Modern Computer Arithmetic

Richard P. Brent and Paul Zimmermann

Version 0.2

Preface

This is a book about algorithms for performing arithmetic, and their implementation on modern computers. We are concerned with software more than hardware — we do not cover computer architecture or the design of computer hardware since good books are already available on these topics. Instead we focus on algorithms for efficiently performing arithmetic operations such as addition, multiplication and division, and their connections to topics such as modular arithmetic, greatest common divisors, the Fast Fourier Transform (FFT), and the computation of special functions.

The algorithms that we present are mainly intended for arbitrary-precision arithmetic. That is, they are not limited by the computer wordsize of 32 or 64 bits, only by the memory and time available for the computation. We consider both integer and real (floating-point) computations.

The book is divided into four main chapters, plus an appendix. Chapter 1 covers integer arithmetic. This has, of course, been considered in many other books and papers. However, there has been much recent progress, inspired in part by the application to public key cryptography, so most of the published books are now partly out of date or incomplete. Our aim has been to present the latest developments in a concise manner.

Chapter 2 is concerned with the FFT and modular arithmetic, and their applications to computer arithmetic. We consider different number representations, fast algorithms for multiplication, division and exponentiation, and the use of the Chinese Remainder Theorem (CRT).

Chapter 3 covers floating-point arithmetic. Our concern is with high-precision floating-point arithmetic, implemented in software if the precision provided by the hardware (typically IEEE standard 64-bit arithmetic) is inadequate. The algorithms described in this chapter focus on *correct rounding*, extending the IEEE standard to arbitrary precision.

Chapter 4 deals with the computation, to arbitrary precision, of functions

such as sqrt, exp, ln, sin, cos, and more generally functions defined by power series or continued fractions. We also consider the computation of certain constants, such as π and (Euler's constant) γ. Of course, the computation of special functions is a huge topic so we have had to be selective. In particular, we have concentrated on methods that are efficient and suitable for arbitrary-precision computations.

For details that are omitted we give pointers in the *Notes and References* sections of each chapter, and in the bibliography. Finally, the Appendix contains pointers to implementations, useful web sites, mailing lists, and so on.

The book is intended for anyone interested in the design and implementation of efficient algorithms for computer arithmetic, and more generally efficient numerical algorithms. We did our best to present algorithms that are ready to implement in your favorite language, while keeping a high-level description.

Although the book is not specifically intended as a textbook, it could be used in a graduate course in mathematics or computer science, and for this reason, as well as to cover topics that could not be discussed at length in the text, we have included exercises at the end of each chapter. For solutions to the exercises, please contact the authors.

We thank the French National Institute for Research in Computer Science and Control (INRIA), the Australian National University (ANU), and the Australian Research Council (ARC), for their support. The book could not have been written without the contributions of many friends and colleagues, too numerous to mention here, but acknowledged in the text and in the *Notes and References* sections at the end of each chapter.

Finally, we thank Erin Brent, who first suggested writing the book, and our wives Judy-anne and Marie, for their patience and encouragement.

This is a preliminary version — there are still a few sections to be completed. We welcome comments and corrections. Please send them to either of the authors.

Richard Brent and Paul Zimmermann
MCA@rpbrent.com
Paul.Zimmermann@inria.fr
Canberra and Nancy, June 2008

Contents

Notation

$\mathbb{C}$	set of complex numbers
$\mathbb{N}$	set of natural numbers (nonnegative integers)
$\mathbb{Q}$	set of rational numbers
$\mathbb{R}$	set of real numbers
$\mathbb{Z}$	set of integers
$\mathbb{Z}/n\mathbb{Z}$	ring of residues modulo n
C^n	set of (real or complex) functions with n continuous derivatives in the region of interest
$\Re(z)$	real part of a complex number z
$\Im(z)$	imaginary part of a complex number z
$\bar{z}$	conjugate of the complex number z
$\|z\|$	Euclidean norm of the complex number z
β	"word" base (usually 2^{32} or 2^{64})
n	number of base β digits in integer or in floating-point significand, depending on the context
ε	"machine precision" $\frac{1}{2}\beta^{1-n}$
η	smallest positive subnormal number
$\circ(x)$	rounding of real number x
$\mathrm{ulp}(x)$	for a floating-point number x, one unit in the last place
$M(n)$	time to multiply n-bit integers or polynomials of degree $n-1$, depending on the context
$M(m,n)$	time to multiply an m-bit integer by an n-bit integer
$D(n)$	time to divide a $2n$-bit integer by an n-bit integer
$D(m,n)$	time to divide an m-bit integer by an n-bit integer

