [1*1 + 1*2 + 3*4]
16: [16]  [1*1]
17:  [1]  [1*1 + 2*8]
```

An entry: `n: [e] [m1*d1 + m2*d2 + ... ]` says that $(x^n - 1) = P(x)^e$ and $P(x)$ factors into `m1` different irreducible polynomials of degree `d1`, `m2` different irreducible polynomials of degree `d2` and so on. As an example, for $n = 6$ we have

$$x^6 - 1 \;=\; \left[x^3 - 1\right]^2 \;=\; \left[(x+1)\left(x^2 + x + 1\right)\right]^2 \tag{23.28}$$

x^6 is the square ($e = 2$) of a product of one irreducible polynomial of degree one and one of degree two. Therefore we have the entry: `6: [2] [1*1 + 1*2]`. Another example, $n = 15$,

$$x^{15} - 1 \;=\; \left[(x+1)\left(x^2 + x + 1\right)\left(x^4 + x + 1\right)\left(x^4 + x^3 + 1\right)\left(x^4 + x^3 + x^2 + x + 1\right)\right]^1 \tag{23.29}$$

corresponding to the entry `15: [1] [1*1 + 1*2 + 3*4]`.

Now one has for the number of normal polynomials:

$$A_n \;=\; \frac{2^n}{n} \prod_i \left(1 - \frac{1}{2^{d_i}}\right)^{m_i} \tag{23.30}$$

Note that the quantity e does not appear in the formula.

For example, with $n = 6$

$$A_6 \;=\; \frac{2^6}{6} \cdot \left(1 - \frac{1}{2^1}\right)^1 \cdot \left(1 - \frac{1}{2^2}\right)^1 \;=\; \frac{64}{6} \cdot \frac{1}{2} \cdot \frac{3}{4} \;=\; 4 \tag{23.31}$$

With $n = 15$:

$$A_{15} \;=\; \frac{2^{15}}{15} \cdot \left(1 - \frac{1}{2^1}\right)^1 \cdot \left(1 - \frac{1}{2^2}\right)^1 \cdot \left(1 - \frac{1}{2^4}\right)^3 \;=\; 675 \tag{23.32}$$

Efficient computation of A_n

It is actually possible to compute the number of degree-n normal binary polynomials without explicitly factorizing $x^n - 1$. We have

$$x^n - 1 \;=\; \prod_{d \backslash n} Y_d(x) \tag{23.33}$$

where $Y_d(x)$ is is the d-th cyclotomic polynomial (see section 22.7 on page 596). We further know that $Y_d(x)$ factors into $\varphi(d)/r$ polynomials of degree r where $r = \mathrm{ord}_d(2)$ is the order of 2 modulo d. Let $a_n := A_n / \left(\frac{2^n}{n}\right)$, then a_n can for odd n be computed as

$$a_n = \prod_{d \backslash n} \left(1 - \frac{1}{2^r}\right)^{\varphi(d)/r} \tag{23.34}$$

Using pari/gp:

```
p=2 /* global */
nnodd(n)=
{
    local( r, i, pp );
    pp = 1;
    fordiv (n, d,
        r = znorder(Mod(p,d));
        i = eulerphi(d)/r;
        pp *= (1 - 1/p^r)^i;
    );
    return( pp );
}
```

The number A_n can be computed (for arbitrary n) as $A_n = a_q \left(\frac{2^n}{n}\right)$ where q odd, and $n = q\,2^t$:

```
numnormal(n)=
{
    local( t, q, pp );
    t = 1;  q = n;
    while ( 0==(q%p), q/=p; t+=1; );
    /* here: n==q*p^t */
    pp = nnodd(q);
    pp *= p^n/n;
    return( pp );
}
```

The quantity t is not used in the computation. The implementation is quite efficient: the computation of A_n for all $n \le 10,000$ takes less than three seconds. The computation of A_n for $n = 1234567 = 127 \cdot 9721$ (A_n is a number with 371,636 decimal digits) takes about 200 milliseconds.

23.11 A construction for normal polynomials of degree 2^n *

Gao [GaoNB], p.63, gives a method to construct normal polynomials of degree 2^n. The polynomials a_k, b_k and p_k are computed as

$$a_{k+1} = a_k\, b_k \tag{23.35a}$$
$$b_{k+1} = a_k^2 + b_k^2 \;=\; (a_k + b_k)^2 \tag{23.35b}$$
$$p_k = a_k + b_k \tag{23.35c}$$

where $a_0 = x$ and $b_0 = 1$. The polynomial p_k is a normal polynomial of degree 2^k. The first such polynomials $p_k(x)$ and $q_k(x) := p_k(x+1)$ are

```
k=1,  deg=2:
p(x)= 111
q(x)= 111
k=2,  deg=4:
p(x)= 11111
q(x)= 11..1  // == x^4 + x^3 + 1
k=3,  deg=8:
p(x)= 111.1.111
q(x)= 11.1111.1
k=4,  deg=16:
p(x)= 111111..1..111111
```

```
q(x)= 11....11.1.1.11.1
k=5,  deg=32:
p(x)= 111.111.1.1....111....1.1.111.111
q(x)= 11.11....111..111.1..1.11.11.11.1
k=6,  deg=64:
p(x)= 11111...1.1.11..11..11.1.1...1111111...1.1.11..11..11.1.1...11111
q(x)= 11...1..111....11.11.1.1111.1.11....1..111....11.11.1.1111.1.11.1
```

The polynomials q_k are also normal. Moreover, for all $k \leq 8$, the order of p_{k-1} modulo p_k is maximal (it equals $2^k - 1$). Thereby the minimal polynomials of p_{k-1} modulo p_k are primitive. Their reciprocals are also normal, the first few are

```
111
11..1
111....11
11..11.11.1111..1
111..11111.11111..1.11...11...111
11..111..11.1.......111.1.....11.1.11.1.......1....1.111.11...1.1
```

Chapter 24

Synthetic Iterations *

It is easy to construct arbitrary many iterations that converge super-linearly. Guided by some special constants that in base 2 can be obtained by recursive constructions we build iterations that allow the computation of the constant in a base independent manner. The iterations lead to functions that typically cannot be identified in terms of known (named) functions. Some of the functions can be expressed as infinite sums or products.

For the constructions in terms of repeated string substitutions see section 5.21.

24.1 A variation of the iteration for the inverse

We start with the product form for the most simple iteration, the one for the inverse:

$$\begin{aligned} I(y) &:= \frac{1}{1-y} && (24.1a)\\ &= 1 + y + y^2 + y^3 + y^4 + \ldots && (24.1b)\\ &= (1+y)\,(1+y^2)\,(1+y^4)\,(1+y^8)\,\ldots\,(1+y^{2^k})\,\ldots && (24.1c)\\ &= (1+Y_0)\,(1+Y_1)\,(1+Y_2)\,(1+Y_3)\,\ldots\,(1+Y_k)\,\ldots && (24.1d)\\ &\quad \text{where} \quad Y_0 = y,\ Y_{k+1} = Y_k^2 \end{aligned}$$

We now modify the signs in the infinite product:

$$\begin{aligned} J(y) &:= (1-y)\,(1-y^2)\,(1-y^4)\,(1-y^8)\,\ldots\,(1-y^{2^k})\,\ldots && (24.2a)\\ &= 1 - y - y^2 + y^3 - y^4 + y^5 + y^6 - y^7 - y^8 \pm \ldots && (24.2b)\\ &= (1-Y_0)\,(1-Y_1)\,(1-Y_2)\,(1-Y_3)\,\ldots\,(1-Y_k)\,\ldots && (24.2c) \end{aligned}$$

The sign of the n-th coefficient equals plus one if the parity of n is zero, else minus one (sequence A106400 of [OEIS], the *Thue-Morse sequence*). The related *parity number* (or *Prouhet-Thue-Morse constant*) is

$$\begin{aligned} P &= 0.41245403364010759778336136825845528308947837445576955 75\ldots && (24.3)\\ \text{[base 2]} &= 0.0110, 1001, 1001, 0110, 1001, 0110, 0110, 1001, 1001, 0110, 0110, 1001, \ldots\\ \text{[base 16]} &= 0.6996, 9669, 9669, 6996, 9669, 6996, 6996, 9669, 9669, 6996, 6996, 9669, \ldots\\ \text{[CF]} &= [0, 2, 2, 2, 1, 4, 3, 5, 2, 1, 4, 2, 1, 5, 44, 1, 4, 1, 2, 4, 1, 1, 1, 5, 14, 1, 50, 15, 5, 1, 1, 1, 4, 2, 1, \ldots] \end{aligned}$$

The sequence of zeros and ones in the binary expansions is sequence A010060 of [OEIS]. The constant P can be computed defining

$$\begin{aligned} K(y) &= \frac{[I(y) - J(y)]}{2} \\ &= y + y^2 + y^4 + y^7 + y^8 + y^{11} + y^{13} + y^{14} + y^{16} + y^{19} + y^{21} + y^{22} + \dots \end{aligned} \tag{24.4}$$

Then

$$P = \frac{1}{2} K\left(\frac{1}{2}\right) = \frac{1}{2} \frac{\left[I(\frac{1}{2}) - J(\frac{1}{2})\right]}{2} = \frac{1}{2} - \frac{1}{4} J\left(\frac{1}{2}\right) \tag{24.5}$$

Thereby (see also [HAKMEM], item 125),

$$2 - 4\,P = J\left(\frac{1}{2}\right) = \prod_{n=0}^{\infty} \left(1 - \frac{1}{2^{2^n}}\right) = 0.350183865439569608866554526966178\dots \tag{24.6}$$

The sequence of bits of the parity number can also be obtained by starting with a single 0 and repeated application of the substitution rules $0 \to 01$ and $1 \to 10$.

The following relations follow directly from the definitions of the functions $I(.)$ and $J(.)$:

$$I(y) = (1 + y)\, I(y^2) \tag{24.7a}$$

$$J(y) = (1 - y)\, J(y^2) \tag{24.7b}$$

$$I(-y) = \frac{1 - y}{1 + y} I(y) \tag{24.7c}$$

$$J(-y) = \frac{1 + y}{1 - y} J(y) \tag{24.7d}$$

One finds

$$I(y) = 1 + \sum_{k=0}^{\infty} \left[y^{2^k} \prod_{j=0}^{k-1} \left(1 + y^{2^j}\right) \right] \tag{24.8a}$$

$$J(y) = 1 - \sum_{k=0}^{\infty} \left[y^{2^k} \prod_{j=0}^{k-1} \left(1 - y^{2^j}\right) \right] \tag{24.8b}$$

A functional equation for $K(.)$ is given by

$$K(y) = (1 - y)\, K(y^2) + \frac{y}{1 - y^2} \tag{24.9}$$

It is solved by the series

$$K(y) = \sum_{k=0}^{\infty} \left[\frac{y^{2^k}}{1 - y^{2^{k+1}}} \prod_{j=0}^{k-1} \left(1 - y^{2^j}\right) \right] \tag{24.10}$$

For the inverse of $J(.)$ one finds

$$\frac{1}{J(y)} = 1 + y + 2y^2 + 2y^3 + 4y^4 + 4y^5 + 6y^6 + 6y^7 + 10y^8 + 10y^9 + 14y^{10} + \dots \tag{24.11a}$$

$$= \left[(1 - y)\,(1 - y^2)\,(1 - y^4)\,(1 - y^8)\,\dots\right]^{-1} = \prod_{n=0}^{\infty} I(y^{2^n}) \tag{24.11b}$$

$$= (1 - y) \prod_{n=0}^{\infty} \frac{1 + y^{2^n}}{1 - y^{2^n}} \tag{24.11c}$$

$$= (1 + y)\,(1 + y^2)^2\,(1 + y^4)^3\,(1 + y^8)^4\,\dots\,(1 + y^{2^k})^{k+1}\,\dots \tag{24.11d}$$

The sequence of coefficients of the even exponents in relation 24.11a

```
1, 2, 4, 6, 10, 14, 20, 26, 36, 46, 60, 74, 94, 114, 140, 166, 202, ...
```

is sequence A000123 of [OEIS], the number of binary partitions of the even numbers. The sequence $\frac{1}{2}\,[2, 4, 6, 10, 14, \ldots]$ modulo two equals the period-doubling sequence, see section 24.5 on page 641.

Series for $I(y)$ are

$$I(y) \;=\; \sum_{k=0}^{\infty} \frac{y^{2^k-1}}{1-y^{2^{k+1}}} \;=\; \sum_{k=0}^{\infty} \frac{2^k\, y^{2^k-1}}{1+y^{2^k}} \tag{24.12a}$$

Integration gives (compare with relation 15.28 on page 403)

$$-\log(1-y) \;=\; \sum_{k=0}^{\infty} \frac{1}{2^{k+1}}\, \log\left(\frac{1+y^{2^k}}{1-y^{2^k}}\right) \;=\; \sum_{k=0}^{\infty} \log\left(1+y^{2^k}\right) \tag{24.13a}$$

The derivative of J can be computed via

$$J'(y) \;=\; -J(y) \sum_{k=0}^{\infty} \frac{2^k\, y^{2^k-1}}{1-y^{2^k}} \tag{24.14}$$

The Komornik-Loreti constant

One has $K(1/\beta) = 1$ for

$$\begin{aligned}
\frac{1}{\beta} &= 0.55952455849672652513220976515743228583107647896866030 76\ldots && (24.15a)\\
[\text{base } 2] &= 0.10001111001111010000000001100000000011010110001000101 10\ldots \\
\beta &= 1.78723165018296593301327489033700838533793140296181099 7\ldots && (24.15b)\\
[\text{base } 2] &= 1.11001001100010000000001101101111111010010111010110100 00\ldots \\
[\text{CF}] &= [1, 1, 3, 1, 2, 3, 188, 1, 12, 1, 1, 22, 33, 1, 10, 1, 1, 7, 1, 9, 1, 1, 20, 2, 15, 1, \ldots]
\end{aligned}$$

The constant β is the smallest real number in the interval $(1, 2)$ so that 1 has a unique expansion of the form $\sum_{n=1}^{\infty} \delta_n\, \beta^{-n}$ where $\delta_n \in \{0, 1\}$. It is called the *Komornik-Loreti constant* (see [2]). That in fact $\delta_n = 1$ exactly where the Thue-Morse sequence equals 1 was used for the computation of β: one solves $K(y) = 1$ for y. The transcendence of β is proved (using the fact that $J(y)$ is transcendental for algebraic y) in [3].

Third order variants

Variations of the third order iteration for the inverse

$$\begin{aligned}
I(y) &:= \frac{1}{1-y} && (24.16a)\\
&= 1 + y + y^2 + y^3 + y^4 + \ldots && (24.16b)\\
&= (1 + y + y^2)\,(1 + y^3 + y^6)\,(1 + y^9 + y^{18}) \ldots (1 + y^{3^k} + y^{2\cdot 3^k}) \ldots && (24.16c)\\
&= (1 + Y_0 + Y_0^2)\,(1 + Y_1 + Y_1^2)\,(1 + Y_2 + Y_2^2) \ldots (1 + Y_k + Y_k^2) \ldots && (24.16d)\\
&\qquad \text{where} \quad Y_0 = y,\; Y_{k+1} = Y_k^3
\end{aligned}$$

lead to series related to the base-3 analogue of the parity. The obvious generalization $\prod_{n=0}^{\infty} (1 + \omega\, Y_n + \omega^2\, Y_n^2)$ where $\omega^3 = 1$ involves computations with complex numbers. The most simple example with real numbers might be

$$\begin{aligned}
T(y) &= (1 + Y_0 - Y_0^2)\,(1 + Y_1 - Y_1^2)\,(1 + Y_2 - Y_2^2) \ldots (1 + Y_k - Y_k^2) \ldots && (24.17a)\\
&= 1 + y - y^2 + y^3 + y^4 - y^5 - y^6 - y^7 + y^8 + y^9 + y^{10} - y^{11} + y^{12} \pm \ldots && (24.17b)
\end{aligned}$$

The sign of the n-th coefficient is the parity of the number of twos in the radix-3 expansion of n. One has

$$\begin{aligned}
\frac{1}{2}\,(I(y)-T(y)) &= y^2+y^5+y^6+y^7+y^{11}+y^{14}+y^{15}+y^{16}+y^{18}+y^{19}+y^{21}+\ldots && (24.18a)\\
\frac{1}{4}\left(I\left(\frac{1}{2}\right)-T\left(\frac{1}{2}\right)\right) &= 0.152644523625407582531924921475791679311504514871489254 8\ldots && (24.18b)\\
\text{[base 2]} &= 0.00100111000100111011011000100100111000100111011011000 11\ldots\\
\text{[CF]} &= [0,6,1,1,4,2,1,1,2,4,1,1,4,1,4,2,1,1,1,2,1,18,3,24,1,6,1,3,\ldots]
\end{aligned}$$

The sequence of zeros and ones in the binary expansion is entry A064990 of [OEIS], the *Mephisto Waltz sequence*.

24.2 An iteration related to the Thue constant

We construct a sequence of zeros and ones that can be generated by starting with a single 0 and repeated application of the substitution rules $0 \to 111$ and $1 \to 110$. The evolution starting with a single zero is:

```
T0 =  0
T1 =  111
T2 =  110110110
   == 3 times 11.0
T3 =  110110111110110111110110111
   == 3 times 110110.111
T4 =  11011011111011011111011011011011011111011011111011011011011111011011111011011011
   == 3 times 110110111110110111.110110110
T --> 11011011111011011111011011011011011111011011111011011011011111011011111011011111...
```

The crucial observation is that is T_n is three time repeated the string $T'_{n-1}.T_{n-2}$ where T'_k consists of the first and second third of T_k. The length of the n-th string is 3^n. Let $T(y)$ be the function whose power series corresponds to the string T_∞:

$$T(y) = 1+y+y^3+y^4+y^6+y^7+y^8+y^9+y^{10}+y^{12}+y^{13}+y^{15}+y^{16}+y^{17}+y^{18}+\ldots \quad (24.19)$$

It can be computed by the iteration

$$\begin{aligned}
L_0 &= 0 && (24.20a)\\
A_0 &= 1+y && (24.20b)\\
B_0 &= y^2 && (24.20c)\\
Y_0 &= y && (24.20d)\\
R_n &= A_n + y^2\,L_n && (24.20e)\\
L_{n+1} &= A_n + B_n && (24.20f)\\
Y_{n+1} &= Y_n^3 && (24.20g)\\
A_{n+1} &= R_n\,(1+Y_{n+1}) \quad \to T(y) && (24.20h)\\
B_{n+1} &= R_n\,Y_{n+1}^2 && (24.20i)
\end{aligned}$$

The implementation is slightly tricky:

```
th(y, N=5)=
{
    local(L, R, A, B, y2, y3, t);
    /* correct up to order 3^(N+1)-1 */
    L=0;
    A=1+y;  B=y^2; /* R = A.B */
    for(k=1, N,
      /* (L, A.B) --> (A.B,  A.L.A.L . A.L) */
      y2 = y^2;
      R = A + y2*L; /* A.L */
      L = A + B;   /* next L = A.B */
      y3 = y * y2;
```

```
      B = R * (y3*y3);
      A = R * (1+y3);    /* next A = A.L.A.L  */
      y = y3;
    );
    return( A + B )
}
```

The *Thue constant* (which should be *Roth's constant*, see entry. A014578 of [OEIS]) can be computed as

$$\begin{aligned}
\frac{1}{2}\,T(\frac{1}{2}) &= 0.859099796854703104903572502841974202614239955559439087 4\ldots && (24.21)\\
[\text{base } 2] &= 0.110, 110, 111, 110, 110, 111, 110, 110, 110, 110, 110, 111, 110, 110, 111, 110, 110, \ldots\\
[\text{base } 8] &= 0.667, 667, 666, 667, 667, 666, 667, 667, 667, 667, 667, 666, 667, 667, 666, 667, 667, \ldots\\
[\text{CF}] &= [0, 1, 6, 10, 3, 2, 513, 1, 1, 2, 1, 4, 2, 6576668769, 1, 1, 4,\\
&\quad 1, 2, 2, 256, 1, 1, 2, 1, 2, 3, 1, 3, 3, 241785163922925834941235 3,\\
&\quad 1, 2, 3, 1, 3, 2, 1, 2, 1, 1, 256, 2, 2, 1, 4, 2, 3288334384,\\
&\quad 1, 1, 4, 1, 2, 2, 146, 2, 3, 3, 2, 1, 2, 1, 12, X, \ldots]
\end{aligned}$$

The term X in the continued fraction has 74 decimal digits. By construction the bits at positions n not divisible by three are one and otherwise the complement of the bit at position $n/3$. As a functional equation (see also section 24.5 on page 641):

$$y\,T(y) + y^3\,T(y^3) = \frac{y}{1-y} \qquad (24.22)$$

From this relation we can obtain a series for $T(y)$:

$$T(y) = \sum_{k=0}^{\infty} (-1)^n\,\frac{y^{3^n-1}}{1-y^{3^n}} \qquad (24.23)$$

24.3 An iteration related to the Golay-Rudin-Shapiro sequence

Define $Q(y)$ by

$$\begin{aligned}
L_0 &= 1 && (24.24a)\\
R_0 &= y && (24.24b)\\
Y_0 &= y && (24.24c)\\
L_{n+1} &= L_n + R_n \quad \to Q(y) && (24.24d)\\
Y_{n+1} &= Y_n^2 && (24.24e)\\
R_{n+1} &= Y_{n+1}\,(L_n - R_n) && (24.24f)
\end{aligned}$$

then

$$Q(y) = 1 + y + y^2 - y^3 + y^4 + y^5 - y^6 + y^7 + y^8 + y^9 + y^{10} - y^{11} - y^{12} - y^{13} + y^{14} - y^{15} + \ldots \qquad (24.25)$$

The sequence of coefficients is the *Golay-Rudin-Shapiro sequence* (or *GRS sequence*, entry A020985 of [OEIS], see also page 37). The constant

$$\begin{aligned}
Q &= 0.92924386959737885325397664472205076441287553952432552 22\ldots && (24.26)\\
[\text{base } 2] &= 0.1110, 1101, 1110, 0010, 1110, 1101, 0001, 1101, 1110, 1101, 1110, 0010, \ldots\\
[\text{base } 16] &= 0.ede2, ed1d, ede2, 12e2, ede2, ed1d, 121d, ed1d, ede2, ed1d, ede2, 12e2, \ldots\\
[\text{CF}] &= [0, 1, 13, 7, 1, 1, 15, 4, 1, 3, 1, 2, 2, 1000, 12, 2, 1, 6, 1, 1, 1, 1, 1, 1, 1, 8, 2, 1, 1, 2, 4, 1, 1, 3, \ldots]
\end{aligned}$$

can be computed as

$$Q = \frac{1 + \frac{1}{2} Q(\frac{1}{2})}{2} \tag{24.27}$$

The implementation using pari/gp:

```
qq(y, N=8)=
{
    local(L, R, Lp, Rp);
    /* correct up to order 2**(N+1) */
    L=1;  R=y;
    for(k=0,N, Lp=L+R; y*=y; Rp=y*(L-R); L=Lp; R=Rp);
    return( L + R )
}
```

One has

$$Q(y^2) = \frac{Q(y) + Q(-y)}{2} \tag{24.28}$$

and

$$Q\left(\frac{1}{q}\right) = Q\left(\frac{1}{q^2}\right) + \frac{1}{q} Q\left(-\frac{1}{q^2}\right) \tag{24.29a}$$

$$Q\left(-\frac{1}{q}\right) = Q\left(\frac{1}{q^2}\right) - \frac{1}{q} Q\left(-\frac{1}{q^2}\right) \tag{24.29b}$$

Combining the latter two and setting $y = \frac{1}{q}$ gives

$$Q(y) = (1+y)\, Q\left(y^4\right) + \left(y^2 - y^3\right)\, Q\left(-y^4\right) \tag{24.30}$$

Counting zeros and ones in the binary expansion of Q

The number of ones and zeros in the first 4^k bits of the constant Q can be computed via a string-substitution engine (see section 5.21 on page 206). The hexadecimal expansion can be obtained as:

```
    Number of symbols = 4
    Start: e
    Rules:
      e --> ed
      d --> e2
      2 --> 1d
      1 --> 12
    -------------
    0:   (#=1)
      e
    1:   (#=2)
      ed
    2:   (#=4)
      ede2
    3:   (#=8)
      ede2ed1d
    4:   (#=16)
      ede2ed1dede212e2
    5:   (#=32)
      ede2ed1dede212e2ede2ed1d121ded1d
    6:   (#=64)
      ede2ed1dede212e2ede2ed1d121ded1dede2ed1dede212e2121d12e2ede212e2
```

A few lines of pari/gp code to count the occurrences of the symbols (and thereby of zeros and ones):

```
/*   e --> ed  ;  d --> e2  ;  2 --> 1d  ;  1 --> 12    */
/*   e  d      ;  e     2   ;     d     1;        2  1; */
mg= [1, 1, 0, 0;  1, 0, 1, 0;  0, 1, 0, 1;  0, 0, 1, 1];
mg=mattranspose(mg)
{ for (k=0, 40,
    print1( k, ":  " );
    mm=mg^k;
    mv = mm*[1,0,0,0]~;
    t =  sum(i=1,4, mv[i]);
    /* e and d have three ones and one zero */
```

```
    /* 1 and 2 have one one and three zeros */
    n0 = 3*(mv[3]+mv[4]) + (mv[1]+mv[1]); /* # of zeros */
    n1 = 3*(mv[1]+mv[2]) + (mv[3]+mv[4]); /* # of ones  */
    print( t, "   ",  mv~,
     "   #0=",  n0, " #1=", n1, "  diff=", n1-n0, "  #1/#0=", 1.0*n1/n0 );
 ) }
```

```
                #e   #d   #2   #1
   0:      1   [  1,   0,   0,   0]   #0=   2   #1=   3  diff= 1  #1/#0=1.5000000
   1:      2   [  1,   1,   0,   0]   #0=   2   #1=   6  diff= 4  #1/#0=3.0000000
   2:      4   [  2,   1,   1,   0]   #0=   7   #1=  10  diff= 3  #1/#0=1.4285714
   3:      8   [  3,   3,   1,   1]   #0=  12   #1=  20  diff= 8  #1/#0=1.6666666
   4:     16   [  6,   4,   4,   2]   #0=  30   #1=  36  diff= 6  #1/#0=1.2000000
   5:     32   [ 10,  10,   6,   6]   #0=  56   #1=  72  diff=16  #1/#0=1.2857142
   6:     64   [ 20,  16,  16,  12]   #0= 124   #1= 136  diff=12  #1/#0=1.0967741
   7:    128   [ 36,  36,  28,  28]   #0= 240   #1= 272  diff=32  #1/#0=1.1333333
   8:    256   [ 72,  64,  64,  56]   #0= 504   #1= 528  diff=24  #1/#0=1.0476190
   9:    512   [136, 136, 120, 120]   #0= 992   #1=1056  diff=64  #1/#0=1.0645161
  10:   1024   [272, 256, 256, 240]   #0=2032   #1=2080  diff=48  #1/#0=1.0236220
    [--snip--]
  40: 1099511627776   [274878431232, 274877906944, 274877906944, 274877382656]  \
         #0=2199022731264 #1=2199024304128  diff=1572864  #1/#0=1.00000071525
```

Figure 24.1: Number of symbols, zeros and ones with the n-th step of the string substitution engine for the GRS sequence. For long strings the ratio of the number of zeros and ones approaches one.

We obtain the data shown in figure 24.1. The sequence of the numbers of ones is entry A005418 of [OEIS]. It is identical to the sequence of numbers of equivalence classes obtained by identifying bit-strings that are mutual reverses or complements, see section 3.8, page 124.

24.4 Iterations related to the ruler function

The ruler function $r(n)$ can be defined to be the highest exponent e so that 2^e divides n. Here we consider the function that equals $r(n)+1$ for $n \neq 0$ and zero for $n = 0$. The partial sequences up to indices $2^n - 1$ are

```
R0 = 0
R1 = 0 1
R2 = 0 1 2 1
R3 = 0 1 2 1 3 1 2 1
R4 = 0 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
R5 = 0 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
R6 = 0 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 6 1 ...
```

Observe that $R_n = R_{n-1}.(R_{n-1} + [n, 0, 0, \ldots, 0])$.

Define the function $R(y)$ by

$$R_1 = y \qquad Y_1 = y \tag{24.31a}$$

$$Y_{n+1} = Y_n^2 \tag{24.31b}$$

$$R_{n+1} = R_n + Y_{n+1}\,[R_n + (1+n)] \quad \rightarrow R(y) \tag{24.31c}$$

Implementation in pari/gp:

```
r2(y, N=11)=
{ /* correct to order = 2^N-1 */
    local(A);
    A=y;
    for(k=2, N,  y *= y;  A += y*(A + k); );
    return( A );
}
```

If one replaces in the statement `A += y*(A + k);` by `A += y*(A + 1);` then the iteration computes $\frac{y}{1-y}$.

One finds for the function $R(.)$:

$$R\left(\frac{1}{q^2}\right) = \frac{1}{2q}\left[(q-1)\,R\left(\frac{1}{q}\right) + (q+1)\,R\left(-\frac{1}{q}\right)\right] \tag{24.32a}$$

$$R\left(\frac{1}{q}\right) = R\left(\frac{1}{q^2}\right) + \frac{1}{q-1} \tag{24.32b}$$

$$R(y) = R(y^2) + \frac{y}{1-y} = R(y^4) + \frac{y}{1-y} + \frac{y^2}{1-y^2} = \ldots \tag{24.32c}$$

and so

$$R(y) = \sum_{n=0}^{\infty} \frac{y^{2^n}}{1-y^{2^n}} \tag{24.33}$$

One further has

$$R(y) = \sum_{n=0}^{\infty} (1+n)\,\frac{y^{2^n}}{1-y^{2^{n+1}}} \tag{24.34}$$

Define the *ruler constant* as $R := R(1/2)/2$, then

$$\begin{aligned} R &= 0.7019684139410891602881030370686046772688193807609450337\ldots \\ [\text{base } 2] &= 0.10110011101101000011001110110100101100111011010000110011\ldots \\ [\text{CF}] &= [0,1,2,2,1,4,2,1,1,1,2,2,1,3,3,4,5,6,1,5,1,1,9,49,1,8,1,1,5,1, \\ &\quad 6,5,1,3,3,1,2,4,3,1,2,4,2,1,1,3,1,9,1,11,18,2,4,5,1,3,2,25,9, \\ &\quad 2,3,1,2,3,1,9,1,2,8,1,3,4,1,1,1,1,31,1,1,6,1,13,1,1,14,1,6,1,\ldots] \end{aligned} \tag{24.35}$$

A similar sum

We now want to compute

$$P(y) = \sum_{n=0}^{\infty} \frac{y^{2^n}}{1+y^{2^n}} = \sum_{n=0}^{\infty} (1-n)\,\frac{y^{2^n}}{1-y^{2^{n+1}}} \tag{24.36}$$

The partial sequences of coefficients of the Taylor expansion are

```
P0 = 0
P1 = 0 1
P2 = 0 1 0 1
P3 = 0 1 0 1 -1 1 0 1
P4 = 0 1 0 1 -1 1 0 1 -2 1 0 1 -1 1 0 1
P5 = 0 1 0 1 -1 1 0 1 -2 1 0 1 -1 1 0 1 -3 1 0 1 -1 1 0 1 -2 1 0 1 -1 1 0 1
P6 = 0 1 0 1 -1 1 0 1 -2 1 0 1 -1 1 0 1 -3 1 0 1 -1 1 0 1 -2 1 0 1 -1 1 0 1 -4 1...
```

Observe that $P_n = P_{n-1}.(P_{n-1} - [n-2, 0, 0, \ldots, 0])$.

Compute $P(y)$ by

$$P_1 = y \qquad Y_1 = y \tag{24.37a}$$

$$Y_{n+1} = Y_n^2 \tag{24.37b}$$

$$P_{n+1} = P_n + Y_{n+1}\,[P_n + (1-n)] \quad \to P(y) \tag{24.37c}$$

Implementation in pari/gp:

```
p2(y, N=11)=
{ /* correct to order = 2^N-1 */
    local(A);
    A=y;
    for(k=2, N,  y *= y;  A += y*(A - (k-2)); );
    return( A );
}
```

One finds for $P(.)$:

$$P\left(\frac{1}{q^2}\right) = \frac{1}{2q}\left[(q+1)\,P\left(\frac{1}{q}\right) + (q-1)\,P\left(-\frac{1}{q}\right)\right] \tag{24.38a}$$

$$P\left(\frac{1}{q}\right) = P\left(\frac{1}{q^2}\right) + \frac{1}{q+1} \tag{24.38b}$$

Relations that involve both $P(.)$ and $R(.)$ are

$$P\left(\frac{1}{q^2}\right) = \frac{1}{2}\left[\frac{q+3}{q+1}\,P\left(\frac{1}{q}\right) - \frac{q-1}{q+1}\,R\left(\frac{1}{q}\right)\right] \tag{24.38c}$$

$$R\left(\frac{1}{q}\right) = \frac{1}{2}\left[(q+1)\,P\left(\frac{1}{q^2}\right) + (q+3)\,R\left(\frac{1}{q^2}\right)\right] \tag{24.38d}$$

$$P\left(\frac{1}{q}\right) = \frac{1}{2}\left[(q+1)\,P\left(\frac{1}{q^2}\right) + (q-1)\,R\left(\frac{1}{q^2}\right)\right] \tag{24.38e}$$

$$\frac{R(y)+P(y)}{2} = \frac{y}{1-y} = \sum_{n=1}^{\infty} y^n \tag{24.38f}$$

$$\frac{R(y)-P(y)}{2} = R(y^2) \tag{24.38g}$$

$$R^2(y) - P^2(y) = \frac{4\,y}{1-y}\,R(y^2) \tag{24.38h}$$

24.5 An iteration related to the period-doubling sequence

Define

$$T(y) = \sum_{n=0}^{\infty} \frac{y^{2^n}}{1+(-1)^n\,y^{2^n}} \tag{24.39a}$$

$$= \sum_{n=0}^{\infty} (-1)^n\,\frac{y^{2^n}}{1-y^{2^n}} \tag{24.39b}$$

$$= y + y^3 + y^4 + y^5 + y^7 + y^9 + y^{11} + y^{12} + y^{13} + y^{15} + y^{16} + y^{17} + \ldots \tag{24.39c}$$

The function can be computed by the iteration

$$A_1 = 0 \qquad L_1 = y \qquad R_1 = y \qquad Y_1 = y \tag{24.40a}$$

$$A_{n+1} = L_n + Y_n\,R_n \quad \to T(y) \tag{24.40b}$$

$$L_{n+1} = L_n + Y_n\,A_n \tag{24.40c}$$

$$R_{n+1} = R_n + Y_n\,A_n \tag{24.40d}$$

$$Y_{n+1} = Y_n^2 \tag{24.40e}$$

Implementation in pari/gp:

```
t2(y, N=11)=
{ /* correct to order = 2^N-1 */
    local(A, L, R, t);
    A=0;  L=y;  R=y;
    for(k=2, N,
        t = y*A;
        A = L + y*R;
        L += t;  R += t;
        y *= y;
    );
    return( A );
}
```

The computed Taylor series can be obtained symbolically by starting with a single zero and applying the substitution rules $0 \to 11$ and $1 \to 10$. The evolution is:

```
T0 =  0
T1 =  1 1
T2 =  1 0 1 0
T3 =  1 0 1 1 1 0 1 1
T4 =  1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0
T5 =  1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1
T6 =  1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 ...
```

Observe that $T_n = L(T_{n-1}).T_{n-2}.R(T_{n-1}).T_{n-2}$ where $L(.)$ and $R(.)$ denote the left and right half of its arguments. The limiting sequence is the so-called *period-doubling sequence*. This is sequence A035263 of [OEIS] where it is called the *first Feigenbaum symbolic sequence*.

Define the *period-doubling constant* as $T := T(1/2)$, then

$$\begin{aligned} T &= 0.729427023494948405709066206894052617060026944465854741 7\ldots \qquad (24.41)\\ [\text{base } 2] &= 0.101110101011101110111010101110101011101010111011101110 10\ldots \\ [\text{CF}] &= [0,1,2,1,2,3,2,8,1,1,1,2,1,8,6,1,2,1,2,8,1,2,2,1,1,24,2,2,2,1,\\ &\quad 8,2,1,2,1,8,6,1,2,1,2,2,1,1,1,13,1,1,8,2,13,1,1,1,2,2,1,2,1,\\ &\quad 6,8,1,2,1,2,8,1,2,2,1,1,24,2,2,2,1,8,2,1,2,1,11,1,9,2,1,116,\ldots] \end{aligned}$$

One finds the functional equation

$$T(y) + T(y^2) = \frac{y}{1-y} \qquad (24.42)$$

The Taylor series of $T(y^2)$ has coefficients one where the series of $T(y)$ has coefficients zero.

Connection to the towers of Hanoi puzzle

	pile_0	pile_+	pile_-	moved disk	summary	direction of move
0:	1111	...	....	...1	0 0 0 0	
1:	111.	...1	....	...1	0 0 0 +	1
2:	11..	...1	..1.	..1.	0 0 - +	0
3:	11..	....	..11	...1	0 0 - -	1
4:	1...	.1..	..11	.1..	0 + - -	1
5:	1..1	.1..	..1.	...1	0 + - 0	1
6:	1..1	.11.	....	..1.	0 + + 0	0
7:	1...	.111	....	...1	0 + + +	1
8:	....	.111	1...	1...	- + + +	0
9:	....	.11.	1..1	...1	- + + -	1
10:	..1.	.1..	1..1	..1.	- + 0 -	0
11:	..11	.1..	1...	...1	- + 0 0	1
12:	..11	...1	11..	.1..	- - 0 0	1
13:	..1.	...1	11..	...1	- - 0 +	1
14:	....	...1	111.	..1.	- - - +	0
15:	....	....	1111	...1	- - - -	1

Figure 24.2: Solution of the towers of Hanoi puzzle for 4 disks. The rightmost column corresponds to the direction of the move made, it is the period-doubling sequence.

The *towers of Hanoi* puzzle consists of three piles and n disks of different size. The initial configuration is that all disks are on the leftmost pile ordered by size (smallest on top). The task is to move all disks to the rightmost pile by moving only one disk at a time and never putting a bigger disk on top of a smaller one.

The puzzle with n disks can be solved in $2^n - 1$ steps. Figure 24.2 shows the solution for $n = 4$ [FXT: demo bits/hanoi-demo.cc]. Here the piles are represented as binary words. Note that with each move the lowest bit in one of the three words is moved to another word where it is again the lowest bit.

A simple solution can be obtained by observing that the disk moved with step $k = 1, \ldots, 2^n - 1$ corresponds to the lowest set bit in the binary representation of k and the index of the untouched pile changes by $+1$ mod 3 for n even and -1 mod 3 for n odd.

The essential part of the implementation is

```
void
hanoi(ulong n)
{
    ulong f[3];
    f[0] = first_comb(n);  f[1] = 0;  f[2] = 0;  // Initial configuration
    int dr = (2*(n&1)-1);  // == +1 (if n even),  else  == -1
    // PRINT configuration
    int u;  // index of tower untouched in current move
    if ( dr<0 ) u=2;  else u=1;
    ulong n2 = 1UL<<n;
    for (ulong k=1; k<n2; ++k)
    {
        ulong s = lowest_bit(k);
        ulong j = 3;  while ( j-- )  f[j] ^= s;  // change all piles
        f[u] ^= s;  // undo change for untouched pile
        u += dr;
        if ( u<0 ) u=2;  else if ( u>2 ) u=0;  // modulo 3
        // PRINT configuration
    }
}
```

Now with each step the transferred disk is moved by $+1$ or -1 position (modulo 3). The rightmost column in figure 24.2 on the preceding page consists of zeros and ones corresponding to the direction of the move. It is the period-doubling sequence.

24.5.1 Generalizations

The functional equation for the period-doubling sequence, relation 24.42 can be generalized in several ways. One can look for a function for which $F_3(y) + F_3(y^3) = \frac{y}{1-y}$. It is given by

$$F(y) = \sum_{k=0}^{\infty} (-1)^n \frac{y^{3^n}}{1-y^{3^n}} \tag{24.43a}$$

$$= z + z^2 + z^4 + z^5 + z^7 + z^8 + z^9 + z^{10} + z^{11} + z^{13} + z^{14} + z^{16} + z^{17} + \ldots \tag{24.43b}$$

We can compute a constant:

$$F_3(1/2) = 0.859099796854703104903572502841974202614239955559439087 4\ldots \tag{24.44}$$

But this is just the Thue constant, see 24.2 on page 636. Large terms occur in the continued fraction expansion of this constant. Even larger terms occur in the continued fractions of $F_m(1/2)$ for $m > 3$ (replace 3 by m in relation 24.43a). For example, the 45-th term of $F_5(1/2)$ has 565 digits. In contrast, the greatest of the first 1630 terms of the continued fraction of $F_2(1/2) = T(1/2)$ equals 288. A few of the sequences corresponding to the higher order analogues of the period-doubling sequence are shown in figure 24.3 on the next page.

A different way to generalize is to search functions for which, for example, the following functional equation holds:

$$F(y) + F(y^2) + F(y^3) = \frac{y}{1-y} \tag{24.45}$$

The equation can be solved by writing $F(y) = y/(1-y) - F(y^2) - F(y^3)$ and using recursions that terminate when a prescribed order is reached.

```
F(z, R)=
{/* solve F(y) + F(y^2) + F(y^3) = y/(1-y) */
    local(s, y);
    y = z + R;
```

```
? default(realprecision,85);
? tm(y,m,N=8)=sum(n=0,N,(-1)^n*(y^(m^n)/(1-y^(m^(n)))))
? for(m=2,9,print(m,": ",tm(1.0/10,m,4));print("   ",tm(1.0/10^m,m,4)))
    2: 1.111.1.1.111.111.111.1.1.111.111.111.1.1.111.111.111.1.1.111.111.111
       .1...1.1.1...1...1...1.1.1...1.1.1...1.1.1...1...1...1.1.1...1.1.1...1...1...
    3: 11.11.11111.11.11111.11.11.11.11.11111.11.11111.11.11.11.11.11111.11.11111.11.11111.1
       ..1..1.....1..1.....1..1..1..1..1.....1..1.....1..1..1..1..1.....1..1.....1..1.....1.
    4: 111.111.111.1111111.111.111.1111111.111.111.1111111.111.111.111.111.111.111.1111111.1
       ...1...1...1.......1...1...1.......1...1...1.......1...1...1...1...1...1...1.......1.
    5: 1111.1111.1111.1111.111111111.1111.1111.1111.111111111.1111.1111.1111.111111111.1111.
       ....1....1....1....1.........1....1....1....1.........1....1....1....1.........1....1
    6: 11111.11111.11111.11111.11111.1111111111.11111.11111.11111.11111.1111111111.11111.1
       .....1.....1.....1.....1.....1..........1.....1.....1.....1.....1..........1.....1.
    7: 111111.111111.111111.111111.111111.111111.111111111111.111111.111111.111111.111111.1
       ......1......1......1......1......1......1............1......1......1......1......1.
    8: 1111111.1111111.1111111.1111111.1111111.1111111.1111111.11111111111111.1111111.11111
       .......1.......1.......1.......1.......1.......1.......1..............1.......1.....
    9: 11111111.11111111.11111111.11111111.11111111.11111111.11111111.11111111.111111111111
       ........1........1........1........1........1........1........1........1............
```

Figure 24.3: Visual verification of the higher order analogues of the period-doubling sequence. In the output lines the leading '0.' was removed and all zeros were replaced by dots.

```
    s = y/(1-y);
    if ( y^2!=R,  s -= F(z^2, R) );
    if ( y^3!=R,  s -= F(z^3, R) );
    return(s);
}
```

We obtain

```
? N=55;
? default(seriesprecision,N);
? s=F(y,O(y^N))
  y + y^4 + y^5 + y^6 + y^7 + y^9 + y^11 - y^12 + y^13 + y^16 + y^17 -
  y^18 + y^19 + y^20 + y^23 + 2*y^24 + y^25 + y^28 + y^29 + y^30 + y^31 +
  y^35 + 3*y^36 + y^37 + y^41 + y^42 + y^43 + y^44 + y^45 + y^47 -
  2*y^48 + y^49 + y^52 + y^53 + 2*y^54 + O(y^55)
```

To verify that the function $F(.)$ actually satisfies the given functional equation we show the sequences of coefficients of the power series of $F(y)$, $F(y^2)$, $F(y^3)$ and their sum:

```
  F(y)=   [0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1,-1, 1, 0, 0, 1, 1,-1, 1, 1, 0, 0, 1, 2, 1, ...]
  F(y^2)= [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0,-1, 0, ...]
  F(y^3)= [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, ...]
  sum=    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...]
```

The sequence of coefficients seems to contain all natural numbers.

The Taylor series of $F(y)$ where $F(y) + F(y^2) + F(y^3) + F(y^6) = y/(1-y)$ contains only ones and zeros. One has

$$F(y) \;=\; \sum_{k=1}^{\infty} R(k)\,\frac{x^k}{1-x^k} \quad \text{where} \quad R(k) \;=\; \begin{cases} (-1)^{e_2+e_3} & \text{if} \quad k = 2^{e_2}\,3^{e_3} \\ 0 & \text{else} \end{cases} \tag{24.46}$$

This is a *Lambert series*, it can be converted into a Taylor series by

$$F(y) \;=\; \sum_{k=1}^{\infty} r(k)\,x^k \quad \text{where} \quad r(k) \;=\; \sum_{d\backslash k} R(k) \tag{24.47}$$

It turns out that with $k = u\,2^{e_2}\,3^{e_3}$ (where neither 2 nor 3 divides u),

$$r(k) \;=\; \begin{cases} 0 & \text{if either of } e_2 \text{ or } e_3 \text{ is odd} \\ 1 & \text{else} \quad \text{(i.e. both } e_2 \text{ and } e_3 \text{ are even)} \end{cases} \tag{24.48}$$

$F(y) = y + y^4 + y^5 + y^7 + y^9 + y^{11} + y^{13} + y^{16} + y^{17} + y^{19} + y^{20} + y^{23} + y^{25} + y^{28} + y^{29} + \ldots$

```
F(y)=   .1..11.1.1.1.1..11.11..1.1..11.1...111...1.111.1.1..11.1...1.1.111.11..1.1
F(y^2)= ..1.....1.1...1...1...1...1.....1.1...1.1.....1...1.....1.1...1.......1.1.
F(y^3)= ...1........1..1.....1.....1.....1.....1........1..1.....1..1........1....
F(y^6)= ......1.................1.....1...........1...........1...........1.......
sum=    .1111111111111111111111111111111111111111111111111111111111111111111111111
```

Figure 24.4: Visual demonstration that $F(y)$ satisfies the functional equation $F(y) + F(y^2) + F(y^6) + F(y^6) = y/(1-y)$. Dots are used for zeros.

We finally note that a Taylor series can be converted to a Lambert series via

$$R(k) \quad = \quad \sum_{d \backslash k} \mu(k/d)\, r(k) \tag{24.49}$$

24.6 An iteration from substitution rules with sign

Substitution rules: $0 \to$ L1, $1 \to 10$, and L $\to$ L0.

```
D0 =  1
D1 =  10
D2 =  10L1
D3 =  10L1L010
D4 =  10L1L010L01L10L1
D5 =  10L1L010L01L10L1L01L10L010L1L010
D --> 10L1L010L01L10L1L01L10L010L1L010L01L10L010L1L01L10L1L010L01L...
```

Identify L with -1 and observe that for $n > 1$ one has $D_n = D_{n-1}.(-D_{n-2}).D_{n-2}$. The length of the strings is 2^n. Define $D(y)$ by the iteration

$$\begin{aligned}
L_0 &= 1 && (24.50a)\\
R_0 &= 1(+0\,y) && (24.50b)\\
Y_0 &= y && (24.50c)\\
L_{n+1} &= R_n && (24.50d)\\
R_{n+1} &= R_n + Y_n^2\ (-L_n + Y_n\, L_n) \quad \to D(y) && (24.50e)\\
Y_{n+1} &= Y_n^2 && (24.50f)
\end{aligned}$$

Implementation:

```
dd(y, N=7)=
{
    local(R, L, y2, t);
    /* correct up to order 2^(N) */
    L = 1;
    R = 1 + 0*y;
    for(k=1, N,
      /* (L, R) --> (R,  R.(-L).L) */
      y2 = y^2;
      t = R;
      R = R + y2*(-L + y*L);  /* R.(-L).L */
      L = t;
      y = y2;
    );
    return( R )
}
```

One finds:

$$D(y) \quad = \quad 1 - y^2 + y^3 - y^4 + y^6 - y^8 + y^{10} - y^{11} + y^{12} - y^{14} + y^{15} - y^{16} + y^{18} \pm \ldots \tag{24.51}$$

and (see relation 24.4 on page 634)

$$\frac{y}{1-y}\, D(y) \quad = \quad K(y) \;=\; y + y^2 + y^4 + y^7 + y^8 + y^{11} + \ldots \tag{24.52}$$

The coefficients are one where the Thue-Morse sequence equals minus one. Thereby the parity number can be computed as $P = \frac{1}{2}\, D\left(\frac{1}{2}\right)$.

A functional equation for $D(.)$ is

$$D\,(y) \quad = \quad y\,\frac{1-y}{1+y}\,D\left(y^2\right) + \frac{1}{1+y} \tag{24.53}$$

24.7 Iterations related to the sum of digits

The sequence of the base-2 sum of digits of the natural numbers starting with zero can be constructed as follows:

```
S0 = 0
S1 = 0 1
S2 = 0 1 1 2
S3 = 0 1 1 2 1 2 2 3
S4 = 0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4
S5 = 0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4 1 2 2 3 2 3 3 4 2 3 3 4 3 4 4 5
```

Observe that $S_n = S_{n-1}.(S_{n-1} + I_{n-1})$ where I_n is a sequence of n ones and addition is element-wise.

Define $S(y)$ by

$$I_1 \;=\; 1 \qquad A_1 \;=\; y \qquad Y_1 \;=\; y \tag{24.54a}$$

$$I_{n+1} \;=\; I_n\,(1+Y_n) \;=\; \sum_{k=0}^{2^n-1} y^k \tag{24.54b}$$

$$Y_{n+1} \;=\; Y_n^2 \tag{24.54c}$$

$$A_{n+1} \;=\; A_n + Y_{n+1}\,(I_{n+1} + A_n) \quad \to S(y) \tag{24.54d}$$

Implementation in pari/gp:

```
s2(y, N=7)=
{
    local(in, A);
    /* correct to order = 2^N-1 */
    in = 1; /* 1+y+y^2+y^3+...+y^(2^k-1) */
    A = y;
    for(k=2, N,
        in *= (1+y);
        y *= y;
        A += y*(in + A);
    );
    return( A );
}
```

The Taylor series is

$$S(y) \quad = \quad 0 + y + y^2 + 2y^3 + y^4 + 2y^5 + 2y^6 + 3y^7 + y^8 + 2y^9 + 2y^{10} + 3y^{11} + 2y^{12} + 3y^{13} + \ldots \tag{24.55}$$

Define the *sum-of-digits constant* as $S := S(1/2)/2$, then

$$S \quad = \quad 0.596063172117821679423793925862790645462361238478109932 6\ldots \tag{24.56}$$

$$[\text{base } 2] \quad = \quad 0.100110001001011110011000100101101001100010010111100110001\ldots$$

$$[\text{CF}] \quad = \quad [0, 1, 1, 2, 9, 1, 3, 5, 1, 2, 1, 1, 1, 1, 8, 2, 1, 1, 2, 1, 12, 19, 24, 1, 18, 12, 1, \ldots]$$

For the function $S(.)$ one finds:

$$S(y) \quad = \quad (1+y)\,S(y^2) + \frac{y}{1-y^2} \tag{24.57}$$

This is a functional equation of the form

$$F(y) \quad = \quad A(y)\,F(y^2) + B(y) \tag{24.58}$$

where $A(y) = 1 + y$ and $B(y) = y/(1 - y^2)$. It has the solution

$$F(y) \;=\; \sum_{k=0}^{\infty} \left[\prod_{j=0}^{k-1} \left[A(y^{2^j}) \right] B(y^{2^k}) \right] \tag{24.59}$$

This can be seen by applying the functional equation several times:

$$\begin{aligned} F(y) &= A(y)\, F(y^2) + B(y) && (24.60a)\\ &= A(y)\, \left[A(y^2)\, F(y^4) + B(y^2) \right] + B(y) && (24.60b)\\ &= A(y)\, \left[A(y^2)\, \left[A(y^4)\, F(y^8) + B(y^4) \right] + B(y^2) \right] + B(y) \;=\; \ldots && (24.60c)\\ &= B(y) + A(y)\, B(y^2) + A(y)\, A(y^2)\, B(y^4) + \ldots && (24.60d) \end{aligned}$$

We also have (see [100])

$$S(y) \;=\; \frac{1}{1-y} \sum_{k=0}^{\infty} \frac{y^{2^k}}{1 + y^{2^k}} \tag{24.61}$$

Thereby $S(y)$ equals both sides of

$$\frac{1}{1-y} \sum_{k=0}^{\infty} \frac{y^{2^k}}{1 + y^{2^k}} \;=\; \sum_{k=0}^{\infty} \left[\prod_{j=0}^{k-1} \left[1 + y^{2^j} \right] \frac{y^{2^k}}{1 - y^{2^k}} \right] \tag{24.62}$$

Such an identity follows whenever we have both a relation of the form 24.61 and 24.58.

24.7.1 Weighted sum of digits

Define $W(y)$ by

$$\begin{aligned} I_1 &= \frac{1}{2} \qquad A_1 \;=\; 1 \qquad Y_1 \;=\; y && (24.63a)\\ l_{n+1} &= I_n\, \frac{(1 + Y_n)}{2} \;=\; \frac{1}{2^n} \sum_{k=0}^{2^n - 1} y^k && (24.63b)\\ Y_{n+1} &= Y_n^2 && (24.63c)\\ A_{n+1} &= A_n + Y_{n+1}\, (I_{n+1} + A_n) \quad \rightarrow W(y) && (24.63d) \end{aligned}$$

Implementation in pari/gp:

```
w2(y, N=7)=
{
    local(in, y2, A);
    /* correct to order = 2^N-1 */
    in = 1/2; /* 1/2^k * (1+y+y^2+y^3+...+y^(2^k-1)) */
    A = y/2;
    for(k=2, N,
        in *= (1+y)/2;
        y *= y;
        A += y*(in + A);
    );
    return( A );
}
```

In the Taylor series

$$\begin{aligned} W(y) &= 0+\frac{1}{2}y+\frac{1}{4}y^2+\frac{3}{4}y^3+\frac{1}{8}y^4+\frac{5}{8}y^5+\frac{3}{8}y^6+\frac{7}{8}y^7+ \\ &+\frac{1}{16}y^8+\frac{9}{16}y^9+\frac{5}{16}y^{10}+\frac{13}{16}y^{11}+\frac{3}{16}y^{12}+\frac{11}{16}y^{13}+\frac{7}{16}y^{14}+\frac{15}{16}y^{15}+ \\ &+\frac{1}{32}y^{16}+\frac{17}{32}y^{17}+\frac{9}{32}y^{18}+\frac{25}{32}y^{19}+\frac{5}{32}y^{20}+\frac{21}{32}y^{21}+\frac{13}{32}y^{22}+\frac{29}{32}y^{23}+ \\ &+\frac{3}{32}y^{24}+\frac{19}{32}y^{25}+\frac{11}{32}y^{26}+\frac{27}{32}y^{27}+\frac{7}{32}y^{28}+\frac{23}{32}y^{29}+\frac{15}{32}y^{30}+\frac{31}{32}y^{31}+\frac{1}{64}y^{32}+\ldots \end{aligned} \tag{24.64}$$

the coefficient of the y^n is the weighted sum of digits $w(n)=\sum_{i=0}^{\infty}2^{-(i+1)}\,b_i$ where $b_0,b_1,\ldots$ is the base-2 representation of n. Note that the numerator in the n-th coefficient is the reversed binary expansion of n.

The corresponding *weighted sum-of-digits constant* or *revbin constant* is $W:=W(1/2)$. Then

$$\begin{aligned} W &= 0.448526550676272378923687721254526097616278813538448133 6\ldots \\ [\text{base 2}] &= 0.011100101101001010100010111010010100010101101001010100 01\ldots \\ [\text{CF}] &= [0,2,4,2,1,4,18,1,2,6,5,17,2,14,1,1,1,2,1,1,2,1,3,1,29,4,1,\ldots] \end{aligned} \tag{24.65}$$

One finds for the function $W(.)$:

$$W\left(\frac{1}{q}\right) = \frac{1}{2}\left[\frac{1+q}{q}\,W\left(\frac{1}{q^2}\right)+\frac{q}{q^2-1}\right] \tag{24.66}$$

24.8 Iterations related to the binary Gray code

24.8.1 Series where coefficients are the Gray code of exponents

We construct a function with Taylor series coefficients that are the binary Gray code of the exponent of y. A list of the Gray codes (second line below) can in pari/gp be generated via `for(k=0,33, print1(" ",gray(k)))` where `gray(k)=bitxor(k,floor(k/2))` (see section 1.16 on page 33):

```
k ==
0 1 2 3 4 5 6 7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0 1 3 2 6 7 5 4 12 13 15 14 10 11  9  8 24 25 27 26 30 31 29 28 20 21 23 22 18 19 17 16
== graycode(k)
```

Compute $G(y)$ by

$$\begin{aligned} F_1 &= y \qquad B_1 = 1 \qquad Y_1 = y \qquad I_1 = 1+y && \text{(24.67a)} \\ Y_n &= Y_{n-1}^2 && \text{(24.67b)} \\ F_n &= (F_{n-1} \qquad\qquad) + Y_n\,(B_{n-1}+2\,I_{n-1}) \quad \to G(y) && \text{(24.67c)} \\ B_n &= (F_{n-1}+2\,I_{n-1}) + Y_n\,(B_{n-1} \qquad\qquad) && \text{(24.67d)} \\ I_n &= I_{n-1}\,(1+Y_n) && \text{(24.67e)} \end{aligned}$$

Implementation in pari/gp:

```
gg(y, N=15)=
{
    local(N, t, ii, F, B, Fp, Bp);
    /* correct up to order 2^N-1 */
    F=0+y;  B=1+0;  ii=1+y;
    for(k=2,N,
      y *= y;
      ii *= 2;  /* delete for sum of digits */
      Fp = (F     ) +  y * (B + ii);
      Bp = (F + ii) +  y * (B     );
      F = Fp;  B = Bp;
      ii *= (1+y);
```

```
    );
    return( F )
}
```

In the algorithm `F` contains the approximation so far and `B` contains the reversed polynomial:

```
 ---- k = 1 :
F = (y)
B = (1)
 ---- k = 2 :
F = (2y^3+3y^2+y)
B = (y^2+3y+2)
 ---- k = 3 :
F = (4y^7+5y^6+7y^5+6y^4+2y^3+3y^2+y)
B = (y^6+3y^5+2y^4+6y^3+7y^2+5y+4)
 ---- k = 4 :
F = (8y^15+9y^14+11y^13+10y^12+14y^11+15y^10+13y^9+12y^8+4y^7+5y^6+7y^5+6y^4+2y^3+3y^2+y)
B = (y^14+3y^13+2y^12+6y^11+7y^10+5y^9+4y^8+12y^7+13y^6+15y^5+14y^4+10y^3+11y^2+9y+8)
```

Indeed, we get:

$$\begin{aligned} G(y) &= 0 + 1y + 3y^2 + 2y^3 + 6y^4 + 7y^5 + 5y^6 + 4y^7 + \\ &\quad +12y^8 + 13y^9 + 15y^{10} + 14y^{11} + 10y^{12} + 11y^{13} + 9y^{14} + 8y^{15} + \\ &\quad +24y^{16} + 25y^{17} + 27y^{18} + 26y^{19} + 30y^{20} + 31y^{21} + 29y^{22} + 28y^{23} + \\ &\quad +20y^{24} + 21y^{25} + 23y^{26} + 22y^{27} + 18y^{28} + 19y^{29} + 17y^{30} + 16y^{31} + \ldots \end{aligned} \tag{24.68}$$

Yet another constant, $G := G(1/2)$, the *Gray code constant*:

$$\begin{aligned} G &= 2.302218287787689301229333006391310761000431077704369505\ldots \\ [\text{base } 2] &= 10.010011010101111000101101011111100100110100011111000101\ldots \\ [\text{CF}] &= [2, 3, 3, 4, 4, 1, 4, 4, 1, 2, 1, 1, 1, 2, 24, 205, 1, 4, 2, 2, 1, 1, 4, 10, 8, 1, 9, 1, \ldots] \end{aligned} \tag{24.69}$$

For the function $G(.)$ one finds:

$$G\left(\frac{1}{q^2}\right) = \frac{1}{4}\left[\frac{q}{q+1}\,G\left(\frac{1}{q}\right) + \frac{q}{q-1}\,G\left(-\frac{1}{q}\right)\right] \tag{24.70a}$$

$$G\left(\frac{1}{q}\right) = \frac{2\,(q+1)}{q}\,G\left(\frac{1}{q^2}\right) + \frac{q^2}{(q-1)\,(q^2+1)} \tag{24.70b}$$

$$G(y) = 2\,(1+y)\,G(y^2) + \frac{y}{(1-y)\,(1+y^2)} \tag{24.70c}$$

$$G\left(-\frac{1}{q}\right) = \frac{2\,(q-1)}{q}\,G\left(\frac{1}{q^2}\right) - \frac{q^2}{(q+1)\,(q^2+1)} \tag{24.70d}$$

$$G\left(\frac{1}{q}\right) = \frac{2\,(q^2+1)}{q\,(q-1)}\,G\left(-\frac{1}{q^2}\right) + \frac{q^2\,(q^4+4\,q^3+4\,q+1)}{(q^4+1)\,(q^2+1)\,(q-1)} \tag{24.70e}$$

$$G\left(\frac{1}{q}\right) = \frac{1}{2\,q^2}\left[\frac{(q+1)\,(q^4+4\,q^3+4\,q+1)}{(q^2+1)}\,G\left(\frac{1}{q^2}\right) - \frac{(q^4+1)}{(q-1)}\,G\left(-\frac{1}{q^2}\right)\right] \tag{24.70f}$$

The function $G(y)$ can be expressed as:

$$G(y) = \frac{1}{1-y}\sum_{k=0}^{\infty}\frac{2^k\,y^{2^k}}{1+y^{2^{k+1}}} \tag{24.71}$$

24.8.2 Differences of the Gray code

If one defines $F(y) = (1-y)\,G(y)$ to get the successive differences of the Gray code itself, the Taylor coefficients are powers of two in magnitude:

$$F(y) = 0 + y + 2y^2 - y^3 + 4y^4 + y^5 - 2y^6 - y^7 + 8y^8 + y^9 + 2y^{10} - y^{11} - 4y^{12} + y^{13} \pm \ldots \tag{24.72}$$

One finds

$$F(y) = 2F\left(y^2\right) + \frac{y}{1+y^2} \tag{24.73}$$

Now, as $y/(1+y) = q/(1+q)$ for $q = 1/y$:

$$F(y) = F\left(\frac{1}{y}\right) \tag{24.74a}$$

$$= \sum_{k=0}^{\infty} 2^k \frac{y^{2^k}}{1+y^{2^{k+1}}} \tag{24.74b}$$

Thereby $F(y)$ can be computed everywhere except on the unit circle. The sum

$$\sum_{k=0}^{\infty} 2^k \frac{y^{2^k}}{1-y^{2^{k+1}}} \tag{24.75}$$

leads to a series with coefficients

```
0 1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 16 1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 32 1 ...
```

corresponding to the 'exponential' version of the ruler function which is defined as the highest power of two that divides n. The 'usual' ruler function (section 24.4 on page 639)

```
. 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 5 0 1 ...
```

is the base-2 logarithm of that series.

24.8.3 Sum of Gray code digits

The sequence of the sum of digits of the Gray code of $k \geq 0$ is (entry A005811 in [OEIS]):

```
0 1 2 1 2 3 2 1 2 3 4 3 2 3 2 1 2 3 4 3 4 5 4 3 2 3 4 3 2 3 2 1 2 3 ...
```

Omit the factor 2 in relations 24.67c and 24.67d. That is, in the implementation simply remove the line

```
ii *= 2;  /* delete for sum of digits */
```

Let $R(y)$ be the corresponding function, and define the *sum of Gray code digits constant* as $R := R(1/2)/2$, then

$$\begin{aligned} R &= 0.701472376403734520735595521064133208822798986165421295 4\ldots \\ [\text{base } 2] &= 0.101100111001001110110001100100111011001110010001101100 0\ldots \\ [\text{CF}] &= [0, 1, 2, 2, 1, 6, 10, 1, 9, 53, 1, 1, 3, 10, 1, 2, 1, 3, 2, 14, 2, 1, 2, 1, 3, 4, 2, \\ &\quad 1, 34, 1, 1, 3, 1, 1, 109, 1, 1, 4, 2, 9, 1, 642, 51, 4, 3, 2, 2, 2, 2, 1, 2, 3, \ldots] \end{aligned} \tag{24.76}$$

One finds:

$$R\left(\frac{1}{q^2}\right) = \frac{1}{2}\left[\frac{q}{q+1} R\left(\frac{1}{q}\right) + \frac{q}{q-1} R\left(-\frac{1}{q}\right)\right] \tag{24.77a}$$

$$R\left(\frac{1}{q}\right) = \frac{q+1}{q} R\left(\frac{1}{q^2}\right) + \frac{q^2}{q^3-q^2+q-1} \tag{24.77b}$$

$$R\left(-\frac{1}{q}\right) = \frac{q-1}{q} R\left(\frac{1}{q^2}\right) - \frac{q^2}{q^3+q^2+q+1} \tag{24.77c}$$

The function $R(y)$ can be expressed as

$$R(y) = \frac{1}{1-y} \sum_{k=0}^{\infty} \frac{y^{2^k}}{1+y^{2^{k+1}}} \tag{24.78}$$

Define the constant P as $P = (R+1)/2$

$$\begin{aligned} P &= 0.850736188201867260367797760532066604411399493082710647 7\ldots \\ [\text{base } 2] &= 0.110110011100100111011000110010011101100111001000110110 0\ldots \\ [\text{CF}] &= [0,1,5,1,2,3,21,1,4,107,7,5,2,1,2,1,1,2,1,6,1,2,6,1,1,8,1, \\ &\quad 2,17,3,1,1,3,1,54,3,1,1,1,2,1,4,2,321,102,2,6,1,4,1,5,2,\ldots] \end{aligned} \tag{24.79}$$

Its binary expansion is the *paper folding sequence* (or *dragon curve sequence*), entry A014577 of [OEIS].

24.8.4 Differences of the sum of Gray code digits

Now define $E(y) = (1-y)\,R(y)$ to get the differences of the sum of Gray code digits. From this definition (and relation 24.78 on the preceding page) one sees that

$$E(y) = \sum_{k=0}^{\infty} \frac{y^{2^k}}{1+y^{2^{k+1}}} \tag{24.80}$$

All Taylor coefficients (except for the constant term) are either plus or minus one:

$$E(y) = 0 + y + y^2 - y^3 + y^4 + y^5 - y^6 - y^7 + y^8 + y^9 + y^{10} - y^{11} - y^{12} + y^{13} \pm \ldots \tag{24.81}$$

We have

$$E\left(\frac{1}{q}\right) = E\left(\frac{1}{q^2}\right) + \frac{q}{q^2+1} \tag{24.82a}$$

$$E\left(y\right) = E\left(y^2\right) + \frac{y}{y^2+1} \tag{24.82b}$$

(use $\frac{y}{y^2+1} = \frac{q}{q^2+1}$ where $q = \frac{1}{y}$ for the latter relation), thereby

$$E\left(y\right) = E\left(\frac{1}{y}\right) \tag{24.83}$$

So we can compute $E(y)$ everywhere except on the unit circle.

Compute $E(y)$ by the iteration

$$L_0 = 0 \qquad R_0 = 1 \qquad Y_0 = y \tag{24.84a}$$

$$L_{n+1} = L_n + Y_n\,R_n \quad \to E(y) \tag{24.84b}$$

$$R_{n+1} = (L_n+1) + Y_n\,(R_n-2) \tag{24.84c}$$

$$Y_{n+1} = Y_n^2 \tag{24.84d}$$

Implementation in pari/gp:

```
ge(y, N=7)=
{
    local(L, R, Lp, Rp);
    /* correct up to order 2^N-1 */
    L=0;  R=1;
    for(k=2, N,
      Lp = (L    ) +  y * (R    );
      Rp = (L + 1) +  y * (R - 2);
      L = Lp;  R = Rp;
      y *= y;
    );
    return( L + y*R )
}
```

The symbolic representations of the polynomials L and R should make the underlying idea clear:

```
L = 0
R = +

L = 0+
R = +-

L = 0++-
R = ++--

L = 0++-++--
R = +++--+--

L = 0++-++--+++--+--
R = +++-++---++--+--

L = 0++-++--+++--+--+++-++---++--+--
R = +++-++--+++--+---++-++---++--+--

L = 0++-++--+++--+--+++-++---++--+--+++-++--+++--+---++-++---++--+--
R = +++-++--+++--+--+++-++---++--+---++-++--+++--+---++-++---++--+--
```

The limit of the sequence L is entry A034947 of [OEIS]. It is a signed version of the paper-folding sequence. The sequence after the initial zero is identical to the sequence of the Kronecker symbols $\left(\frac{-1}{n}\right)$ for $n = 0, 1, 2, 3, \ldots$, see section 21.8 on page 535. Quick verification:

```
? for(n=1,88,print1(if(-1==kronecker(-1,n),"-","+")))
  ++-++--+++--+--+++-++---++--+--+++-++--++--+---++-++---++--+--+++-++--+++--+--+++-++---
```

We note that the (division-less) algorithm is fast in practice, we compute the constant R as $R = E(1/2)$ to more than 600 thousand decimal digits:

```
? N=21 ; \\ number of iterations
? B=2^N-1  \\ precision in bits
  2097151
? D=ceil(B * log(2)/log(10)) + 1  \\ precision in decimal digits
  631307
? default(realprecision,D);
? r=ge(1.0/2, N)
  0.70147237640373452073559552106 4 [... + > 600k digits]
? ##
  ***   last result computed in 6,096 ms.
? r-ge(1.0/2, N+2)  \\ checking precison
  -3.744192708419695540526787319381583798 4 E-631306
? p=(1+r)/2
  0.850736188201867260367797760532 [... + > 600k digits]
```

24.8.5 Weighted sum of Gray code digits

Define $H(y)$ by its Taylor series $H(y) = \sum_{k=1}^{\infty} h(k)y^k$ where $h(k)$ is the weighted sum of digits of the Gray code:

```
wgs(k)=
{
    local(g,t,s);  s=0;  g=gray(k);
    for(n=0,33, s+=if(bittest(g,n),1/2^(n+1),0));
    return(s);
}
for(k=1,33,  print1(" ",wgs(k)))
  1/2
  3/4 1/4
  3/8 7/8 5/8 1/8
 3/16 11/16 15/16 7/16 5/16 13/16 9/16 1/16
 3/32 19/32 27/32 11/32 15/32 31/32 23/32 7/32 5/32 21/32 29/32 13/32 9/32 25/32 17/32 1/32
 3/64 35/64 ...
```

An iteration for the computation of $H(y)$ is:

$$\begin{aligned}
F_1 &= y/2 \qquad B_1 = 1/2 \qquad Y_1 = y \qquad I_1 = (1+y)/2 && (24.85a)\\
Y_n &= Y_{n-1}^2 && (24.85b)\\
K_n &= I_{n-1}/2 && (24.85c)\\
F_n &= (F_{n-1} \qquad) + Y_n\,(B_{n-1} + K_n) \quad \to H(y) && (24.85d)\\
B_n &= (F_{n-1} + K_n) + Y_n\,(B_{n-1} \qquad) && (24.85e)\\
I_n &= K_n\,(1 + Y_n) && (24.85f)
\end{aligned}$$

Implementation in pari/gp:

```
gw(y, N=11)=
{
    local(t, ii, F, B, Fp, Bp);
    /* correct up to order 2^N-1 */
    F=0+y;  B=1+0;  ii=1+y;
    ii /= 2;  F /= 2;  B /= 2;
    for(k=2,N,
      y *= y;
      ii /= 2;
      Fp = (F     ) +  y * (B + ii);
      Bp = (F + ii) +  y * (B     );
      F = Fp;  B = Bp;
      ii *= (1+y);
    );
    return( F )
}
```

Still another constant, define the *weighted sum of Gray code digits constant* as $H := H(1/2)$, then

$$\begin{aligned}
H &= 0.533700488639284991958880481482124285854919322545611891 1\ldots && (24.86)\\
[\text{base } 2] &= 0.100010001010000010011000011000001001000001010000000111100\ldots\\
[\text{CF}] &= [0,1,1,6,1,11,4,5,6,1,13,1,3,1,18,5,77,1,2,2,3,1,2,1,1,\ldots]
\end{aligned}$$

One finds:

$$\begin{aligned}
H\left(\frac{1}{q^2}\right) &= \frac{q}{q+1}\,H\left(\frac{1}{q}\right) + \frac{q}{q-1}\,H\left(-\frac{1}{q}\right) && (24.87a)\\
H\left(\frac{1}{q}\right) &= \frac{1}{2}\left[\frac{q+1}{q}\,H\left(\frac{1}{q^2}\right) + \frac{q^2}{q^3-q^2+q-1}\right] && (24.87b)\\
H\left(-\frac{1}{q}\right) &= \frac{1}{2}\left[\frac{q-1}{q}\,H\left(\frac{1}{q^2}\right) - \frac{q^2}{q^3+q^2+q+1}\right] && (24.87c)
\end{aligned}$$

24.9 A function that encodes the Hilbert curve

We define a function $H(y)$ by the following iteration:

$$\begin{aligned}
H_1 &= +i\,y + 1\,y^2 - i\,y^3 && (24.88a)\\
R_1 &= +i\,y - 1\,y^2 - i\,y^3 && (24.88b)\\
Y_1 &= y && (24.88c)\\
Y_{n+1} &= Y_n^4 && (24.88d)\\
H_{n+1} &= -i\,R_n + Y_n\,(+i + H_n + Y_n\,(+1 + H_n + Y_n\,(-i + i\,R_n))) && (24.88e)\\
R_{n+1} &= +i\,H_n + Y_n\,(+i + R_n + Y_n\,(-1 + R_n + Y_n\,(-i - i\,H_n))) && (24.88f)\\
H_{2\,n} &\to H(y) && (24.88g)
\end{aligned}$$

As the real and imaginary parts are 'swapped' with each step we agree on iterating an even number of times. The resulting function $H(y)$ is

$$H(y) \;=\; 0 + y + i\,y^2 - y^3 + i\,y^4 + i\,y^5 + y^6 - i\,y^7 + y^8 + i\,y^9 + y^{10} - i\,y^{11} \pm \ldots \tag{24.89}$$

The coefficients of the series are ± 1 and $\pm i$, except for the constant term which is zero. If the sequence of coefficients is interpreted as follows:

```
 0  :=  goto start  '0'
+1  :=  move right  '>'
-1  :=  move left   '<'
+i  :=  move up     '^'
-i  :=  move down   'v'
```

Then, symbolically:

```
H = 0>^<^^>v>^>vv<v>>^>v>>^<^>^<<v<^^^>v>>^<^>^<<v<^<<v>vv<^<v<^^>^< ...
```

Follow the signs to walk along the Hilbert curve.

An implementation in pari/gp:

```
hh(y, N=4)=
{
    /* correct to order = 4^N-1 */
    local(H, R, tH, tR);
    H= +I*y + 1*y^2 -I*y^3;  R= +I*y - 1*y^2 -I*y^3;
    for(k=2, N,
        y=y^4;
        tH = -I*R + y*(+I + H + y*(+1 + H + y*(-I + I*R)));
        tR = +I*H + y*(+I + R + y*(-1 + R + y*(-I - I*H)));
        H=tH;  R=tR;
    );
    return( H );
}
```

The value of $H(y)$ for $y < 1$ gives the limiting point in the complex plane when the walk according to the coefficients is done with decreasing step lengths: step number k has step length y^k. The least positive y where the real and imaginary part of the endpoint are equal is $y_1 = 0.5436890126920\ldots$. It turns out that y_1 is the real solution of the polynomial $y^3 + y^2 + y - 1$. On might suspect that $M(y) := \operatorname{Re} H(y) - \operatorname{Im} H(y)$, the difference between the real and the imaginary part of $H(y)$, has the factor $y^3 + y^2 + y - 1$. Indeed, $M(y) = y\,(y^3 + y^2 + y - 1)\,(y^{12} + y^8 + y^4 - 1)\cdots$ and a similar statement is true for the $P(y) := \operatorname{Re} H(y) + \operatorname{Im} H(y)$. We use this observation for the construction of a simplified and quite elegant algorithm for the computation of $H(y)$.

24.9.1 A simplified algorithm

Define $P(y)$ by the iteration

$$\begin{aligned} Y_1 &= y \qquad P_1 \;=\; 1 && \text{(24.90a)}\\ P_{n+1} &= P_n\,\left(+1 + Y_n + Y_n^2 - Y_n^3\right) && \text{(24.90b)}\\ Y_{n+1} &= Y_n^4 && \text{(24.90c)}\\ P_n - 1 &\to P(y) && \text{(24.90d)} \end{aligned}$$

and $M(y)$ by

$$\begin{aligned} Y_1 &= y \qquad M_1 \;=\; 1 && \text{(24.91a)}\\ M_{n+1} &= M_n\,\left(-1 + Y_n + Y_n^2 + Y_n^3\right) && \text{(24.91b)}\\ Y_{n+1} &= Y_n^4 && \text{(24.91c)}\\ y\,M_n &\to M(y) && \text{(24.91d)} \end{aligned}$$

The function $H(y)$ can be computed as

$$H(y) \quad = \quad \frac{1}{2}\Big[(P(y)+M(y)) + i\,(P(y)-M(y))\Big] \tag{24.92}$$

Implementation in pari/gp:

```
/* functions correct to order = 4^N-1 */

fpp(y, N=4)=
{
    local( t, Y );
    t = 1;  Y=y;
    for (k=1, N,  t *= (+1+Y+Y^2-Y^3);  Y=Y^4; );
    return( t-1 );
}

fmm(y, N=4)=
{
    local( t, Y );
    t = 1;  Y=y;
    for (k=1, N,  t*= (-1+Y+Y^2+Y^3);  Y=Y^4; );
    return( t*y-Y );
}

hhpm(y, N=4)=
{
    local( tp, tm );
    tp = fpp(y);
    tm = fmm(y);
    return( ((tp+tm) + I*(tp-tm))/2 );
}
```

With a routine `tdir()` that prints a power series with coefficients $\in \{-1,\, 0,\, +1\}$ symbolically we get:

```
? N=4;
? tdir(fpp(y));tdir(fmm(y));
  0++-+++-++-----++++-+++-+++----++++-+++-++----+---+---+---++++-++
  0+---+++-+++-+++-++++---+---+----++++---+---+----++++---+---+----

? tdir((fpp(y)+fmm(y))/2);tdir((fpp(y)-fmm(y))/2);
  0+0-00+0+0+00-0++0+0++0-0+0--0-000+0++0-0+0--0-0--0+00-0-0-00+0-00
  00+0++0-0+0--0-00+0-00+0+0+00-0+++0-00+0+0+00-0+00-0--0+0-0++0+0++
```

The n-th coefficient of the Taylor series of $P(y)$ equals the parity of the number of threes in the radix-4 representation of n. This fact can be used for an efficient bit level algorithm, see section 1.20 on page 47.

The coefficients of the non-constant terms in the Taylor series of the function $P(.)$ can be obtained by a string substitution engine:

```
Number of symbols = 2
Start: +
Rules:
  + --> +++-
  - --> ---+
-------------
0:   (#=1)
  +
1:   (#=4)
  +++-
2:   (#=16)
  +++-+++-++----+
3:   (#=64)
  +++-+++-++----++++-+++-++----++++-+++-++----+---+---+---++++-
4:   (#=256)
  +++-+++-++----++++-+++-++----++++-+++-++----+---+---+---++++-+++-+++-++ ...

  0++-+++-++----++++-+++-++----++++-+++-++----+---+---+---++++-++ == fpp()
```

Similarly for $M(.)$:

```
Number of symbols = 2
Start: +
Rules:
  + --> -+++
  - --> +---
-------------
0:   (#=1)
  +
1:   (#=4)
  -+++
2:   (#=16)
  +----+++-+++-+++
```

```
3:   (#=64)
 -+++----+---+---+----+++-+++-++++----+++-+++-++++----+++-+++-+++
4:   (#=256)
 +----+++-+++-+++-++++---+---+----++++---+---+----++++---+---+----++++---+- ...
 0+----+++-+++-+++-++++---+---+----++++---+---+----++++---+---+---- == fmm()
```

24.9.2 The turns of the Hilbert curve

A function with real series coefficients can be found by considering the turns of the Hilbert curve. Let us agree on a '+' for a right turn, a '-' for a left turn and a zero for 'no turn'. The sequences of turns starts as

```
0--+0++--++0+--0-++-0--++--0-++00++-0--++--0-++-0--+0++--++0+--+ \
0++-0--++--0-++0+--+0++--++0+--00--+0++--++0+--+0++-0--++--0-++- \
-++-0--++--0-++0+--+0++--++0+--00--+0++--++0+--+0++-0--++--0-++0 \
+--+0++--++0+--0-++-0--++--0-++00++-0--++--0-++-0--+0++--++0+-- ...
```

The computation is slightly tricky:

```
hht(y, N=4)=
{
    /* correct to order = 4^N-1 */
    local( t, Y, F, s, p );
    t = 1;  Y=y;  p = 1;
    F = y + y^2;
    for(k=2, N,
        Y = Y^4;
        t = -F + Y*F + Y^2*F  - Y^3*F;
        p *= 4;
        if ( 0==(k%2),
            t +=  y^(1*p-1);
            t +=  y^(3*p);
            t -=  (y+1)*y^(2*p-1);
            , /* else */
            t +=  y^(1*p);
            t +=  y^(3*p-1);
        );
        F = t;
    );
    if ( 1==N%2,  F = -F );  \\ same result for even and odd N
    return( F );
}
```

24.10 Sparse variants of the inverse

24.10.1 A fourth order iteration

Define $F(y)$ as the result of the iteration

$$F_0 = 1 \qquad Y_0 = y \tag{24.93a}$$

$$F_{n+1} = F_n\,(1 + Y_n) \quad \to F(y) \tag{24.93b}$$

$$Y_{n+1} = Y_n^4 \tag{24.93c}$$

The terms of the continued fraction of $F(1/10)$ grow doubly exponentially:

```
[1, 9, 1, 90, 110, 909000, 11001100000, 9090000090900000000000,
 1100110000000000110011000000000000000000000,
 909000009090000000000000000000000909000009090000000000000000000000000000000000000000000000000000
  ... ]
```

Let $[t_0, t_1, t_2, \ldots]$ be the continued fraction of $F(1/q)$, then

$$\begin{aligned}
t_0 &= 1 && (24.94a)\\
t_1 &= q - 1 && (24.94b)\\
t_2 &= 1 && (24.94c)\\
t_3 &= q^2 - q && (24.94d)\\
t_4 &= q^2 + q && (24.94e)\\
t_5 &= q^6 - q^5 + q^4 - q^3 && (24.94f)\\
t_6 &= q^{10} + q^9 + q^6 + q^5 && (24.94g)\\
t_7 &= q^{22} - q^{21} + q^{20} - q^{19} + q^{14} - q^{13} + q^{12} - q^{11} && (24.94h)\\
t_8 &= q^{42} + q^{41} + q^{38} + q^{37} + q^{26} + q^{25} + q^{22} + q^{21} && (24.94i)\\
t_9 &= q^{86} - q^{85} + q^{84} - q^{83} + q^{78} - q^{77} + q^{76} - q^{75} + && (24.94j)\\
&\quad + q^{54} - q^{53} + q^{52} - q^{51} + q^{46} - q^{45} + q^{44} - q^{43}
\end{aligned}$$

For $j \geq 4$ one has

$$\frac{t_j}{t_{j-2}} = q^{2J} + q^J = q^J\,(q^J + 1) \quad \text{where} \quad J = 2^{j-4} \tag{24.95}$$

A functional relation for F is given by

$$F(y)\,F(y^2) = \frac{1}{1-y} \tag{24.96}$$

The relation is (mutatis mutandis) also true for the truncated product. The binary expansions of $F(y)$, $F(y^2)$ and their product for $y = 1/2$ are (dots for zeros):

```
11..11..........11..11.....................................
1111111111111111111111111111111111111111111111111111....
11111111111111111111111111111111111111111111111111111111
```

Since the expansions are palindromes also their correlation is a sequence of ones.

24.10.2 A different fourth order iteration

To define the function $F(y)$ we modify the third order iteration for $\frac{1}{1+y}$

```
inv3m(y, N=6)=  /* third order --> 1/(1+y) */
{ /* correct to order 3^N */
    local(T);
    T = 1;
    for(k=1, N,
      T *= ( 1 - y + y^2 );
      y = y^3;
    );
    return( T );
}
```

to get a fourth-order iteration:

```
f43(y, N=6)=
{ /* correct to order 4^N */
    local(T, yt);
    T = 1;
    for(k=1, N,
      T *= ( 1 - y + y^2 );
      y = y^4;  /* ! */
    );
    return( T );
}
```

That is,

$$\begin{aligned} F_0 &= 1 \qquad Y_0 = y && (24.97a)\\ F_{n+1} &= F_n\,(1 - Y_n + Y_n^2) \quad \to F(y) && (24.97b)\\ Y_{n+1} &= Y_n^4 && (24.97c) \end{aligned}$$

The first few terms of the power series are

$$F(y) = 1 - y + y^2 - y^4 + y^5 - y^6 + y^8 - y^9 + y^{10} - y^{16} + y^{17} - y^{18} + y^{20} - y^{21} \pm \ldots \qquad (24.98)$$

Let $[t_0, t_1, t_2, \ldots]$ be the continued fraction of $F(1/q)$, then

$$\begin{aligned} t_0 &= 0 && (24.99a)\\ t_1 &= 1 && (24.99b)\\ t_2 &= q && (24.99c)\\ t_3 &= q && (24.99d)\\ t_4 &= q^2 - q && (24.99e)\\ t_5 &= q^4 + q^3 - q - 1 && (24.99f)\\ t_6 &= q^8 - q^7 + q^6 - q^4 + q^3 - q^2 && (24.99g)\\ t_7 &= q^{16} + q^{15} - q^{13} + q^{11} + q^{10} - q^8 - q^7 + q^5 - q^3 - q^2 && (24.99h) \end{aligned}$$

For $j \geq 6$ one has

$$\frac{t_j}{t_{j-2}} = q^{6J} + q^{4J} + q^{3J} + q^J = q^J\,(q^J + 1)\,(q^{2J} - q^J + 1)\,(q^{2J} + 1) \qquad (24.100)$$

where $J = 2^{j-6}$. The terms of the continued fraction of $F(1/q)$ for integer q grow doubly exponentially:

```
? contfrac(f43(0.5))
  [0, 1, 2, 2, 2, 21, 180, 92820, 3032435520, 26126907554432455680,
   240254294248527099500117907463345274880,
   164001256750215347067944129734442019102853751066678216639025390799096507269120,
   ... ]
/* number of decimal digits of the terms in the CF: */
  [-, 1, 1, 1, 1, 2, 3, 5, 10, 20, 39, 78, 154, 309, ... ]
```

By construction,

$$\begin{aligned} F(y) &= \left(1 - y + y^2\right)\,F\left(y^4\right) && (24.101a)\\ F(-y) &= \left(1 + y + y^2\right)\,F\left(y^4\right) && (24.101b) \end{aligned}$$

The equivalent forms with $y = 1/q$ are

$$\begin{aligned} F\left(\frac{1}{q}\right) &= \frac{q^2 - q + 1}{q^2}\,F\left(\frac{1}{q^4}\right) && (24.102a)\\ F\left(-\frac{1}{q}\right) &= \frac{q^2 + q + 1}{q^2}\,F\left(\frac{1}{q^4}\right) && (24.102b) \end{aligned}$$

Now $q^2 - q + 1 = p^2 - p + 1$ if $p = 1 - q$, so

$$\begin{aligned} F\left(1 - \frac{1}{q}\right) &= \frac{q^2 - q + 1}{q^2}\,F\left(\left(1 - \frac{1}{q}\right)^4\right) \quad \text{where} \quad q > 1 && (24.103a)\\ F\left(1 + \frac{1}{q}\right) &= \frac{q^2 + q + 1}{q^2}\,F\left(\left(1 + \frac{1}{q}\right)^4\right) \quad \text{where} \quad q < -1 && (24.103b) \end{aligned}$$

Adding relations $\alpha\times$ (24.102a) and $\beta\times$ (24.103a) and simplifying gives

$$\frac{\alpha\,F(y) + \beta\,F(1-y)}{\alpha\,F(y^4) + \beta\,F((1-y)^4)} = y^2 - y + 1 \quad \text{where} \quad \alpha, \beta \in \mathbb{C} \qquad (24.104)$$

24.10.3 A sixth order iteration

Define $F(y)$ by the iteration

$$\begin{aligned} F_0 &= 1 \qquad Y_0 = y && (24.105a)\\ F_{n+1} &= F_n\left(1+Y_n+Y_n^2\right) \quad \to F(y) && (24.105b)\\ Y_{n+1} &= Y_n^6 && (24.105c)\end{aligned}$$

Let $[t_0, t_1, t_2, \ldots]$ be the continued fraction of $F(1/q)$, then

$$\begin{aligned} t_0 &= 1 && (24.106a)\\ t_1 &= q-1 && (24.106b)\\ t_2 &= q+1 && (24.106c)\\ t_3 &= q^2-q && (24.106d)\\ t_4 &= q^{10}+q^9+q^8+q^4+q^3+q^2 && (24.106e)\\ t_5 &= q^8-q^7+q^5-q^4 && (24.106f)\\ t_6 &= q^{64}+q^{63}+q^{62}+q^{58}+q^{57}+q^{56}+q^{52}+q^{51}+q^{50}+ && (24.106g)\\ &\quad +q^{28}+q^{27}+q^{26}+q^{22}+q^{21}+q^{20}+q^{16}+q^{15}+q^{14} && \\ t_7 &= q^{44}-q^{43}+q^{41}-q^{40}+q^{26}-q^{25}+q^{23}-q^{22} && (24.106h)\\ t_8 &= q^{388}+q^{387}+\ldots+q^{87}+q^{86} && (24.106i)\end{aligned}$$

For $j \geq 4$ one has

$$\frac{t_j}{t_{j-2}} = \begin{cases} q^J + q^{J/2} & \text{if} \quad j \quad \text{odd} \\ \left(q^{10J}+q^{9J}+q^{8J}+q^{4J}+q^{3J}+q^{2J}\right) / \left(q^J+1\right) & \text{else} \end{cases} \qquad (24.107)$$

where $J = 6^{j-4}$.

24.11 An iteration related to the Fibonacci numbers

The so-called *rabbit constant* is

$$\begin{aligned} A &= 0.709803442861291314641787399444575597012502205767860516 9\ldots && (24.108)\\ [\text{base } 2] &= 0.10110101101101011010110110101101101101011010110110101101011010110101\ldots \\ [\text{CF}] &= [0, 1, 2, 2, 4, 8, 32, 256, 8192, 2097152, 17179869184, \\ &\quad 36028797018963968, 6189700196426901374495621 12, \ldots]\end{aligned}$$

The sequence of zeros and ones after the decimal point is referred to as *rabbit sequence* or *infinite Fibonacci word*. It is sequence A005614 of [OEIS].

The rabbit sequence can be obtained by starting with a single zero and repeated application of the following substitution rules: simultaneously replace all zeros by one ($0 \to 1$, 'young rabbit gets old') and all ones by one-zero ($1 \to 10$, 'old rabbit gets[1] child'). The evolution is:

```
A0 = 0
A1 = 1
A2 = 10
A3 = 101
A4 = 10110
A5 = 10110101
A6 = 1011010110110
A7 = 10110101101101010110101
A -->1011010110110101101011011010110101...
```

[1] No sex, no death.

The crucial observation is that each element A_n can be obtained by appending A_{n-2} to A_{n-1}, that is $A_n = A_{n-1}.A_{n-2}$. If one wants to compute the value of the rabbit constant in base 2 to N bits precision the whole process requires only copying N bits of data which may be the minimal conceivable work for a (non-sparse) computation.

We define a function $A(y)$ that has the special value

$$A \;=\; \frac{1}{2}\,A\left(\frac{1}{2}\right) \tag{24.109}$$

by the equivalent operation in the power series expansion. One obtains the following iteration:

$$\begin{aligned}
L_0 &= 0 \qquad R_0 = 1 \qquad l_0 = 1 \qquad r_0 = y && \text{(24.110a)}\\
l_{n+1} &= r_n = y^{F_n} && \text{(24.110b)}\\
r_{n+1} &= r_n\, l_n = y^{F_{n+1}} && \text{(24.110c)}\\
L_{n+1} &= R_n && \text{(24.110d)}\\
R_{n+1} &= R_n + r_{n+1}\, L_n = R_n + y^{F_{n+1}}\, L_n \quad \to A(y) && \text{(24.110e)}
\end{aligned}$$

Here F_n denotes the n-th Fibonacci number (sequence A000045 of [OEIS]):

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Fn	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610

After the n-th step the series in y is correct up to order $F_{n+2} - 1$. That is, the order of convergence equals $\frac{\sqrt{5}+1}{2} \approx 1.6180$. The function $A(y)$ has the Taylor series:

$$A(y) \;=\; 1 + y^2 + y^3 + y^5 + y^7 + y^8 + y^{10} + y^{11} + y^{13} + y^{15} + y^{16} + y^{18} + y^{20} + y^{21} \ldots \tag{24.111}$$

The sequence of exponents of y in the series is entry A022342 of [OEIS].

The implementation in pari/gp is:

```
fa(y, N=10)=
{
    local(t, yl, yr, L, R, Lp, Rp);
    /* correct up to order fib(N+2)-1 */
    L=0;  R=1;   yl=1;  yr=y;
    for(k=1, N,
      t=yr;  yr*=yl;  yl=t;
      Lp=R;  Rp=R+yr*L;  L=Lp;  R=Rp;
    );
    return( R )
}
```

24.11.1 Modified initializations

We use the initialization parameters (relations 24.110a) in order to define a more general function $A[L_0, R_0, l_0, r_0]$. Then

$$A(y) \;=\; A[0,1,1,y] \;=\; A[1,1,y,y] \tag{24.112}$$

One gets a power series with coefficients R at the positions corresponding to the ones in the rabbit constant:

$$\begin{aligned}
A[L,R,1,y] &= R + Ly + Ry^2 + Ry^3 + Ly^4 + Ry^5 + Ly^6 + Ry^7 + Ry^8 + Ly^9 + Ry^{10} + && \text{(24.113)}\\
&\quad + Ry^{11} + Ly^{12} + Ry^{13} + Ly^{14} + Ry^{15} + Ry^{16} + Ly^{17} + \ldots\\
A[1,1,1,y] &= \frac{1}{1-y} && \text{(24.114)}
\end{aligned}$$

$$\begin{aligned}
A[L,R,y,y] &= R + Ly^2 + Ry^3 + Ry^5 + Ly^7 + Ry^8 + Ly^{10} + Ry^{11} + Ry^{13} + && \text{(24.115)}\\
&\quad + Ly^{15} + Ry^{16} + Ry^{18} + Ly^{20} + Ry^{21} + Ly^{23} + Ry^{24} + Ry^{26} + \ldots\\
A[\alpha L,\, \beta R,\, l,\, r] &= \alpha\, A[L,0,l,r] + \beta\, A[0,R,l,r] && \text{(24.116)}
\end{aligned}$$

$$A[0,1,T,y] = 1 + Ty^2 + T^2y^3 + T^3y^5 + T^4y^7 + T^5y^8 + T^6y^{10} + T^7y^{11} + T^8y^{13} + \dots \quad (24.117)$$
$$A[0,1,y,T] = 1 + T^2y + T^3y^2 + T^5y^3 + T^7y^4 + T^8y^5 + T^{10}y^6 + T^{11}y^7 + T^{13}y^8 + \dots \quad (24.118)$$

24.11.2 Fibonacci representation

The greedy algorithm to obtain the *Fibonacci representation* (or *Zeckendorf representation*) of an integer repeatedly subtracts the largest Fibonacci number that is greater or equal to it until the number is zero.

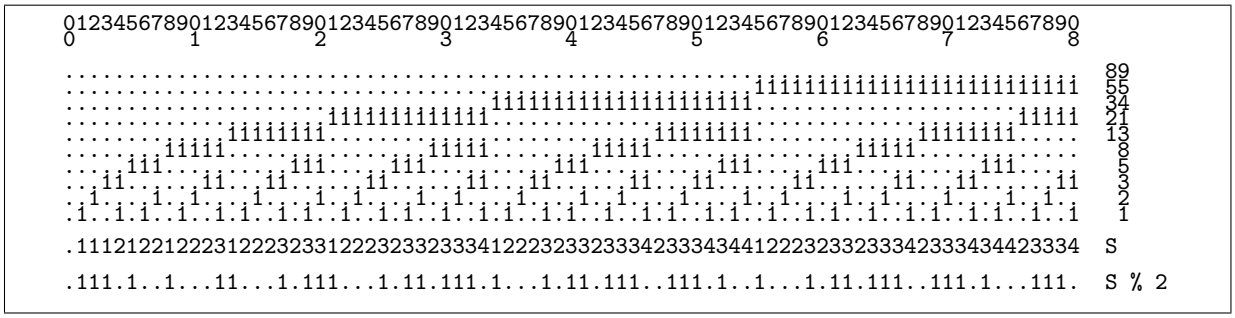


Figure 24.5: Fibonacci representations of the numbers $0 \dots 80$. A dot is used for zero. The two lower lines are the sum of digits and the sum of digits modulo two, the *Fibonacci parity*.

The Fibonacci representations of the numbers $0 \dots 80$ are shown in figure 24.5.

The sequence of lowest Fibonacci bits is

0100101001001010010100100101001001010010100100101001010010100100101001...

Interpreted as the binary number $x = 0.1001010010010\dots = 0.5803931\dots$ it turns out that $A = 1 - x/2$ (that is, $x = 2 - A(1/2)$). Alternatively, one can compute the number as $x = A[1,0,1,1/2]$.

The numbers of digits in the Fibonacci representations (second lowest row in figure 24.5) is sequence A007895 of [OEIS]. This sequence modulo two gives the *Fibonacci parity*. The sequence of the Fibonacci parities can be computed by initializing $L_0 = 1$ and changing relation 24.110e on the preceding page to

$$R_{n+1} = R_n - r_{n+1}\,L_n = R_n - y^{F_{n+1}}\,L_n \quad \to A_p(y) \quad (24.119)$$

Let the corresponding function be $A_p(y)$. We define the *Fibonacci parity constant* A_p as

$$A_p = 1 - A_p(1/2)/2 \quad (24.120a)$$
$$= 0.910533470863561763804686886771098007344581229006937645 4\dots \quad (24.120b)$$
$$[\text{base } 2] = 0.111010010001100010111000101101110100010110111001110100 10\dots$$
$$[\text{CF}] = [0,1,10,5,1,1,1,3,4,2,6,25,4,5,1,1,3,5,1,3,2,1,1,1,3,1,3,22,1,$$
$$10,1,2,3,2,73,1,111,46,1,51,2,1,1,5,1,65,3,1,3,2,5,6,1,4,1,2,\dots]$$

The sequence of bits in the binary expansion of A_p is entry A095076 of [OEIS].

24.11.3 Fibonacci representation as binary numbers

The sequence of the Fibonacci representations interpreted as binary numbers is:

0, 1, 2, 4, 5, 8, 9, 10, 16, 17, 18, 20, 21, 32, 33, 34, 36, 37, 40,
41, 42, 64, 65, 66, 68, 69, 72, 73, 74, 80, 81, 82, 84, 85, 128, 129,
130, 132, 133, 136, 137, 138, 144, 145, 146, 148, 149, 160, 161, 162,
164, 165, 168, 169, 170, 256, 257, 258, 260, 261, 264, 265, 266, 272,
273, 274, 276, 277, 288, 289, 290, 292, 293, 296, 297, 298, 320, 321,
322, 324, 325, ...

This is sequence A003714 of [OEIS], where the numbers are called *Fibbinary numbers*.

Define $F_2(y)$ to be the function that has the same sequence of Taylor coefficients:

$$F(y) \;=\; 0 + 1y + 2y^2 + 4y^3 + 5y^4 + 8y^5 + 9y^6 + 10y^7 + 16y^8 + 17y^9 + 18y^{10} + \ldots \tag{24.121}$$

A slightly more general function $F_b(y)$ (which for $b = 2$ gives the Taylor series above) can be computed by the iteration:

$$\begin{aligned}
L_0 &= 0 \qquad R_0 = y \qquad l_0 = y \qquad r_0 = y && \text{(24.122a)}\\
A_0 &= 1 \qquad B_0 = 1 \qquad b = 2 && \text{(24.122b)}\\
A_{n+1} &= b\,B_n && \text{(24.122c)}\\
B_{n+1} &= b\,[B_n + r_n\,A_n] && \text{(24.122d)}\\
l_{n+1} &= r_n = y^{F_{n+1}} && \text{(24.122e)}\\
r_{n+1} &= r_n\,l_n = y^{F_{n+2}} && \text{(24.122f)}\\
L_{n+1} &= R_n && \text{(24.122g)}\\
R_{n+1} &= R_n + r_{n+1}\,[L_n + A_{n+1}] \quad \to F_b(y) && \text{(24.122h)}
\end{aligned}$$

Implementation in pari/gp:

```
ffb(y, b=2, N=13)=
{ /* correct up to order fib(N+3)-1 */
    local(t, yl, yr, L, R, Lp, Rp, Ri, Li);
    L=0;  R=0+1*y;
    Li=1;  Ri=1;
    yl=y;  yr=y;
    for (k=1, N,
        Li*=b;  Ri*=b;
        Lp=Ri;  Rp=Ri+yr*Li;  Li=Lp;  Ri=Rp;
        t=yr;  yr*=yl;  yl=t;
        Lp=R;  Rp=R+yr*(L+Li);  L=Lp;  R=Rp;
    );
    return( R )
}
```

Using $b = 10$ and printing the Taylor coefficient (in decimal) gives:

```
? t=taylor(ffb(y,10),y)
  y + 10*y^2 + 100*y^3 + 101*y^4 + 1000*y^5 + 1001*y^6 + 1010*y^7 +
  10000*y^8 + 10001*y^9 + 10010*y^10 + 10100*y^11 + 10101*y^12 +
  100000*y^13 + 100001*y^14 + 100010*y^15 + 100100*y^16 + 100101*y^17 +
  101000*y^18 + 101001*y^19 + 101010*y^20 + 1000000*y^21 + 1000001*y^22 +
  [--snip--]

? for(i=0,14,print(polcoeff(t,i));)
         0
         1
        10
       100
       101
      1000
      1001
      1010
     10000
     10001
     10010
     10100
     10101
    100000
    100001
```

Setting $b = 1/q$ gives weighted sums of digits as coefficients. For example,

```
? t=taylor(ffb(y/10,1/10),y)
  1/10*y + 1/1000*y^2 + 1/100000*y^3 + 101/1000000*y^4 +
  1/100000000*y^5 + 1001/10000000000*y^6 + 101/10000000000*y^7 +
  1/1000000000000*y^8 + 10001/10000000000000*y^9 +
  1001/100000000000000*y^10 + 101/1000000000000000*y^11 +
  10101/10000000000000000*y^12 + 1/100000000000000000*y^13 +
  100001/1000000000000000000*y^14 + ...
```

and the *decimal weighted Fibonacci representation constant* F_{10} can be defined as

$$\begin{aligned} F_{10} &:= F_{1/10}(1/10) && \text{(24.123a)} \\ &= 0.10111202211223121231332221313322334232221332233423233424\ldots && \text{(24.123b)} \\ [\text{base } 2] &= 0.00011001111000100111101000111100001101000101011110011101\ldots \\ [\text{CF}] &= [0, 9, 1, 8, 10, 1, 3, 1, 9, 5, 3, 2, 21, 7, 1, 2, 3, 2, 4, 5, 1, 3, 4, 3, 1, 1, 2, 25, \\ &\quad 18, 1, 2, 67, 1, 1, 1, 70, 1, 18, 1, 19, 3, 4, 1, 2, 2, 1, 28, 2, 19, 6, 3, 4, 3, 94, \ldots] \end{aligned}$$

The first appearances of the decimal digits $1, 2, \ldots, 9$ are at the positions:

```
digit  position
  1         1
  2         6
  3        14
  4        35
  5        90
  6       234
  7       611
  8      1598
  9      4182
```

A generating function for a sequence starting with the given terms is

$$\frac{1 + 2\,x - 6\,x^2 + 2\,x^3}{1 - 4\,x + 4\,x^2 - x^3} = 1 + 6\,x + 14\,x^2 + 35\,x^3 + 90\,x^4 + 234\,x^5 + 611\,x^6 + 1598\,x^7 + \ldots \quad (24.124)$$

The sequence of coefficients

```
1, 6, 14, 35, 90, 234, 611, 1598, 4182, 10947, 28658, 75026, 196419, 514230, ...
```

coincides (disregarding the initial one) with entries A032908 and A093467 of [OEIS].

An functional equation for the function with Taylor coefficients 1 if the exponent is a Fibbinary number and zero else,

$$B(x) := 1 + x + x^2 + x^4 + x^5 + x^8 + x^9 + x^{10} + x^{16} + x^{17} + x^{18} + x^{20} + \ldots \quad (24.125)$$

is given (entry A003714 of [OEIS]) as

$$B(x) = x\,B(x^4) + B(x^2) \quad (24.126)$$

We turn the relation into an recursion for the computation of $B(x)$ correct up to the term x^N:

```
fibbi(x, N=25, nr=1)=
{
    if ( nr>N, return( 1 ) );
    return( x*fibbi(x^4,N,4*nr)+fibbi(x^2,N,2*nr) );
}
```

We check the functional relation:

```
? N=30;  R=O(x^(N+1));  \\ R is used to truncate terms of order >N
? t=fibbi(x,N)+R
  1 + x + x^2 + x^4 + x^5 + x^8 + x^9 + x^10 + x^16 + x^17 + x^18 + x^20 + x^21 + O(x^31)
? t2=fibbi(x^2,N)+R
  1 + x^2 + x^4 + x^8 + x^10 + x^16 + x^18 + x^20 + O(x^31)
? t4=x*fibbi(x^4,N)+R
  x + x^5 + x^9 + x^17 + x^21 + O(x^31)
? t-(t4+t2)
  O(x^31)
```

24.11.4 Digit extract algorithms for the rabbit constant

The *spectrum* of a real number x is the sequence of integers $\lfloor k \cdot x \rfloor$ where $k = 1, 2, 3, \ldots$. As mentioned in [MathWorld], the spectrum of the golden ratio $g = \frac{\sqrt{5}+1}{2} = 1.61803\ldots$ gives the exponents of y where the series for $y\,A(y)$ has coefficient one:

```
bt(x, n=25)=
{
    local(v);
    v = vector(n);
    for (k=1, n, v[k]=floor(x*k));
    return ( v );
}
```

```
g=(sqrt(5)+1)/2
 1.6180339887498948482045868343656381177203091798057628 62
```

```
n=40;
bt(g, n)
  [1, 3, 4, 6, 8, 9, 11, 12, 14, 16, 17, 19, 21, 22, 24, 25, 27, 29,
  30, 32, 33, 35, 37, 38, 40, 42, 43, 45, 46, 48, 50, 51, 53, 55, 56,
  58, 59, 61, 63, 64]
```

```
t=taylor(y*fa(y),y)
  y + y^3 + y^4 + y^6 + y^8 + y^9 + y^11 + y^12 + y^14 + y^16 + y^17 +
  y^19 + y^21 + y^22 + y^24 + y^25 + y^27 + y^29 + y^30 + y^32 + y^33 +
  y^35 + y^37 + y^38 + y^40 + y^42 + y^43 + y^45 + y^46 + y^48 + y^50 +
  y^51 + y^53 + y^55 + y^56 + y^58 + y^59 + y^61 + y^63 + y^64 + O(y^66)
```

The sequence [1, 3, 4, 6, ...] of exponents where the coefficient equals one is sequence A000201 of [OEIS]. As an easy consequence, there is a digit extract algorithm for the binary expansion of the rabbit constant. We use a binary search algorithm:

```
bts(x, k)=
{ /* return 0 if k is not in the spectrum, else return index >=1  */
    local(nlo, nhi, t);
    if ( 0==k, return(0) );
    t = 16;
    while ( floor(t*x)<k,  t*=2 );
    nlo = 1;  nhi = t;
    while ( nlo!=nhi,
        t = floor( (nlo+nhi)/2 );
        if ( floor(t*x) < k,  nlo=t+1,  nhi=t);
    );
    if ( floor(nhi*x) == k,  return(nhi),  return (0));
}
```

```
g=(sqrt(5)+1)/2
for(k=1,65,if(bts(g,k),print1("1"),print1("0")));print();
  10110101101101011010110110101101101011010110110101101011011010110
```

The algorithm is extremely fast, computing 1000 bits starting from position 1,000,000,000,000 takes less than a second:

```
g=(sqrt(5)+1)/2
dd=10^12;  /* digits starting at position dd... */
for(k=dd,dd+1000,if(bts(g,k),print1("1"),print1("0")));print();
  11011010110110101101011011010110101101101011010110101101101011010110
  [--snip--]
  ***   last result computed in 236 ms.
```

However, the connection between the sequence of lowest Fibonacci bits and the rabbit constants allows even more. Subtracting the Fibonacci numbers > 1 until zero or one is reached, gives the complement of the rabbit sequence:

```
fpn=999;
vpv=vector(fpn, j, fibonacci(j+2)); /* vpv=[2,3,5,8,...] */
t=vpv[length(vpv)];
log(t)/log(10)
  208.8471...  /* ok for range up to >10^200 (!) */
```

```
flb(x)=
{ /* return the lowest bit of the Fibonacci representation */
    local(k, t);
    k=bsearchgeq(x, vpv);
    while ( k>0,
        t = vpv[k];
        if (x>=t,  x-=t);
        k-- );
    return ( x );
}
```

```
dd=0;
for(k=dd,dd+40,t=flb(k);print1(1-t))
```

```
     10110101101101011010110110101101101011010
/* 0.10110101101101011010110110101101101011010  rabbit constant */
```

The routine `bsearchgeq()` does a binary search (see section 3.2 on page 111) that is greater or equal to the element sought:

```
bsearchgeq(x, v)=
{ /* return index of first element in v[] that is >=x, 0 if x>max(v[])  */
    local(nlo, nhi, t);
    nlo = 1;  nhi = length(v);
    while ( nlo!=nhi,
        t = floor( (nlo+nhi)/2 );
        if ( v[t] < x,  nlo=t+1,  nhi=t);
    );
    if ( v[nhi] >= x,  return(nhi),  return (0));
}
```

We compute the first 1000 bits starting from position 10^{100}:

```
dd=10^100-1;
for (k=dd, dd+1000, t=flb(k); print1(1-t))
  11010110110101101011011010110110101101011011010
  [--snip--]
  ***   last result computed in 1,305 ms.
```

24.12 An iteration related to the Lucas numbers

Substitution rules as for the Fibonacci system, but the symbols are A and B: (A → B) and (B → BA). The evolution is:

```
L0 = A
L1 = B
L2 = BA
L3 = BAB
L4 = BABBA
L5 = BABBABAB
L6 = BABBABABBABBA
L7 = BABBABABBABBABABBABAB
L -->BABBABABBABBABABBABABBABBABABBABABBABAB...
```

Now identify A with 001 and B with 1100:

```
L0 = 100
L1 = 1100
L2 = 1100100
L3 = 11001001100
L4 = 1100100110011001100
L5 = 1100100110011001100110011001100100110011001100
L6 = 1100100110011001100110011001100100110011001100100110011001100100110011001100
L7 = 1100100110011001100110011001100100110011001100100110011001100100110011001100100110011001100
L -->1100100110011001100110011001100100110011001100100110011001100100110011001100100110011001100110...
```

Define the function $L(y)$ by the iteration

$$L_1 = 1 \qquad R_1 = 1+y \qquad l_1 = y \qquad r_1 = y^3 \tag{24.127a}$$

$$l_{n+1} = r_n = y^{L_n} \tag{24.127b}$$

$$r_{n+1} = r_n\, l_n = y^{L_{n+1}} \tag{24.127c}$$

$$L_{n+1} = R_n \tag{24.127d}$$

$$R_{n+1} = R_n + r_{n+1}\, L_n = R_n + y^{L_{n+1}}\, L_n \quad \to L(y) \tag{24.127e}$$

Here L_n denotes the n-th Lucas number:

```
n   0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
Ln  2  1  3  4  7  11  18  29  47  76  123  199  322  521  843  1364
```

After the n-th step the series in y is correct up to order $L_{n+2}-1$. So the order of convergence is again $\frac{\sqrt{5}+1}{2} \approx 1.6180$.

The implementation of the iteration for $L(y)$ in pari/gp is:

```
fl(y, N=17)=
{
    local(t, yl, yr, L, R, Lp, Rp);
    /* correct up to order lucas(N+2)-1 */
```

```
    L=1;  R=1+y;   yl=y;  yr=y^3;
    for(k=2, N,
      t=yr;  yr*=yl;  yl=t;
      Lp=R; Rp=R+yr*L;  L=Lp;  R=Rp;
    );
    return( R )
}
```

As Taylor series:

$$L(y) \quad = \quad 1 + y + y^4 + y^7 + y^8 + y^{11} + y^{12} + y^{15} + y^{18} + y^{19} + y^{22} + y^{25} + y^{26} + y^{29} + \ldots \qquad (24.128)$$

Define the *Lucas constant* $L := L(1/2)/2$ as

$$\begin{aligned} L &= 0.787493848777364657776791839898022666300486017427068415 7\ldots \qquad (24.129)\\ [\text{base } 2] &= 0.11001001100110010011001001100110011001001100110010011 0\ldots \\ [\text{CF}] &= [0, 1, 3, 1, 2, 2, 1, 1, 24, 1, 4, 2, 1, 1, 1, 767, 2, 1, 1, 1, 10922, 23, 1, \\ &\quad 22369620, 2, 1, 1, 1, 52776558133247, 2, 1, 1, 1, 3148244321913096809130, \\ &\quad 23, 1, 443075998594971957634602353426781524, \ldots] \end{aligned}$$

If the initialization (relations 24.127a) of R_1 is changed to $R_1 = 1 + y + y^2$ the iteration computes $\frac{1}{1-y}$.

Finally, as the recursion for the Lucas numbers is the same as for the Fibonacci numbers,

$$L(y) \quad = \quad A[1, 1 + y, y, y^3] \qquad (24.130)$$

24.13 Iterations related to the Pell numbers

Replacement rules: simultaneously replace all zeros by one ($0 \to 1$) and all ones by one-one-zero ($1 \to 110$).

```
B0 =  0
B1 =  1
B2 =  110
B3 =  1101101
B4 =  110110111101101110
B5 =  1101101110110111011011101101110110111011011101101101
B --> 1101101110110111011011101101110110111011011101101110110111011011101101110...
```

Now the construction is $B_n = B_{n-1}.B_{n-1}.B_{n-2}$. The length of the n-th string is

$$p_n \quad = \quad 1,\ 1,\ 3,\ 7,\ 17,\ 41,\ 99,\ 239,\ \ldots \quad p_k = 2p_{k-1} + p_{k-2}$$

This is sequence A001333 of [OEIS], the numerators of the continued fraction of $\sqrt{2}$. The sequence `B` of zeros and ones is A080764 of [OEIS]. The Pell numbers are the first differences (and the denominators of the continued fraction of $\sqrt{2}$), sequence A000129:

$$0,\ 1,\ 2,\ 5,\ 12,\ 29,\ 70,\ 169,\ 408,\ 985,\ 2378,\ 5741,\ \ldots$$

Now define the function $B(y)$ by the iteration

$$\begin{aligned} L_0 &= 1 \qquad R_0 \ = \ 1 + y \qquad l_0 \ = \ y \qquad r_0 \ = \ y && (24.131a)\\ l_{n+1} &= r_n && (24.131b)\\ r_{n+1} &= r_n^2\, l_n && (24.131c)\\ L_{n+1} &= R_n && (24.131d)\\ R_{n+1} &= R_n + r_{n+1}\, R_n + r_{n+1}^2\, L_n \quad \to B(y) && (24.131e) \end{aligned}$$

After the n-th step the series in y is correct up to order p_n. That is, the order of convergence is $\sqrt{2} + 1 \approx 2.4142$. Implementation in pari/gp:

```
fb(y, N=8)=
{
    local(t, yr, yl, L, R, Lp, Rp);
    L=1;  R=1+y;  yl=y;  yr=y;
    for(k=1,N,
      t=yr; yr*=yr*yl; yl=t;
      Lp=R; Rp=R+yr*R+yr^2*L; L=Lp; R=Rp;
    );
    return( R )
}
```

One finds

$$B(y) \;=\; 1 + y + y^3 + y^4 + y^6 + y^7 + y^8 + y^{10} + y^{11} + y^{13} + y^{14} + y^{15} + y^{17} + y^{18} + y^{20} + \ldots \tag{24.132}$$

Define the *Pell constant* B as $B = \frac{1}{2}\,B(\frac{1}{2})$, then

$$\begin{aligned} B &= 0.858267656461002055792260308433375148664905190083506778 6\ldots && (24.133)\\ [\text{base } 2] &= 0.1101101110110111011011011101101110110110111011011011011\ldots \\ [\text{CF}] &= [0, 1, 6, 18, 1032, 16777344, 288230376151842816, \\ &\quad 139379657490816394634598239204272161737932 8, \ldots] \end{aligned}$$

For the terms of the continued fraction we note

```
                                                  6 ==   2^2 +  2^1
                                                 18 ==   2^4 +  2^1
                                               1032 ==  2^10 +  2^3
                                           16777344 ==  2^24 +  2^7
                                  288230376151842816 ==  2^58 + 2^17
    139379657490816394634598239204272161737932 8 == 2^140 + 2^41
```

Define $B[L_0, R_0, l_0, r_0]$ to be the generalized function of the initialization parameters, then

$$B(y) \;=\; B[1,\, 1+y,\, y,\, y] \tag{24.134}$$

and

$$B[\alpha L,\, \beta R,\, l,\, r] \;=\; \alpha\, B[L,\, 0,\, l,\, r] + \beta\, B[0,\, R,\, l,\, r] \tag{24.135}$$

24.13.1 Pell palindromes

Define the function $P(y)$ by

$$\begin{aligned} L_0 &= 1 \qquad R_0 \;=\; 1 + y^2 \qquad l_0 \;=\; y \qquad r_0 \;=\; y && (24.136a)\\ l_{n+1} &= r_n && (24.136b)\\ r_{n+1} &= r_n^2\, l_n && (24.136c)\\ L_{n+1} &= R_n && (24.136d)\\ R_{n+1} &= R_n + r_{n+1}\, L_n + r_{n+1}\, l_{n+1}\, R_n \quad \to P(y) && (24.136e) \end{aligned}$$

Note that R_0 is a palindrome and in relation 24.136e the combination of the parts gives a palindrome. For $R_0 = 1 + y + y^2$ the iteration computes $\frac{1}{1-y}$.

Define the *Pell palindromic constant* as $P = P(1/2)/2$, then

$$\begin{aligned} P &= 0.732160433063532837164590187177304465727298658960411239 0\ldots && (24.137)\\ [\text{base } 2] &= 0.1011101101101110110111011011101101110110111011011011011\ldots \\ [\text{CF}] &= [0, 1, 2, 1, 2, 1, 3, 17, 1, 7, 2063, 1, 63, 268437503, 1, 8191, 590295810358974087167, \\ &\quad 1, 1073741823, 37414441915671114706014331717595874884227743665356 7, 1, \ldots] \end{aligned}$$

By construction, the binary expansion is a palindrome up to lengths $1, 3, 7, 17, 41, 99, 239, \ldots$.

The sequence of zeros and ones in the binary expansion of P is entry A104521 of [OEIS]. It can be obtained by the replacement rules $(0 \to 1)$ and $(1 \to 101)$:

```
P0 =  0
P1 =  1
P2 =  101
P3 =  1011101
P4 =  10111011011011101
P5 =  1011101101101110111011011101110110111011011011101101
P --> 10111011011011101110111011011101101110111011011011101101...
```

24.13.2 Pell representation

Same construction as for the Fibonacci numbers with maximal digit two.

```
012345678901234567890123456789012345678901234567890123456789012345678901234567890
0         1         2         3         4         5         6         7         8

.....................................................................................  239
.....................................iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii  99
.................iiiiiiiiiiiiiiiii22222222.................iiiiiiiiiiiiiiii22222222  41
.....iiiiiiiii1111111111111111222222.....iiiiiiiii1111111111111111222222...iiiiiii  17
.....1111111222.....1111111222.....1111111222.....1111111222.....1111111222.....  7
...1112...1112...1112...1112...1112...1112...1112...1112...1112...1112...1112...  3
.12.12..12.12..12.12.12..12.12..12.12.12..12.12..12.12..12.12.12..12.12..12.12.12  1

.121232123234323412323432343454345234345412323432343454345234345434545654563454 56  S

.1212.212.2.1.2.112.2.1.2.1.121.122.1.12112.2.1.2.1.121.122.1.121.1212.212..1212.  S % 3
.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.  S % 2
```

Figure 24.6: Pell representations of the numbers $0 \ldots 80$. A dot is used for zero. The three lower lines are the sum of digits and the sum of digits modulo three and two.

The function $S(y)$ that has Taylor series coefficients corresponding to the sum of Pell digits

$$S(y) \quad = \quad 0 + 1y + 2y^2 + 1y^3 + 2y^4 + 3y^5 + 2y^6 + 1y^7 + 2y^8 + 3y^9 + 2y^{10} + 3y^{11} + 4y^{12} + \ldots \qquad (24.138)$$

can be computed via the iteration (see section 24.7 on page 646)

$$\begin{aligned}
L_0 &= 0 \qquad R_0 = 0 + y + 2\,y^2 \qquad l_0 = y \qquad r_0 = y && (24.139a)\\
A_0 &= 1 \qquad B_0 = 1 + y + y^2 && (24.139b)\\
l_{n+1} &= r_n && (24.139c)\\
r_{n+1} &= r_n^2\, l_n && (24.139d)\\
L_{n+1} &= R_n && (24.139e)\\
R_{n+1} &= R_n + r_{n+1}\,(R_n + B_n) + r_{n+1}^2\,(L_n + 2\,A_n) \quad \to S(y) && (24.139f)\\
A_{n+1} &= B_n && (24.139g)\\
B_{n+1} &= B_n + r_{n+1}\,B_n + r_{n+1}^2\,A_n && (24.139h)
\end{aligned}$$

Implementation in pari/gp:

```
fs(y, N=8)=
{
    local(t, yr, yl, L, R, Lp, Rp, Li, Ri);
    L =  0;  R = 0+y+2*y^2;
    Li = 1;  Ri = 1+y+y^2;
    yl = y;  yr = y;
    for(k=1,N,
      t=yr; yr*=yr*yl; yl=t;
      Lp=R;  Rp=R+yr*(R+Ri)+yr^2*(L+2*Li);  L=Lp; R=Rp;
      Lp=Ri; Rp=Ri+yr*Ri+yr^2*Li;   Li=Lp; Ri=Rp;
    );
    return( R )
}
```

The series coefficients grow slowly, so the first few of them can nicely be displayed as

$$S\left(\frac{1}{10}\right) \quad = \quad 0.12123212323432341232343234345434523434541232343234345434523434543454565456345456\ldots \qquad (24.140)$$

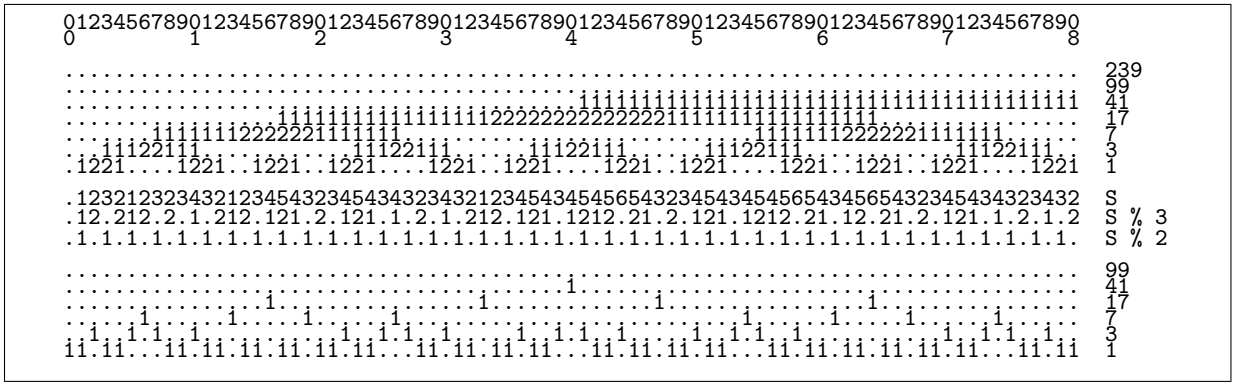


Figure 24.7: A Gray code for Pell representations. A dot is used for zero. The three following lines are the sum of digits and the sum of digits modulo three and two. The sequence is $0, 1, 2, 5, 4, 3, 6, 13, 10, 11, 12, 9, 8, 7, 14 \ldots$, the difference between successive elements is a Pell number. The lowest block gives the Pell representations of the (absolute) differences, the *Pell ruler function*.

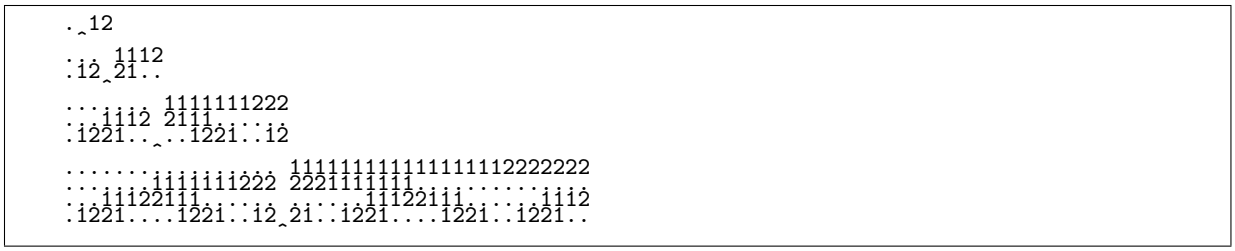


Figure 24.8: Construction for a Gray code for Pell representations.

24.13.3 Pell Gray code

Figure 24.7 gives a Gray code for the Pell representations. The Gray code can be constructed recursively as shown in figure 24.8. In the algorithm each block is split into a left and a right part (indicated by the '^'. The next block is created by appending to the current block its reverse with ones on top and appending the left part with twos on top. The iteration can actually be started with a block of a single zero (the left part being also a single zero). This is done in the following algorithm.

$$F_0 = 0 \quad F'_0 = 0 \quad B_0 = 0 \quad B'_0 = 0 \tag{24.141a}$$
$$I_0 = 1 \quad I'_0 = 1 \quad Y_0 = y \quad Y'_0 = y \tag{24.141b}$$
$$b_n = 4^{n-1} \quad c_n = 2\,b_n \tag{24.141c}$$
$$F_{n+1} = (F_n \quad) + Y_n\,(B_n + b_n\,I_n) + \; Y_n^2\,(F'_n + c_n\,I'_n) \quad \to G_b(y) \tag{24.141d}$$
$$B_{n+1} = (B'_n + c_n\,I'_n) + Y'_n\,(F_n + b_n\,I_n) + Y'_n\,Y_n\,(B_n \quad) \tag{24.141e}$$
$$I_{n+1} = I_n + Y_n\,I_n + Y_n^2\,I'_n \tag{24.141f}$$
$$Y_{n+1} = Y_n^2\,Y'_n \tag{24.141g}$$
$$F'_{n+1} = F_n \quad B'_{n+1} = B_n \tag{24.141h}$$
$$I'_{n+1} = I_n \quad Y'_{n+1} = Y_n \tag{24.141i}$$

Implementation in pari/gp:

```
pgr(y, N=11)= /* Pell Gray code */
{
    local(iir, iil, yl, yr, Fl, F, Bl, B, b, c);
    local(t, tf, tb);
    /* correct up to order pell(N+1)-1 */
    F=0;  Fl=0;   B=0;  Bl=0;
    iil=1;  iir=1;  yl=y;  yr=y;
```

```
    for(k=1, N,
      b = 4^(k-1);  c = 2*b;  /* b = pell(k);*/
      tf = (F         ) +  yr * (B + b*iir) + yr*yr * (Fl + c*iil);
      tb = (Bl + c*iil) +  yl * (F + b*iir) + yl*yr * (B         );
      Fl = F;   Bl = B;
      F = tf;   B = tb;
      t = iir;  iir += yr*(iir + yr*iil);  iil = t;
      t = yr;  yr *= (yr*yl);  yl = t;
    );
    return( F )
}
```

It is instructive to look at the variables in the first few steps of the iteration:

```
 ----- k = 0
yl = (y)  yr = (y)
iir = 1
iil = 1
F = 0
B = 0
 ----- k = 1    b=1
yl = (y)  yr = (y^3)
iir = (y^2 + y + 1)
iil = 1
F = (2 y^2 + y)
B = (y + 2)
 ----- k = 2    b=4
yl = (y^3)  yr = (y^7)
iir = (y^6 + y^5 + y^4 + y^3 + y^2 + y + 1)
iil = (y^2 + y + 1)
F = (8 y^6 + 4 y^5 + 5 y^4 + 6 y^3 + 2 y^2 + y)
B = (y^5 + 2 y^4 + 6 y^3 + 5 y^2 + 4 y + 8)
 ----- k = 3   b=16
yl = (y^7)  yr = (y^17)
iir = (y^16 + y^15 + y^14 + y^13 + y^12 + y^11 + y^10 + y^9
      + y^8 + y^7 + y^6 + y^5 + y^4 + y^3 + y^2 + y + 1)
iil = (y^6 + y^5 + y^4 + y^3 + y^2 + y + 1)
F = (34 y^16 + 33 y^15 + 32 y^14
      + 16 y^13 + 17 y^12 + 18 y^11 + 22 y^10 + 21 y^9 + 20 y^8 + 24 y^7
      + 8 y^6 + 4 y^5 + 5 y^4 + 6 y^3 + 2 y^2 + y)
B = (y^15 + 2 y^14 + 6 y^13 + 5 y^12 + 4 y^11 + 8 y^10
      + 24 y^9 + 20 y^8 + 21 y^7 + 22 y^6 + 18 y^5 + 17 y^4 + 16 y^3
      + 32 y^2 + 33 y + 34)
```

The Taylor series for $G_2(y)$ is

$$\begin{aligned} G_2(y) &= 0 + 1y + 2y^2 + 6y^3 + 5y^4 + 4y^5 + 8y^6 + 24y^7 + 20y^8 + 21y^9 + 22y^{10} + 18y^{11} + \\ &\quad +17y^{12} + 16y^{13} + 32y^{14} + 33y^{15} + 34y^{16} + 98y^{17} + 97y^{18} + 96y^{19} + 80y^{20} + 81y^{21} + \\ &\quad +82y^{22} + 86y^{23} + 85y^{24} + 84y^{25} + 88y^{26} + 72y^{27} + 68y^{28} + 69y^{29} + 70y^{30} + 66y^{31} + \ldots \end{aligned} \tag{24.142}$$

The coefficients corresponds to the Pell representations interpreted as binary numbers, each Pell-digit occupying two bits (figure 24.7 on the preceding page).

When relation 24.141c on the previous page is changed to $b_n = P_n$ (indicated in the code, the function can be defined as `pell(k)=if(k<=1, 1, return(2*pell(k-1)+pell(k-2)));`) then the function $G_P(y)$ which has the Pell Gray code sequence as coefficients is computed:

$$\begin{aligned} G_P(y) &= 0 + 1y + 2y^2 + 5y^3 + 4y^4 + 3y^5 + 6y^6 + 13y^7 + 10y^8 + 11y^9 + 12y^{10} + 9y^{11} + \\ &\quad +8y^{12} + 7y^{13} + 14y^{14} + 15y^{15} + 16y^{16} + 33y^{17} + 32y^{18} + 31y^{19} + 24y^{20} + 25y^{21} + \\ &\quad +26y^{22} + 29y^{23} + 28y^{24} + 27y^{25} + 30y^{26} + 23y^{27} + 20y^{28} + 21y^{29} + 22y^{30} + 19y^{31} + \ldots \end{aligned} \tag{24.143}$$

Section 5.18.2 on page 204 gives a recursive algorithm to compute the words of the Pell Gray code.

```
 0     . . . .      .......      0
 1     . . . 1      .......1     1
 2     . . . 2      ......1.     2
 3     . . 1 2      .....11.     6
 4     . . 1 1      .....1.1     5
 5     . . 1 .      .....1..     4
 6     . . 2 .      ....1...     8
 7     . 1 2 .      ...11...    24
 8     . 1 1 .      ...1.1..    20
 9     . 1 1 1      ...1.1.1    21
10     . 1 1 2      ...1.11.    22
11     . 1 . 2      ...1..1.    18
12     . 1 . 1      ...1...1    17
13     . 1 . .      ...1....    16
14     . 2 . .      ..1.....    32
15     . 2 . 1      ..1....1    33
16     . 2 . 2      ..1...1.    34
17     1 2 . 2      .11...1.    98
18     1 2 . 1      .11....1    97
19     1 2 . .      .11.....    96
20     1 1 . .      .1.1....    80
21     1 1 . 1      .1.1...1    81
22     1 1 . 2      .1.1..1.    82
23     1 1 1 2      .1.1.11.    86
24     1 1 1 1      .1.1.1.1    85
25     1 1 1 .      .1.1.1..    84
26     1 1 2 .      .1.11...    88
27     1 . 2 .      .1..1...    72
28     1 . 1 .      .1...1..    68
29     1 . 1 1      .1...1.1    69
30     1 . 1 2      .1...11.    70
31     1 . . 2      .1....1.    66
32     1 . . 1      .1.....1    65
33     1 . . .      .1......    64
34     2 . . .      1.......   128
35     2 . . 1      1......1   129
```

Figure 24.9: Gray code through the Pell representations and its binary equivalent. Dots are used for zeros.

Define the *Pell Gray code constant* as

$$\begin{aligned} G_P &= G_P\left(\frac{1}{2}\right) && \text{(24.144a)} \\ &= 2.24556734836507219572095657243899881986749522914019 2012\ldots && \text{(24.144b)} \\ [\text{base } 2] &= 10.00111110110111011000000001110010001100011000010000111\ldots && \text{(24.144c)} \\ [\text{CF}] &= [2, 4, 13, 1, 5, 1, 1, 1, 27, 1, 9, 1, 3, 8, 1, 2, 1, 1, 3, 14, 1, 8, 1, 1, 6, 3, 1, \\ &\quad 1, 1, 2, 1, 7, 210, 1, 1, 3, 2, 1, 1, 10, 1, 1, 6, 1, 1, 2, 1, 2, 1, 4, 6, 12, 1, \ldots] \end{aligned}$$

Setting $b_n = 1,\ c_n = 2$ in the algorithm gives a function whose series coefficients are the sum of Pell Gray code digits:

$$G_{[1,2]}\left(\frac{1}{10}\right) = 0.123212323432123454323454343234321234543454565432345434 5\ldots \quad (24.145)$$

Using $b_n = 1,\ c_n = 0$ counts the ones in the Pell Gray code:

$$G_{[1,0]}\left(\frac{1}{10}\right) = 0.101210123212101012123234321232121010121012321212323432 3\ldots \quad (24.146)$$

while $b_n = 0,\ c_n = 1$ counts the twos:

$$G_{[0,1]}\left(\frac{1}{10}\right) = 0.011001100110011221100110011001100112211221122110011001100 110\ldots \quad (24.147)$$

The continued fraction of this constant has very large terms:

```
[0, 90, 1, 8, 1, 5501100164, 8, 5, 3, 2, 19, 2, 1, 2, 2, 1, 1, 5, 1,
54, 6, 1, 5, 22, 2, 6, 2, 2, 1, 22445, 1, 45, 2, 2, 5, 1, 5, 1, 8,
5446094555544609444716727 4, 1, 5, 2, 2, 7, 1, 1, 1, 2, 1, 27, 2, 2,
1, 17, 1, 1, 1, 1, 4, 2, 1, 4, 3, 3, 3, 1, 3, 1, 2, 1, 1, 29, 1, ...]
```

Finally, $(1-y)\,G_P(y)$ gives the (signed) Pell ruler function, the first 100 series coefficients are:

```
0 +1 +1 +3 -1 -1 +3 +7 -3 +1 +1 -3 -1 -1 +7 +1 +1 +17 -1 -1 -7 +1 +1
+3 -1 -1 +3 -7 -3 +1 +1 -3 -1 -1 +17 +1 +1 +3 -1 -1 +3 +41 -3 +1 +1
-3 -1 -1 -17 +1 +1 +3 -1 -1 +3 +7 -3 +1 +1 -3 -1 -1 +7 +1 +1 -17 -1
-1 -7 +1 +1 +3 -1 -1 +3 -7 -3 +1 +1 -3 -1 -1 +41 +1 +1 +3 -1 -1 +3
+7 -3 +1 +1 -3 -1 -1 +7 +1 +1 +99 -1
```

Appendix A

Machine used for benchmarking

The machine used for performance measurements is a AMD64 (Athlon64) clocked at 2.2 GHz with dual channel double data rate (DDR) clocked at 200 MHz ('800 MHz'). It has 512 kB (16-way associative) second level cache and separate first level caches for data and instructions, each 64 kB (and 2-way associative). Cache lines are 16 bytes (128 bits). The memory controller is integrated in the CPU.

The CPU has 16 general purpose (64 bit) registers that are addressable as byte, 16 bit word, 32 bit word, or 64 bit (full) word. These are used for integer operations and for passing integer function arguments. There are 16 (128 bit, SSE) registers that are used for floating point operations and for passing floating point function arguments. The SSE registers are SIMD registers. Additionally, there are 8 (legacy, x87) FPU registers.

The performance-wise interesting information reported by the CPUID instruction is:

```
Vendor: AuthenticAMD
Name: AMD Athlon(tm) 64 Processor 3400+
  Family: 15,  Model: 15,  Stepping: 0
Level 1 cache (data):  64 kB,  2-way associative.
  64 bytes per line,  lines per tag: 1.
Level 1 cache (instr):  64 kB,  2-way associative.
  64 bytes per line,  lines per tag: 1.
Level 2 cache:  512 kB,  16-way associative
  64 bytes per line,  lines per tag: 1.

Max virtual addr width: 48
Max physical addr width: 40
Features:
  lm: Long Mode (64-bit mode)
  mtrr: Memory Type Range Registers
  tsc: Time Stamp Counter
  fpu: x87 FPU
  3dnow: AMD 3DNow! instructions
  3dnowext: AMD Extensions to 3DNow!
  mmx: Multimedia Extensions
  mmxext: AMD Extensions to MMX
  sse: Streaming SIMD Extensions
  sse2: Streaming SIMD Extensions-2
  cmov: CMOV instruction (plus FPU FCMOVCC and FCOMI)
  cx8: CMPXCHG8 instruction
  clflush: CLFLUSH instruction
  fxsr: FXSAVE and FXRSTOR instructions
```

Special instructions as SIMD, prefetch and non-temporal moves are not used unless explicitly noted.

See [x86perf] for a comparison of instruction latencies and throughput for various x86 CPU cores. You do want to study the cited document *before* buying a x86-based system.

Appendix B

The pseudo language Sprache

Many algorithms in this book are given in a pseudo language called Sprache. Sprache is meant to be immediately understandable for everyone who ever had contact with programming languages like C, FORTRAN, Pascal or Algol. Sprache is hopefully self explanatory. The intention of using Sprache instead of completely relying an mathematical formulas (like tensor formalism) or algorithm description by words is to minimize the work it takes to translate the given algorithm to one's favorite programming language. It should be mere syntax adaptation.

By the way, 'Sprache' is the German word for language.

```
// a comment:
// comments are useful.

// assignment:
t := 2.71

// parallel assignment:
{s, t, u} := {5, 6, 7}
// same as:
s := 5
t := 6
u := 7

{s, t} := {s+t, s-t}
// same as (avoiding the temporary):
temp := s + t
t := s - t
s := temp

// if conditional:
if a==b then  a:=3

// with block
if a>=3 then
{
    // do something ...
}

// a function returns a value:
function plus_three(x)
{
    return  x + 3
}

// a procedure works on data:
procedure increment_copy(f[],g[],n)
// real f[0..n-1] input
// real g[0..n-1] result
{
    for k:=0 to n-1
    {
        g[k] := f[k] + 1
    }
}

// for loop with stepsize:
```

```
for i:=0 to n step 2  // i:=0,2,4,6,...
{
    // do something
}

// for loop with multiplication:
for i:=1 to 32 mul_step 2
{
    print i, ", "
}
```

will print `1, 2, 4, 8, 16, 32,`

```
// for loop with division:
for i:=32 to 8 div_step 2
{
    print i, ", "
}
```

will print `32, 16, 8,`

```
// while loop:
i:=5
while i>0
{
    // do something 5 times...
    i := i - 1
}
```

The usage of `foreach` emphasizes that no particular order is needed in the array access (so parallelization is possible):

```
procedure has_element(f[],x)
{
    foreach t in f[]
    {
        if t==x then return TRUE
    }
    return FALSE
}
```

Emphasize type and range of arrays:

```
real     a[0..n-1],    // has n elements (floating point reals)
complex  b[0..2**n-1]  // has 2**n elements (floating point complex)
mod_type m[729..1728]  // has 1000 elements (modular integers)
integer  i[]           // has ? elements (integers)
```

Arithmetical operators: `+, -, *, /, %` and `**` for powering. Arithmetical functions: `min(), max(), gcd(), lcm(), ...`

Mathematical functions: `sqr(), sqrt(), pow(), exp(), log(), sin(), cos(), tan(), asin(), acos(), atan(), ...`

Bitwise operators: `~, &, |, ^` for bit-wise complement, AND, OR, XOR, respectively. Bit shift operators: `A<<3` shifts (the integer) `A` 3 bits to the left `A>>1` shifts `A` 1 bits to the right.

Comparison operators: `==, !=, <, > ,<=, >=`

There is no operator '`=`' in Sprache, only '`==`' (for testing equality) and '`:=`' (assignment operator).

A well-known constant: `PI` $= 3.14159265\ldots$

The complex square root of minus one in the upper half plane: `I` $= \sqrt{-1}$

Boolean values: `TRUE` and `FALSE`

Logical operators: `NOT, AND, OR, XOR`

Modular arithmetic: `x := a * b mod m` shall do what it says, `i := a**(-1) mod m` shall set `i` to the modular inverse of `a`.

Bibliography

— Textbooks and Thesis —

[HaMaFu] Milton Abramowitz, Irene A. Stegun (Eds.): **Handbook of Mathematical Functions**, National Bureau of Standards, 1964, third printing, 1965.

[AMD22007] Advanced Micro Devices (AMD) Inc.: **AMD Athlon Processor, x86 code optimization guide**, Publication #22007, Revision H, June-2000. Online at `http://www.amd.com/`.

[AMD24594] Advanced Micro Devices (AMD) Inc.: **AMD64 Architecture Programmer's Manual. Volume 3: General-Purpose and System Instructions**, Publication #24594, Revision 3.11, February-2005. Online at `http://www.amd.com/`.

[AMD25112] Advanced Micro Devices (AMD) Inc.: **Software Optimization Guide for AMD64 Processors**, Publication #25112, Revision 3.06, September-2005. Online at `http://www.amd.com/`.

[PiUnl] Jörg Arndt, Christoph Haenel: **Pi – Unleashed**, Springer-Verlag, 2000. (translation of: **Pi. Algorithmen, Computer, Arithmetik**, 1998).

[RecrTN] Albert H. Beiler: **Recreations in the Theory of Numbers**, Dover Publications, 1964.

[PiAGM] J. M. Borwein, P. B. Borwein: **Pi and the AGM**, Wiley, 1987.

[Fact] J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman, S. S. Wagstaff Jr.: **Factorizations of $b^n \pm 1$ $b = 2, 3, 5, 6, 10, 11$ up to high powers**, Contemporary Mathematics, Volume 22, second edition, American Mathematical Society, 1988. Online at `http://www.ams.org/`.

[Cohen] H. Cohen: **A Course in Computational Algebraic Number Theory**, Springer-Verlag, 1993. Online errata list at `http://www.math.u-bordeaux.fr/~cohen/`.

[Int2Alg] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: **Introduction to Algorithms**, MIT Press, second edition, 2001.

[Num] H.-D. Ebbinghaus, H. Hermes, F. Hirzebruch, M. Koecher, K. Mainzer, J. Neukirch, A. Prestel, R. Remmert: **Zahlen**, second edition, (english translation: **Numbers**), Springer-Verlag, 1988.

[Finch] Steven R. Finch: **Mathematical Constants**, Cambridge University Press, 2003.

[GaoNB] Shuhong Gao: **Normal Bases over Finite Fields**, Thesis, University of Waterloo, Ontario, Canada, 1993.

[MCA] Joachim von zur Gathen, Jürgen Gerhard: **Modern Computer Algebra**, Cambridge University Press, second edition, 2003.

[ConcMath] R. L. Graham, D. E. Knuth, O. Patashnik: **Concrete Mathematics**, second printing, Addison-Wesley, New York, 1988.

[Hous] A. S. Householder: **The Numerical Treatment of a Single Nonlinear Equation**, McGraw-Hill, 1970.

[IreRos] Kenneth Ireland, Michael Rosen: **A Classical Introduction to Modern Number Theory**, second edition, Springer-Verlag, 1990.

[King] Louis V. King: **On the Direct Numerical Calculation of Elliptic Functions and Integrals**, Cambridge University Press, 1924.

[DEK1] D. E. Knuth: **The Art of Computer Programming**, third edition, Volume 1: Fundamental Algorithms, Addison-Wesley, 1997. Online errata list at `http://www-cs-staff.stanford.edu/~knuth/`.

[DEK2] D. E. Knuth: **The Art of Computer Programming**, third edition, Volume 2: Seminumerical Algorithms, Addison-Wesley, 1997. Online errata list at `http://www-cs-staff.stanford.edu/~knuth/`.

[DEK3] D. E. Knuth: **The Art of Computer Programming**, second edition, Volume 3: Sorting and Searching, Addison-Wesley, 1997. Online errata list at `http://www-cs-staff.stanford.edu/~knuth/`.

[DEK4] D. E. Knuth: **The Art of Computer Programming**, pre-fascicles for Volume 4. Online at `http://www-cs-staff.stanford.edu/~knuth/`.

[NTC] Ramanujachary Kumanduri, Cristina Romero: **Number theory with computer applications**, Prentice-Hall, 1998.

[Intr2FF] Rudolf Lidl, Harald Niederreiter: **Introduction to finite fields and their applications**, Cambridge University Press, revised edition, 1994.

[CComb] J. H. van Lint, R. M. Wilson: **A Course in Combinatorics**, Cambridge University Press, 1992.

[Lipson] J. D. Lipson: **Elements of algebra and algebraic computing**, Addison-Wesley, 1981.

[CVL] Charles Van Loan: **Computational Frameworks for the Fast Fourier Transform**, SIAM, 1992.

[LoWaa] Lisa Lorentzen, Haakon Waadeland: **Continued Fractions and Applications**, North-Holland, 1992.

[Muller] Jean-Michel Muller: **Elementary Functions: algorithms and implementation**, Birkhäuser, 1997.

[Nussb] H. J. Nussbaumer: **Fast Fourier Transform and Convolution Algorithms**, second edition, Springer-Verlag, 1982.

[Olds] C. D. Olds: **Continued Fractions**, The Mathematical Association of America, 1963.

[HR] Hans Riesel: **Prime Numbers and Computer Methods for Factorization**, Birkhäuser, 1985.

[Sedge] Robert Sedgewick: **Algorithms in C++. Parts 1-4: Fundamentals, Data Structures, Sorting, Searching**, third edition, Addison-Wesley, 1998.

[Sedge5] Robert Sedgewick: **Algorithms in C++. Part 5: Graph Algorithms**, third edition, Addison-Wesley, 2001.

[SteinNT] William Stein: **An Explicit Approach to Elementary Number Theory**, Harvard University, Fall 2001. Online at `http://modular.fas.harvard.edu:9000/`.

[Traub] J. F. Traub: **Iterative Methods for the Solution of Equations**, Chelsea, 1982, second edition (text identical to the 1964 edition).

[ModAnl] E. T. Whittaker, G. N. Watson: **A Course of Modern Analysis**, Cambridge University Press, fourth edition, 1927, reprinted 1990.

[WilfGF] Herbert. S. Wilf: **generatingfunctionology**, second edition, Academic Press, 1992.

— Software & Online Resources —

[apfloat] Mikko Tommila: **apfloat, A High Performance Arbitrary Precision Arithmetic Package**, 1996. Online at `http://www.apfloat.org/`.

[arXiv] **arXiv.org e-Print archive**, Online at `http://arxiv.org/`. List of mirrors at `http://arxiv.org/servers.html`.

[CiteSeer] **CiteSeer**, Scientific Literature Digital Library. Online at `http://citeseer.ist.psu.edu/cs/`.

[CePA] **Cryptology ePrint Archive**, International Association for Cryptologic Research. Online at `http://eprint.iacr.org/`.

[DOAJ] **Directory of Open Access Journals**, Online at `http://www.doaj.org/`.

[ECCC] **Electronic Colloquium on Computational Complexity**, Cryptology ePrint Archive. Online at `http://www.eccc.uni-trier.de/eccc/`.

[Emacs] Richard Stallman et.al.: **The GNU Emacs**. Online at `http://www.gnu.org/`.

[FXT] Jörg Arndt: **FXT, a library of algorithms**. Online at `http://www.jjj.de/fxt/`.

[GCC] The Free Software Foundation (FSF): **GCC, the GNU Compiler Collection**. Online at `http://www.gnu.org/gcc/`.

[HAKMEM] M. Beeler, R. W. Gosper, R. Schroeppel: **HAKMEM**, MIT AI Memo 239, 29-February-1972. Retyped and converted to html by Henry Baker, April-1995. Online at `http://home.pipeline.com/~hbaker1/`.

[hfloat] Jörg Arndt: **hfloat, a library for high precision computations**. Online at `http://www.jjj.de/hfloat/`.

[MathWorld] Erik Weisstein: **MathWorld**. Online at `http://mathworld.wolfram.com/`.

[OEIS] N. J. A. Sloane: **The On-Line Encyclopedia of Integer Sequences**. Online at `http://www.research.att.com/~njas/sequences/`.

[Pari] The PARI Group (C. Batut, K. Belabas, D. Bernardi, H. Cohen, M. Olivier et.al.): **PARI/GP**. Online at `http://pari.math.u-bordeaux.fr/`.

[Proth] Wilfried Keller: **Fermat factoring status**. Online at `http://www.prothsearch.net/fermat.html`.

[x86perf] Torbjörn Granlund: **Instruction latencies and throughput for AMD and Intel x86 processors**, Version of 6-September-2005. Online at `http://swox.com/doc/x86-timing.pdf`.

— Papers —

[1] Manindra Agrawal, Neeraj Kayal, Nitin Saxena: **PRIMES is in P**, Annals of Math., Vol.160, no.2, pp.781-793, September-2004. Online at `http://www.math.princeton.edu/~annals/`.

[2] Jean-Paul Allouche, Jeffrey Shallit: **The ubiquitous Prouhet-Thue-Morse sequence**, In: C. Ding, T. Helleseth, H. Niederreiter (Eds.), Sequences and Their Applications: Proceedings of SETA'98, pp.1-16, Springer-Verlag, 1999, Online at `http://www.cs.uwaterloo.ca/~shallit/papers.html`.

[3] Jean-Paul Allouche, Michael Cosnard: **The Komornik-Loreti constant is transcendental**, Amer. Math. Monthly 107, pp.448-449, 2000. Online at `http://www.lri.fr/~allouche/`.

[4] Ray Andraka: **A survey of CORDIC algorithms for FPGA based computers**, In: Proceedings of the Sixth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '98), ACM/SIGDA, ACM, pp.191-200, 1998. Online at `http://www.andraka.com/papers.htm`.

[5] Elisardo Antelo, Tomás Lang, Javier D. Bruguera: **Very-High Radix Circular CORDIC: vectoring and Unified Rotation/Vectoring**, IEEE Transactions on Computers, vol.49, no.7, pp.727-739, 2000. Online at `http://www.ac.usc.es/en/articulos.html`.

[6] Francisco Argüello, Emilio L. Zapata: **Fast Cosine Transform based on the Successive Doubling Method**, Electronics Letters, Vol.26, No.19, pp.1616-1618, September-1990. Online at `http://citeseer.ist.psu.edu/400246.html`.

[7] A. O. Atkin, D. J. Bernstein: **Prime sieves using binary quadratic forms**, Mathematics of Computation 73, pp.1023-1030, 2004. Online at `http://cr.yp.to/djb.html`.

[8] John C. Baez: **The Octonions**, Bull. Amer. Math. Soc. 39, pp.145-205, 2002. Online at `http://math.ucr.edu/home/baez/octonions/`.

[9] Henry G. Baker: **Complex Gaussian Integers for 'Gaussian Graphics'**, ACM Sigplan Notices, 28,11, pp.22-27, November-1993. Online at `http://home.pipeline.com/~hbaker1/`.

[10] D. H. Bailey: **FFTs in External or Hierarchical Memory**, Journal of Supercomputing, vol.4, no.1, pp.23-35, March-1990. Online at `http://crd.lbl.gov/~dhbailey/dhbpapers/`.

[11] D. H. Bailey, P. N. Swarztrauber: **The Fractional Fourier Transform and Applications**, SIAM Review, vol.33, no.3, pp.389-404, September-1991. Online at `http://crd.lbl.gov/~dhbailey/dhbpapers/`.

[12] D. H. Bailey: **On the Computational Cost of FFT-based Linear Convolutions**, 7-June-1996. Online at `http://crd.lbl.gov/~dhbailey/dhbpapers/`.

[13] D. H. Bailey, R. E. Crandall: **On the Random Character of Fundamental Constant Expansions**, Experimental Mathematics, vol.10, no.2, pp.175-190, June-2001. Online at `http://crd.lbl.gov/~dhbailey/dhbpapers/`.

[14] D. H. Bailey, J. M. Borwein, P. B. Borwein and S. Plouffe: **The Quest for Pi**, Mathematical Intelligencer, vol.19, no.1, pp.50-57, January-1997. Online at `http://crd.lbl.gov/~dhbailey/dhbpapers/`.

[15] Robert Baillie, S. Wagstaff, Jr: **Lucas Pseudoprimes**. Mathematics of Computation, Vol 35, 152, October-1980.

[16] F. L. Bauer: **An Infinite Product for Square-Rooting with Cubic Convergence**, The Mathematical Intelligencer 20, pp.12-13, 1998.

[17] Bruce C. Berndt: **Flowers which we cannot yet see growing in Ramanujan's garden of hypergeometric series, elliptic functions and q's**, In: Special Functions 2000: Current Perspective and Future Directions (J. Bustoz, M. E. H. Ismail, and S. K. Suslov, eds.), Kluwer, Dordrecht, pp.61-85, 2001. Online at `http://www.math.uiuc.edu/~berndt/publications.html`

[18] Daniel J. Bernstein: **Computing Logarithm Intervals with the Arithmetic-Geometric-Mean Iteration**, draft, 2003. Online at `http://cr.yp.to/djb.html`.

[19] Daniel J. Bernstein: **Multidigit Multiplication for Mathematicians**, August-2001. Online at `http://cr.yp.to/djb.html`.

[20] Daniel J. Bernstein: **Pippenger's exponentiation algorithm**, draft, December-2001. Online at `http://cr.yp.to/djb.html`.

[21] Daniel J. Bernstein: **Fast multiplication and its applications**, draft, January-2003. Online at `http://cr.yp.to/djb.html`.

[22] Pedro Berrizbeitia, T. G. Berry: **Generalized Strong Pseudoprime tests and applications**, J. Symbolic Computation, No.11, 1999.

[23] Ian F. Blake, Shuhong Gao, Robert J. Lambert: **Construction and Distribution Problems for Irreducible Trinomials over Finite Fields**, In: Applications of finite fields, (D. Gollmann, ed.), pp.19-32, 1996. Online at `http://www.math.clemson.edu/faculty/Gao/`.

[24] J. M. Borwein, P. B. Borwein: **Cubic and higher order algorithms for π**, Canad. Math. Bull. Vol.27 (4), pp.436-443, 1984.

[25] J. Borwein, P. Borwein, Frank Garvan: **Hypergeometric Analogues of the Arithmetic-Geometric Mean Iteration**, Constr. Approx. 9, 1993.

[26] J. M. Borwein, P. B. Borwein, F. G. Garvan: **Some cubic modular identities of Ramanujan**, Transactions A.M.S. 343, pp.35-47, 1994. Online at `http://citeseer.ist.psu.edu/borwein94some.html`.

[27] J. M. Borwein, F. G. Garvan: **Approximations to π via the Dedekind eta function**, 27-March-1996. Online at `http://citeseer.ist.psu.edu/article/borwein96approximations.html`.

[28] Jonathan M. Borwein, Petr Lisonek, John A. Macdonald: **Arithmetic-Geometric Means Revisited**, Maple-Tech, vol.4, no.1, pp.20-27, 1997. Online at `http://citeseer.ist.psu.edu/3981.html`

[29] Florian Braun, Marcel Waldvogel: **Fast Incremental CRC Updates for IP over ATM Networks**, In: Proceedings of 2001 IEEE Workshop on High Performance Switching and Routing (HPSR 2001). Online at `http://citeseer.ist.psu.edu/braun01fast.html`.

[30] R. P. Brent: **Fast multiple-precision evaluation of elementary functions**, J. ACM (23), pp.242-251, 1976. Online at `http://wwwmaths.anu.edu.au/~brent/pub/pubsall.html`.

[31] Richard P. Brent, Alfred J. van der Poorten, Herman J. J. te Riele: **A comparative Study of Algorithms for Computing Continued Fractions of Algebraic Numbers**, In: Lecture Notes in Computer Science, Vol.1122, Springer-Verlag, Berlin, pp.35-47, 1996. Online at `http://wwwmaths.anu.edu.au/~brent/pub/pubsall.html`.

[32] R. P. Brent: **Six lectures on Algorithms, Lecture 1: Fast Algorithms for Elementary Functions, π, and γ**, 1999. Online at `http://web.comlab.ox.ac.uk/oucl/people/richard.brent.html`.

[33] K. Cattel, S. Zhang, X. Sun, M. Serra, J. C. Muzio, D. M. Miller: **One-Dimensional Linear Hybrid Cellular Automata: Their Synthesis, Properties, and Applications in VLSI Testing**, Online at `http://citeseer.ist.psu.edu/260274.html`.

[34] Kevin Cattel, Shujian Zhang: **Minimal Cost One-Dimensional Linear Hybrid Cellular Automata of Degree Through 500**, Journal of Electronic Testing: Theory and Applications, pp.255-258, 1995. Online at `http://citeseer.ist.psu.edu/cattell95minimal.html`

[35] Kevin Cattell, Frank Ruskey, Joe Sawada, C. Robert Miers, Micaela Serra: **Generating Unlabeled Necklaces and Irreducible Polynomials over GF(2)**, Department of Computer Science, University of Victoria, Canada, 1998. Online at `http://citeseer.ist.psu.edu/cattell98generating.html`.

[36] D. V. Chudnovsky, G. V. Chudnovsky: **Classical constants and functions: computations and continued fraction expansions**, In: Number Theory: New York seminar 1989-1990, Springer-Verlag, 1991.

[37] H. Cohen, F. R. Villegas, D. Zagier: **Convergence acceleration of alternating series**, Experiment. Math. 9, iss.1, pp.3-12, 2000. Online at `http://citeseer.ist.psu.edu/cohen99convergence.html`.

[38] Henri Cohen: **Analysis of the flexible window powering method**, Preprint. Online at `http://www.math.u-bordeaux.fr/~cohen/`.

[39] R. Crandall, B. Fagin: **Discrete Weighted Transforms and Large Integer Arithmetic**, Math. Comp., Vol.62, pp.305-324, 1994.

[40] Geoffrey Dixon: **Division Algebras, Galois Fields, Quadratic Residues**, arXiv:hep-th/9302113, 23-February-1993. Online at `http://arxiv.org/abs/hep-th/9302113`.

[41] P. Duhamel, H. Hollmann: **Split radix FFT algorithm**, Electronis Letters 20, pp.14-16, 1984.

[42] P. Duhamel: **Implementation of 'split-radix' FFT algorithms for complex, real and real-symmetric data**, IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-34 pp.285-295, 1986.

[43] Shalosh B.EKHAD (Doron Zeilberger): **Forty "Strange" Computer-Discovered Hypergeometric Series Evaluations**, 12-October-2004. Online at `http://www.math.rutgers.edu/~zeilberg/pj.html`.

[44] Shuhong Gao, Daniel Panario: **Tests and Constructions of Irreducible polynomials over Finite Fields**, Foundations of Computational Mathematics (F. Cucker, M. Shub, eds.), Springer, pp.346-361, 1997. Online at `http://www.math.clemson.edu/~sgao/pub.html`.

[45] Frank Garvan: **Cubic modular identities of Ramanujan, hypergeometric functions and analogues of the arithmetic-geometric mean iteration**, Contemporary Mathematics 166, 1993. Online at .

[46] Frank Garvan: **Ramanujan's theories of elliptic functions to alternative bases – a symbolic excursion**, J. Symbolic Computation, 1995. Online at `http://citeseer.ist.psu.edu/garvan95ramanujans.html`.

[47] Daniel M. Gordon: **A survey of fast exponentiation methods**, Journal of Algorithms, Vol.27, No.1, pp.129-146, 1998. Online at `http://citeseer.ist.psu.edu/gordon97survey.html`.

[48] Mark Goresky, Andrew Klapper: **Fibonacci and Galois Representations of Feedback with Carry Shift Registers**, IEEE Trans. Info. Theory 48, pp.2826-2836, 2000. Online at `http://www.math.ias.edu/~goresky/`.

[49] R. W. Gosper: **Acceleration of series**, MIT AI Memo 304, March-1974. Online at `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-304.pdf`.

[50] Johannes Grabmeier, Alfred Scheerhorn: **Finite Fields in AXIOM**, AXIOM Technical Report No. ATR/5, 1992. Online at `http://citeseer.ist.psu.edu/121340.html`.

[51] R. L. Graham, N. J. A. Sloane: **Anti-Hadamard Matrices**, Linear Algebra and Its Applications, Vol.62, pp.113-137, 1984. Online at `http://www.research.att.com/~njas/doc/pub.html`.

[52] Torbjörn Granlund, Peter L. Montgomery: **Division by Invariant Integers using Multiplication**, SIGPLAN Notices, Vol.29, p.61ff, June-1994. Online at `http://swox.com/~tege/`.

[53] Jon Grantham: **Frobenius Pseudoprimes**, Math. Comp., Vol.70, No.234, pp.873-891, April-2001. Online at `http://www.pseudoprime.com/pseudo.html`.

[54] Andrew Granville: **It is easy to determine whether a given integer is prime**, Bulletin (New Series) of the AMS, Vol.42, No.1, pp.3-38, 2004. Online at `http://www.dms.umontreal.ca/~andrew/AllPublic.html`.

[55] S. Gudvangen: **Practical Applications of Number Theoretic Transforms**, NORSIG-99, Norwegian Signal Processing Symposium, 1999. Online at `http://www.norsig.no/norsig99/`.

[56] Bruno Haible, Thomas Papanikolaou: **Fast multiprecision evaluation of series of rational numbers**. Technical Report No. TI-7/97, 18-March-1997. Online at `http://www.informatik.th-darmstadt.de/TI/Veroeffentlichung/TR/Welcome.html`.

[57] Philip Hawkes, Michael Paddon, Gregory G. Rose: **Musings on the Wang et al. MD5 Collision**, Unpublished note, 13-October-2004. Online at `http://eprint.iacr.org/2004/264`.

[58] Gerben J. Hekstra, Ed F. A. Deprettere: **Floating Point Cordic (extended version)**, Technical Report ET/NT 93.15, Delft University of Technology, 8-March-1993. Online at `http://citeseer.ist.psu.edu/hekstra93floating.html`.

[59] Nicholas J. Higham: **The Matrix Sign Decomposition and its Relation to the Polar Decomposition**, Linear Algebra and Appl., 212/213, pp.3-20, 1994. Online at `http://citeseer.ist.psu.edu/higham94matrix.html`.

[60] Nicholas J. Higham: **Stable Iterations for the Matrix Square Root**, Numerical Algorithms, Vol.15, No.2, pp.227-242, 1997. Online at `http://www.ma.man.ac.uk/~higham/`.

[61] Nicholas J. Higham: **Evaluating Padé Approximants of the Matrix Logarithm**, SIAM J. Matrix Anal. Appl. Vol.22, No.4, pp.1126-1135, 2001. Online at `http://www.ma.man.ac.uk/~higham/`.

[62] Thomas D. Howell, Jean-Claude Lafon: **The Complexity of the Quaternion Product**, Technical Report TR-75-245, Department of Computer Science, Cornell University, Ithaca, NY, June-1975. Online at `http://home.pipeline.com/~hbaker1/`.

[63] T. A. Jenkyns: **Loopless Gray Code Algorithms**, Technical Report CS-95-03, Brock University, Canada, July-1995. Online at `http://citeseer.ist.psu.edu/134169.html`.

[64] J. Johnson, R. W. Johnson, D. Rodriguez, R. Tolimieri: **A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures**, Circuits, Systems, and Signal Processing, Vol.9, pp.449-500, 1990. Online at `http://citeseer.ist.psu.edu/johnson90methodology.html`.

[65] Marc Joye, Sung-Ming Yen: **Optimal Left-to-right Binary Signed-Digit Recoding**, IEEE Transactions on Computers, Vol.49, No.7, pp.740-748, 2000. Online at `http://citeseer.ist.psu.edu/713533.html`.

[66] Bahman Kalantari, Jürgen Gerlach: **Newton's Method and Generation of a Determinantal Family of Iteration Functions**, Technical Report DCS-TR 371, Dept. of Computer Science, Rutgers University, New Brunswick, NJ, 1998. Online at `http://citeseer.ist.psu.edu/kalantari98newtons.html`.

[67] Dan Kalman: **A Singularly Valuable Decomposition: The SVD of a Matrix**, preprint, College Math Journal, Vol.27, No.1, January-1996. Online at `http://www.american.edu/academic.depts/cas/mathstat/People/kalman/pdffiles/index.html`.

[68] Andrew Klapper, Mark Goresky: **Feedback Shift Registers, 2-Adic Span and Combiners With Memory**, Journal of Cryptology, Vol.10, pp.111-147, 1997. Online at `http://www.math.ias.edu/~goresky/EngPubl.html`.

[69] V. Kislenkov, V. Mitrofanov, E. Zima: **How fast can we compute products?**, in ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation, 1999. Online at `http://citeseer.ist.psu.edu/434789.html`.

[70] Wolfram Koepf: **Orthogonal Polynomials and Computer Algebra**, In: R. P. Gilbert et al. (Ed.): Recent Developments in Complex Analysis and Computer Algebra, Kluwer, pp.205-234, 1999. Online at `http://www.mathematik.uni-kassel.de/~koepf/Publikationen/index.html`.

[71] Wolfram Koepf: **Efficient Computation of Chebyshev Polynomials in Computer Algebra**, In: M. Wester (Ed.) Computer Algebra Systems: A Practical Guide. John Wiley, Chichester, pp.79-99, 1999. Online at `http://www.mathematik.uni-kassel.de/~koepf/Publikationen/index.html`.

[72] Susan Landau, Neil Immermann: **The Similarities (and Differences) between Polynomials and Integers**, Theoretic and Algebraic Methods in Computer Science, pp.57-59, 1993. Online at `http://citeseer.ist.psu.edu/landau94similarities.html`.

[73] Benoit Leblanc, Evelyne Lutton, Jean-Paul Allouche: **Inverse problems for finite automata: a solution based on Genetic Algorithms**, Lecture Notes in Computer Science (Artificial evolution, 1997, J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, D. Snyers, eds.), 1363, pp.157-166, 1998. Online at `http://www.lri.fr/~allouche/bibliorecente.html`.

[74] Charles E. Leiserson, Harald Prokop, Keith H. Randall: **Using de Bruijn Sequences to Index a 1 in a Computer Word**, MIT Lab for Computer Science, Cambridge, MA, June-1998. Online at `http://citeseer.ist.psu.edu/leiserson98using.html`.

[75] Arjen K. Lenstra: **Integer Factoring**, Designs, Codes and Cryptogaphy, No.19, pp.101-128, 2000.

[76] Ya Yan Lu: **Computing the Logarithm of a Symmetric Positive Definite Matrix**, Applied Numerical Mathematics: Transactions of IMACS, Vol.26, No.4, pp.483-496, 1998. Online at `http://citeseer.ist.psu.edu/331019.html`.

[77] Robert S. Maier: **A generalization of Euler's Hypergeometric Transform**, arXiv:math.CA/0302084, 8-April-2003. Online at `http://arxiv.org/abs/math.CA/0302084`.

[78] Robert S. Maier: **Algebraic Hypergeometric Transformations of Modular Origin**, arXiv:math.NT/0501425, 25-July-2005. Online at `http://arxiv.org/abs/math.NT/0501425`.

[79] H. Malvar: **Fast computation of the discrete cosine transform through fast Hartley transform**, Electronics Letters, Vol.22, No.7, pp.352-353, 1986.

[80] H. Malvar: **Fast Computation of the discrete cosine transform and the discrete Hartley transform**, IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-35, pp.1484-1485, 1987.

[81] Peter L. Montgomery: **A Survey of Modern Integer Factorization Algorithms**, CWI Quarterly, Vol.7, No.4, pp.337-365, 1994. Online at `http://citeseer.ist.psu.edu/montgomery94survey.html`.

[82] Carl Pomerance, J. L. Selfridge, S. Wagstaff, Jr.: **The Pseudoprimes to** $25 \cdot 10^9$. Mathematics of Computation, Vol 35, 151, pp.1003-1026, July-1980.

[83] Peter John Potts: **Computable Real Arithmetic Using Linear Fractional Transformations**, Dept. of Computing, Imperial College of Science, Technology and Medicine, London, June-1996. Online at `http://www.purplefinder.com/~potts/exact.html`.

[84] Helmut Prodinger: **On Binary Representations of Integers with Digits -1, 0, +1**, INTEGERS Vol.0 , 14-June-2000. Online at `http://math.sun.ac.za/~prodinger/`.

[85] Micheal O. Rabin: **Probabilistic algorithms in finite fields**, Technical Report MIT-LCS-TR-213, Massachusetts Institute of Technology, January-1979. Online at `http://publications.csail.mit.edu/`.

[86] Frank Ruskey, Aaron Williams: **Generating combinations by prefix shifts**, Lecture Notes in Computer Science, 3595, 2005. Extended Abstract for COCOON 2005 online at `http://www.cs.uvic.ca/~ruskey/Publications/`.

[87] Frank Ruskey: **Simple combinatorial Gray codes constructed by reversing sublists**, Department of Computer Science, University of Victoria, Canada, 1993. Online at `http://www.cs.uvic.ca/~ruskey/Publications/`.

[88] Frank Ruskey, Joe Sawada: **An Efficient Algorithm for Generating Necklaces with Fixed Density**, SIAM J. Comput., Vol.29, No.2, pp.671-684, 1999. Online at `http://www.cs.uvic.ca/~ruskey/Publications/`.

[89] Frank Ruskey, Carla Savage: **Gray Codes for Set Partitions and Restricted Growth Tails**, Australasian Journal of Combinatorics 10, pp.85-96, 1994. Online at `http://www.cs.uvic.ca/~ruskey/Publications/`.

[90] Eugene Salamin: **Application of Quaternions to Computation with Rotations**, Working Paper, Stanford AI Lab, 1979. Edited and TeX-formatted by Henry G. Baker, 1995. Online at `http://home.pipeline.com/~hbaker1/`.

[91] Joe Sawada: **Generating Bracelets in Constant Amortized Time**, SIAM J. Comput, Vol.31, No.1, pp.259-268, 2001. Online at `http://citeseer.ist.psu.edu/470511.html`.

[92] Joe Sawada: **A fast algorithm to generate necklaces with fixed content**, Theoretical Computer Science, Vol.301, No.1-3, pp.477-489, May-2003.

[93] Carla Savage: **A Survey of Combinatorial Gray Codes**, SIAM Review, 39, No.4, pp.605-629, 1997. Online at `http://www.csc.ncsu.edu/faculty/savage/papers.html`.

[94] E. Schröder: **On Infinitely Many Algorithms for Solving Equations**, (translation by G. W. Stewart of: **Ueber unendlich viele Algorithmen zur Auflösung der Gleichungen**, which appeared 1870 in the 'Mathematische Annalen'). Online at `ftp://thales.cs.umd.edu/pub/reports/`.

[95] Mohammad A. Shokrollahi, Kai Werther: **Generation of Optimal Bilinear Multiplication Algorithms: Theory and Implementation**, Universität Bonn, December-1992.

[96] P. Soderquist, M. Leeser: **An Area/Performance Comparison of Subtractive and Multiplicative Divide/Square Root Implementations**, Cornell School of Electrical Engineering. in Proceedings 12th IEEE Symposium on Computer Arithmetic, S. Knowles and W. H. McAllister (eds.), July-1995. Online at `http://citeseer.ist.psu.edu/soderquist95areaperformance.html`.

[97] Jerome A. Solinas: **Generalized Mersenne Numbers**, Technical report CORR-39, Dept. of C&O, University of Waterloo, 1999. Online at `http://www.cacr.math.uwaterloo.ca/`.

[98] Hong-Yeop Song: **Examples and Constructions of Hadamard Matrices**, Dept. of Electical and Electronics Engineering, Yonsei University, Korea, June-2002. Online at `http://calliope.uwaterloo.ca/~ggong/710T4/Song-lecture.ps`.

[99] H. Sorensen, D. Jones, M. Heideman, C. Burrus: **Real-Valued Fast Fourier Transform algorithms**, IEEE Trans. on Acoustics, Speech and Signal Processing, Vol ASSP-35, No.6, pp.849-863, 1987.

[100] Ralf Stephan: **Divide-and-conquer generating functions. Part I. Elementary sequences**, 2003. Online at `http://arxiv.org/abs/math.CO/0307027`.

[101] Paul N. Swarztrauber: **Bluestein's FFT for Arbitrary N on the Hypercube**, Parallel Computing, Vol.17, pp.607-617, 1991. Online at `http://www.cisl.ucar.edu/css/staff/pauls/papers/bluestein/bluestein.html`.

[102] M. A. Thornton, D. M. Miller, R. Drechsler: **Transformations Amongst the Walsh, Haar, Arithmetic and Reed-Muller Spectral Domains**, International Workshop on Applications of the Reed-Muller Expansion in Circuit Design (RMW), pp.215-225, 2001. Online at `http://engr.smu.edu/~mitch/publications.html`.

[103] San C. Vo: **A Survey of Elliptic Cryptosystems, Part I: Introductory**, NASA Advanced Supercomputing (NAS) Division, August-2003. Online at `http://www.nas.nasa.gov/News/Techreports/2003/2003.html`.

[104] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, Hongbo Yu: **Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD**, revised version 17-August-2004. Online at `http://eprint.iacr.org/2004/`.

[105] Zhong-De Wang: **New algorithm for the slant transform**, IEEE Transactions Pattern Anal. Mach. Intell. PAMI-4, No.5, pp.551-555, September-1982.

[106] André Weimerskirch, Christoph Paar: **Generalizations of the Karatsuba Algorithm for Polynomial Multiplication**, March-2002. Online at `http://citeseer.ist.psu.edu/571363.html`.

[107] André Weimerskirch, Christoph Paar: **Generalizations of the Karatsuba Algorithm for Efficient Implementations**, Technical Report, corrected version, 2003. Online at `http://www.crypto.ruhr-uni-bochum.de/en_publications.html`.

[108] Paweł Zieliński, Krystyna Ziętak: **The Polar Decomposition – Properties, Applications and Algorithms**, Annals of the Polish Mathematical Society, 38, 1995. Online at `http://citeseer.ist.psu.edu/zielinski95polar.html`.

— How to cite this document —

[109] Jörg Arndt: **Algorithms for programmers (working title)**, draft version of 2005-November-20. Online at `http://www.jjj.de/fxt/`.

Index

Symbols

∗, cyclic correlation 289
9^{9^9}, computation 293
⊛, cyclic convolution 285
$\circledast_{ac}$, acyclic convolution 288
$\mathcal{W}_v$ [ ], weighted transform 293

A

`ac_gray_delta()` 242
acyclic (or: linear) convolution 288
addition, modulo m 519
additive inverse, modulo m 522
adjacency matrix, of a graph 234
adjacent changes, Gray code 240
adjacent, nodes in a graph 234
AGM (arithmetic geometric mean) 429
AGM, 4-th order variant 430
algebra 514
`all_compositions_minchange (C++ class)` 175
`all_irredpol (C++ class)` 596
alternating permutations 200
`apply_permutation()` 99, 106
arbitrary length FFT 299
arc, of a graph 233
arithmetic geometric mean (AGM) 429
asymptotics, of an algorithm 389

B

base field 553
basis functions, Reed-Muller transform 328
Bell numbers 125, 170
Ben-Or irreducibility test 583
Berlekamp's Q-matrix algorithm 598
Bhaskara equation 560
big endian 3
binary bracelets 123
binary exponentiation 395
binary finite fields 553, 606
binary GCD algorithm 522
binary necklaces 123
binary powering 395
binary relation 121
binary search 111
bit-array 145
bit-reversal, permutation 79
`bit_cyclic_period()` 26
`bit_fibgray (C++ class)` 66
`bit_necklace (C++ class)` 212
`bit_rotate_sgn()` 45
`bit_subset (C++ class)` 59
`bitarray (C++ class)` 146
`bitpol_factor()` 602
`bitpol_mult()` 78
`bitpol_normal_q()` 616
`bitpol_refine_factors()` 600
`bitpol_sreduce()` 602
`bitpolmod_mult()` 78
bitrev permutation 79
bsearch 111
`bsearch()` 111
`bsearch_approx()` 112
`bsearch_ge()` 112

C

C++ class
- `GF2n`, 608, 619
- `all_compositions_minchange`, 175
- `all_irredpol`, 596
- `bit_fibgray`, 66
- `bit_necklace`, 212
- `bit_subset`, 59
- `bitarray`, 146
- `catalan`, 162
- `comb_rec`, 159
- `combination_altminchange`, 156
- `combination_colex`, 155
- `combination_lex`, 153
- `combination_minchange`, 156
- `combination_revdoor`, 156
- `composition_chase`, 175
- `composition_colex`, 174
- `coroutine`, 150

– `crc32`, 72
– `crc64`, 70, 250
– `cycles`, 102
– `debruijn`, 182
– `digraph_paths`, 235
– `digraph`, 234
– `fcsr`, 224
– `ksubset_gray`, 186
– `ksubset_near_perfect`, 186
– `ksubset_rec`, 183
– `lfsr`, 218
– `lyndon_gray`, 248
– `matrix`, 307
– `mixedradix_lex`, 187
– `mod`, 380, 557
– `necklace2bitpol`, 595
– `paren`, 160
– `partition`, 165
– `pcrc64`, 73
– `perm_derange`, 197
– `perm_lex_rec`, 190
– `perm_lex`, 192
– `perm_minchange`, 195
– `perm_restrpref`, 200
– `perm_rev`, 193
– `perm_star`, 199
– `perm_trotter`, 197
– `prime_string`, 182
– `priority_queue`, 144
– `queue`, 138
– `ringbuffer`, 137
– `set_partition`, 171
– `string_subst`, 208
– `subset_debruijn`, 182
– `subset_gray_delta`, 179
– `subset_gray`, 180
– `subset_lex`, 177
– `subset_monotone`, 181
– `tcrc64`, 72
C2RFT *see* real FFT
C2RFT (complex to real FT) 275
canonical sequence 99
Carmichael numbers 538
carry, in multiplication 391
CAT *see* constant amortized time
`catalan (C++ class)` 162
Catalan constant 487
Catalan numbers 161, 421
Cayley numbers 515
Cayley-Dickson construction 515
characteristic polynomial
– of a matrix, 615
– of a recurrence relation, 461
Chebyshev polynomials
– and product for square root, 479
– and recurrence for subsequences, 467
– as hypergeometric functions, 505
– definition, 471
– with accelerated summation, 487
Chinese remainder theorem 527
chirp z-transform 298
circuit, in a graph 233
co-lexicographic order, for combinations 155
`comb_rec (C++ class)` 159
`combination_altminchange (C++ class)` 156
`combination_colex (C++ class)` 155
`combination_lex (C++ class)` 153
`combination_minchange (C++ class)` 156
`combination_revdoor (C++ class)` 156
companion matrix 462, 615
complete graph 235
composite, modulus 531
composition, of permutations 101
`composition_chase (C++ class)` 175
`composition_colex (C++ class)` 174
concave 126
conference matrix 336
conjugates, of an element in $GF(2^n)$ 611
connected, permutations 200
constant
– Catalan, 487
– CORDIC scaling, 457, 460
– decimal weighted Fibonacci repr., 663
– Fibonacci parity, 661
– Gray code, 649
– Komornik-Loreti, 635
– Lucas, 666
– parity number, 633
– Pell, 667
– Pell Gray code, 671
– Pell palindromic, 667
– period-doubling, 642
– rabbit, 659
– revbin, 648
– Roth's, 637
– ruler, 640
– sum of Gray code digits, 650
– sum-of-digits, 646
– Thue, 637
– weighted sum of Gray code digits, 653
– weighted sum-of-digits, 648
constant amortized time 67, 203
convergent 489
convex 126
convexity 126
convolution
– acyclic (or: linear), 288
– and multiplication, 391

– by FFT, without revbin permutations, 287
– by FHT, 367
– cyclic, 285
– cyclic, by FHT, 367
– dyadic, 317
– exact, 385
– half cyclic, 295
– linear, 288
– mass storage, 292
– negacyclic, 294, 370
– right-angle, 294
– skew circular, 294
– weighted, 294
Cooley-Tukey, FFT algorithm 256
coprime 531
coprime, $\gcd(a, b) = 1$ 522
`copy_reverse_0()` 275
CORDIC algorithms 456
`coroutine (C++ class)` 150
coroutines 149
`cos_rot()` 374
cosine transform (DCT) 374
cosine, CORDIC algorithm 456
counting bits in a binary word 18
counting sort 128
coupled iteration, for the square root 401
CPU instructions, often missed 77
CRC (cyclic redundancy check) 70
`crc32 (C++ class)` 72
`crc64 (C++ class)` 70, 250
CRT, Chinese Remainder Theorem 527
cube root extraction 401
cycle, in a graph 233
`cycles (C++ class)` 102
cycles, of a permutation 101
cyclic convolution 285
cyclic convolution, by FFT 287
cyclic permutation 108
cyclic redundancy check 70
cyclic, ring 532
cyclotomic polynomial 596

D

DCT (discrete cosine transform) 374
DCT via FHT 374
De Bruijn graph 237
De Bruijn sequence 182, 221, 237
`debruijn (C++ class)` 182
decimal weighted Fibonacci repr. constant 663
decimation in time, FFT algorithm 256
delta squared process 427
deque 140
derangement 100, 200
DFT (discrete Fourier transform) 253
DIF (decimation in frequency) 259
digraph 233
`digraph (C++ class)` 234
`digraph::sort_edges()` 244
`digraph_paths (C++ class)` 235
`digraph_paths::next_path()` 236
`digraph_paths::print_turns()` 244
directed graphs 233
discrete cosine transform, inverse (IDCT) 374
discrete Fourier transform 253, 256
discrete sine transform 375
DIT (decimation in time) 256
division, CORDIC algorithm 458
division, using only multiplication 399
double buffer, for mass storage FFT 293
double-ended queue (deque) 140
dragon curve sequence 651
DST (discrete sine transform) 375
`dsth()` 375
dyadic convolution 317
`dyadic_convolution()` 317

E

edge, of a graph 233
element of order n 379
equivalence classes 121
equivalence relation 121
`equivalence_classes()` 122
Eratosthenes, prime sieve 525
Euclidean algorithm 522
Euler numbers 201
Euler's totient function 531
exact convolution 385
exact division 52
exp, iteration for 435
exponent, of a group 530
exponentiation 395
exponentiation, modulo m 522
extension field 553, 605
external, algorithms 291

F

fast Fourier transform 254
fast Hartley transform 357
FCSR (feedback carry shift register) 224
`fcsr (C++ class)` 224
feedback carry shift register 224
Fermat numbers 544
Fermat primes 536
FFT
– as polynomial evaluation, 392
– radix-2 DIF, 260
– radix-2 DIT, 258
– radix-4 DIF, 268

– radix-4 DIT, 265
– split-radix DIF, 271
FFT (fast Fourier transform) 254
FFT caching 397
`fft_complex_convolution()` 287
`fft_dif4l()` 269
`fft_dit4_core_p1()` 266
FHT
– convolution by, 367
– DIF step, 361
– DIF, recursive, 361
– DIT, recursive, 358
– radix-2 DIF, 362
– radix-2 DIT, 359
– radix-2 DIT step, 358
– shift operator, 358
FHT (fast Hartley transform) 357
`fht_dif2()` 362
`fht_dif_core()` 262
`fht_real_complex_fft()` 365
Fibbinary numbers 56, 662
Fibonacci numbers 660
Fibonacci parity 661
Fibonacci parity constant 661
Fibonacci representation 661
Fibonacci setup, of a shift register 220
Fibonacci words, Gray code 203
FIFO (first-in, first-out) 138
finite field 553
finite-state machine 147
fixed points, of lex-order words 60
FKM algorithm 211
four step FFT 283
Fourier shift operator 258
Fourier transform 253, 255
fractional Fourier transform 299
FT (Fourier transform) 253
full path, in a graph 233

G

Galois Field 605
Galois setup, of a shift register 220
Gaussian normal basis 621
GCD, computation 522
generalized Mersenne primes 524
generator, in $GF(2^n)$ 607
generator, modulo p 530
generator, program producing programs 375
`GF2n (C++ class)` 608, 619
`GF2n::init()` 619
`gf2n_fast_trace()` 607
`gf2n_order()` 608
Golay-Rudin-Shapiro sequence 37, 637
Goldschmidt algorithm 412
Gray code
– for Fibonacci words, 203
– for Lyndon words, 245
– for Pell words, 204, 669
– for signed binary words, 204
– single track, 245
Gray code constant 649
Gray code, of a binary word 33
green code 37
`green_permute()` 96
ground field 605
GRS sequence 637
`grs_negate()` 309

H

`haar()` 343
`haar_inplace()` 344
`haar_rev_nn()` 351
Hadamard matrix 334, 516
half cyclic convolution 295
Hamiltonian cycle 233
Hanoi, towers of, puzzle 642
Hartley shift 358
Hartley transform *see* FHT
hash 69
heap 142
Hilbert curve
– by string substitution, 209
– finite state machine for, 147
– function encoding it, 653
– Gray code from, 206
– moves, 47
– turns, 656
`hilbert_gray_code()` 206
HT (Hartley transform), definition 357
hyperbolic sine and cosine, by CORDIC 458
hypercomplex numbers 515
hypergeometric functions 496

I

IDCT (inverse discrete cosine transform) 374
IDCT by FHT 374
IDST (inverse discrete sine transform) 375
`idsth()` 375
indecomposable, permutations 200
integer sequence
– Beatty seq. with Φ, 664
– Catalan numbers, 161, 421
– Feigenbaum symbolic seq., 642
– Fibbinary numbers, 56, 662
– Fibonacci numbers, 660
– fixed points in lex-rev seq., 60
– Golay-Rudin-Shapiro seq., 37, 637
– Kronecker symbols $\left(\frac{-1}{n}\right)$, 652

– Mephisto Waltz seq., 636
– Moser – De Bruijn sequence, 55
– number of XYZ, *see* number of, XYZ
– paper folding seq., 651
– paper folding seq., signed, 652
– period-doubling seq., 642
– primes with primitive root 2, 224, 589
– primitive trinomials, 592
– rabbit seq., 659
– Thue-Morse seq., 35, 305, 633
interpolation binary search 112
inverse cosine transform (IDCT) 374
inverse cube root, iteration for 401
inverse discrete sine transform (IDST) 375
inverse modulo m 380
inverse root extraction 404
inverse root, iteration for 405
inverse square root, iteration for 400
inverse, additive, modulo m 522
inverse, in $GF(Q)$ 606
inverse, multiplicative, modulo m 522
inverse, of a permutation 101
`inverse_haar()` 343
`inverse_haar_inplace()` 345
inversion by exponentiation 535
inversion principle 597
inversion, iteration for 399
invertible, modulo m 522
involution 101, 200
irreducible, polynomial 581
`is_cyclic()` 108
`is_small_prime()` 527
iteration
– for exp, 435
– for inverse cube root, 401
– for inverse root, 405
– for inverse square root, 400
– for inversion, 399
– for log, 431
iteration, for the zero of a function 419

K

Karatsuba multiplication 390
Karatsuba multiplication for polynomials, , 575
Komornik-Loreti constant 635
`kronecker()` 536
Kronecker product 306
`ksubset_gray (C++ class)` 186
`ksubset_near_perfect (C++ class)` 186
`ksubset_rec (C++ class)` 183

L

Lambert series 644
left-to-right powering 396
Legendre symbol 536
Legendre's relation 441
lexicographic order, for combinations 153
lexicographic order, for subsets 177
LFSR (linear feedback shift register) 217
`lfsr (C++ class)` 218
LHCA (linear hybrid cellular automaton) 225
`lhca2poly()` 226
`lhca_next()` 225
LIFO (last-in, first-out) 135
`lin2hilbert()` 147
linear 605
linear convolution 288
linear feedback shift register 217
Linear hybrid cellular automaton 225
little endian 3
localized FHT algorithm 371
localized Walsh transform algorithm 312
log, iteration using exp 431
logarithm, of power series 437
loops, in a graph 233
Lucas constant 666
Lucas numbers 665
Lucas pseudoprimes 550
Lucas test 548
Lucas-Lehmer test 547
lucky path, through a graph 245
Lyndon words 211
Lyndon words with fixed content 216
Lyndon words with fixed density 216
`lyndon_gray (C++ class)` 248

M

m-sequence 221, 334
Möbius function 597
Möbius inversion principle 597
`make_complement_shift_digraph()` 239
`make_complete_digraph()` 234
`make_oddprime_bitarray()` 527
mass storage convolution 292
mass storage FFT 291
`matrix (C++ class)` 307
matrix Fourier algorithm 283
maximal order, modulo m 529
`maxorder_element_mod()` 533
mean, arithmetic geometric 429
Mephisto Waltz sequence 636
Mersenne primes, and Lyndon words 213
Mersenne primes, generalized 524
MFA, matrix Fourier algorithm 283
minimal polynomial, in $GF(2^n)$ 610
minimal-change order, for combinations 156
`minpoly()` 610
`minweight_lhca_rule()` 225

missing, CPU instructions 77
`mixedradix_lex (C++ class)` 187
`mod (C++ class)` 380, 557
modular adjacent changes, Gray path 240
modular multiplication 520
modular reduction, with Mersenne primes 524
modulus, composite 531
modulus, prime 530
modulus, prime, with NTTs 380
monotone 125
Moser – De Bruijn sequence 55
multi-dimensional Walsh transform 305
multigrades 562
multiplication
– by FFT, 391
– carry, 391
– is convolution, 391
– Karatsuba, 390
– modulo m, 520
– of hypercomplex numbers, 515
– of octonions, 518
– of polynomials, 289
– of quaternions, 517
multiplication matrix, for normal bases 616
multiplication table, of an algebra 514
multiplicative inverse, modulo m 522

N

N-polynomials 616
near-perfect 175
necklace 210
`necklace2bitpol (C++ class)` 595
necklaces with fixed content 216
necklaces with fixed density 216
negacyclic convolution 294, 370
neighbors, of a node in a graph 234
node, of a graph 233
non-first-free, turn in a path 245
non-generous primes 534
noncyclic, ring 532
normal basis, for $GF(2^n)$ 616
normal basis, optimal 619
normal polynomials 616
`normal_mult()` 617
NTT
– radix-2 DIF, 382
– radix-2 DIT, 381
– radix-4, 383
NTT (number theoretic transforms) 379
null-space, of a matrix 616
number of
– alternating permutations, 201
– aperiodic necklaces, 214
– bracelets, 124
– cycles in De Bruijn graph, 239
– De Bruijn sequences, 222
– derangements, 201
– equivalence relations, 125
– fixed density Lyndon words, 216
– fixed density necklaces, 216
– generators modulo n, 534
– indecomposable permutations, 201
– integer partitions, 168
– integers coprime to n, 531
– invertible circulant matrices, 628
– involutions, 201
– irreducible polynomials, 593
– Lyndon words, 214, 594
– m-sequences, 221
– necklaces, 124, 214
– normal polynomials, 629
– parenthesis pairs, 161
– primitive polynomials, 594
– shift register sequences, 221
– sparse signed binary words, 205
– units in $GF(Q)$, 606
– units modulo m, 531
– unlabeled bracelets, 124
– unlabeled necklaces, 124
number theoretic transforms 379

O

octonions 515
one-point iteration 419
optimal normal bases 619
order, of an element modulo m 529
order, of an iteration 419
out of core, algorithms 291

P

Padé approximants, for exp 435
Padé approximants, for log 432
paper folding sequence 651
`paren (C++ class)` 160
parity number 633
parity, of a binary word 34
Parseval's equation 254
`partition (C++ class)` 165
partition, of a set 168
partitioning, of an array 110
partitions, of an integer 164
path, in a graph 233
`pcrc64 (C++ class)` 73
Pell constant 667
Pell Gray code constant 671
Pell palindromic constant 667
Pell ruler function 669
Pell words, Gray code 204

Pepin's test 544
period, of a polynomial 581
period-doubling constant 642
period-doubling sequence 642
`perm_derange (C++ class)` 197
`perm_lex (C++ class)` 192
`perm_lex_rec (C++ class)` 190
`perm_minchange (C++ class)` 195
`perm_restrpref (C++ class)` 200
`perm_rev (C++ class)` 193
`perm_star (C++ class)` 199
`perm_trotter (C++ class)` 197
permutation, inverse of 101
permutation, random 107
phi function, number thoeretic 531
pink air algorithm 19
Pocklington-Lehmer test, for primality 546
polar decomposition, of a matrix 410
polynomial multiplication 289
powering 395
powering, modulo m 522
Pratt's certificate of primality 542
`prev_lexrev()` 317
prime modulus, with NTTs 380
prime, modulus 530
`prime_string (C++ class)` 182
primitive n-th root 379
primitive r-th root of unity 529
primitive elements 530
primitive root 299, 379, 530
primitive root, finding 530
primitive, polynomial 582
`print_cycles()` 102
priority queue 143
`priority_queue (C++ class)` 144
Proth's theorem 544
Prouhet-Thue-Morse constant 633
pseudo-inverse, of a matrix 411
pseudoprime to base a 538

Q

Q-matrix 598
quadratic residues modulo p 535
quadruple reversion trick 86
quantization 120
`quantize()` 120
quaternions 515
queue 138
`queue (C++ class)` 138
quicksort 110

R

R2CFT *see* real FFT
R2CFT (real to complex FT) 275
rabbit constant 659
rabbit sequence 659
Rabin's algorithm, to test irreducibility 584
Rabin-Miller test 539
Rader's algorithm 300
radix permutation 83
`radix_sort()` 130
random permutation 107
re-orthogonalization, of a matrix 408
real FFT
  – by FHT, 365
  – split-radix algorithm, 279
  – with wrap routines, 277
recurrence relation 461
Reed-Muller transform, basis functions 328
restricted growth string 162, 172
revbin constant 648
revbin permutation 79
revbin permutation, and convolution by FFT 287
`revbin_permute()` 81
`revbin_permute0()` 83
`reverse_0()` 275
reversed zip permutation 89
RGS *see* restricted growth string
right-angle convolution 294
right-to-left powering 395
ring buffer 137
`ringbuffer (C++ class)` 137
root extraction 404
root, primitive 379
rotate, by triple reversion 85
`rotate_left()` 85
`rotate_sgn()` 330
row-column algorithm 282
ruler constant 640

S

scalar multiplication 605
sedenions 515
selection sort 109
semi-continuous Fourier transform 255
sequence *see* integer sequence
sequency 320
set partition 168
`set_partition (C++ class)` 171
`setup_q_matrix()` 599
shift operator, for Fourier transform 258
shift register sequence 217
shift, for FHT 358
shift-and-add algorithms 451
sieve of Eratosthenes 525
sign decomposition, of a matrix 411
sign, of the Fourier transform 254
signed binary representation 55

signed binary words, Gray code 204
simple, path in a graph 233
sine transform (DST) 375
sine transform, inverse (IDST) 375
sine, CORDIC algorithm 456
single track Gray codes 245
singular value decomposition 409
skew circular convolution 294
slant transform 326
slant transform, sequency ordered 327
`slow_convolution()` 285
`slow_ft()` 254
`solve_quadratic()` 612
`solve_reduced_quadratic()` 612
sparse signed binary representation 55
sparse words, bit counting 19
split-radix FFT 270
square modulo p 535
square root, iteration for 400
squarefree factorization 604
SRS (shift register sequence) 217
stack 135
state-engine 147
Stirling numbers of the second kind 170
Stirling set numbers 170
strictly convex 127
strictly monotone 126
`string_subst (C++ class)` 208
strong minimal-change order 164
strong pseudoprime to base a 538
structured primes 524
subfactorial numbers 202
`subset_debruijn (C++ class)` 182
`subset_gray (C++ class)` 180
`subset_gray_delta (C++ class)` 179
`subset_lex (C++ class)` 177
`subset_monotone (C++ class)` 181
subtraction, modulo m 519
sum of Gray code digits constant 650
sum-of-digits constant 646
sum-of-digits test 394

T

`tcrc64 (C++ class)` 72
tensor product 306
Thue constant 637
Thue-Morse sequence 35, 305, 516, 633
TMFA, transposed matrix Fourier algorithm 283
totient function 531
towers of Hanoi 642
trace, of a polynomial 616
trace, of an element in $GF(2^n)$ 606
transformations, of hypergeometric functions 499
transition count, for a Gray code 245
`transpose()` 84, 85
transposed matrix Fourier algorithm 283
trinomials, primitive 592
triple reversion trick 85
type-t Gaussian normal basis 621
type-1 optimal normal basis 620
type-2 optimal normal basis 620

U

unique 117
units, invertible elements 522
unlabeled binary bracelets 123
unzip permutation 88
`unzip_rev()` 89
up-down permutations 200

V

vertex, of a graph 233

W

Walsh transform, multi-dimensional 305
`walsh_pal_basefunc()` 319
`walsh_q1()` 324
`walsh_q2()` 324
`walsh_wak_basefunc()` 305
`walsh_wak_dif2()` 305, 329
`walsh_wak_dit2()` 304, 329
`walsh_wal_basefunc()` 321
`walsh_wal_rev()` 323, 324
`walsh_wal_rev_basefunc()` 324
weight, of binary polynomial 582
weighted convolution 294
weighted sum of Gray code digits constant 653
weighted sum-of-digits constant 648
weighted transform 293
Wieferich primes 534

X

`xor_permute()` 90
`xrevbin()` 47

Z

Z-order 48
z-transform 298
Zeckendorf representation 661
zero divisors, of an algebra 515
zero padding, for linear convolution 288
`zip()` 87
zip permutation 87
`zip_rev()` 89