$\mathrm{sign}(n)$	$+1$ if $n > 0$, -1 if $n < 0$, and 0 if $n = 0$
$r := a \bmod b$	integer remainder $(0 \le r < b)$
$q := a \operatorname{div} b$	integer quotient $(0 \le a - qb < b)$
$i \wedge j$	bitwise *and* of integers i and j, or logical *and* of two Boolean expressions
$i \oplus j$	bitwise *exclusive-or* of integers i and j
$i \ll k$	integer i multiplied by 2^k
$i \gg k$	quotient of division of integer i by 2^k
$\nu(n)$	2-valuation: largest k such that 2^k divides n $(\nu(0) = \infty)$
$\sigma(e)$	length of the shortest addition chain to compute e
$\phi(n)$	Euler's totient function, $\#\{m : 0 < m \le n \wedge (m, n) = 1\}$
$\deg(A)$	for a polynomial A, the degree of A
$\mathrm{ord}(A)$	for a power series $A = \sum_j a_j z^j$, $\mathrm{ord}(A) = \min\{j : a_j \ne 0\}$ (note the special case $\mathrm{ord}(0) = +\infty$)
log, ln	natural logarithm
$\log_2$, lg	base-2 logarithm
$\mathrm{nbits}(n)$	$\lfloor \lg(n) \rfloor + 1$ if $n > 0$, 0 if $n = 0$
${}^t[a, b]$ or $[a, b]^t$	vector $\begin{pmatrix} a \\ b \end{pmatrix}$
$f(n) = O(g(n))$	$\exists c, n_0$ such that $\lvert f(n) \rvert \le c g(n)$ for all $n \ge n_0$
$f(n) = \Theta(g(n))$	$f(n) = O(g(n))$ and $g(n) = O(f(n))$
$[a, b; c, d]$	2×2 matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$
$xxx.yyy_\rho$	a number $xxx.yyy$ written in base ρ; for example, the decimal number 3.25 is 11.01_2 in binary

Chapter 1

Integer Arithmetic

In this Chapter our main topic is integer arithmetic. However, we shall see that many algorithms for polynomial arithmetic are similar to the corresponding algorithms for integer arithmetic, but simpler due to the lack of carries in polynomial arithmetic. Consider for example addition: the sum of two polynomials of degree n always has degree n at most, whereas the sum of two n-digit integers may have $n + 1$ digits. Thus we often describe algorithms for polynomials as an aid to understanding the corresponding algorithms for integers.

1.1 Representation and Notations

We consider in this Chapter algorithms working on integers. We shall distinguish between the logical — or mathematical — representation of an integer, and its physical representation on a computer. Our algorithms are intended for "large" integers — they are not restricted to integers that can be represented in a single computer word.

Several physical representations are possible. We consider here only the most common one, namely a dense representation in a fixed base. Choose an integral *base* $\beta > 1$. (In case of ambiguity, β will be called the *internal* base.) A positive integer A is represented by the length n and the digits a_i of its base β expansion:

$$A = a_{n-1}\beta^{n-1} + \cdots + a_1\beta + a_0,$$

where $0 \leq a_i \leq \beta - 1$, and a_{n-1} is sometimes assumed to be non-zero. Since the base β is usually fixed in a given program, only the length n and the integers $(a_i)_{0 \leq i < n}$ need to be stored. Some common choices for β are 2^{32} on a 32-bit computer, or 2^{64} on a 64-bit machine; other possible choices are respectively 10^9 and 10^{19} for a decimal representation, or 2^{53} when using double precision floating-point registers. Most algorithms given in this Chapter work in any base; the exceptions are explicitly mentioned.

We assume that the sign is stored separately from the absolute value, which is known as the "sign-magnitude" representation. Zero is an important special case; to simplify the algorithms we assume that $n = 0$ if $A = 0$, and in most cases we assume that this case is treated separately.

Except when explicitly mentioned, we assume that all operations are *off-line*, i.e., all inputs (resp. outputs) are completely known at the beginning (resp. end) of the algorithm. Different models include on-line — also called *lazy* — algorithms, and *relaxed* algorithms.

1.2 Addition and Subtraction

As an explanatory example, here is an algorithm for integer addition. In the algorithm, d is a *carry* bit.

Our algorithms are given in a language which mixes mathematical notation and syntax similar to that found in many high-level computer languages. It should be straightforward to translate into a language such as C. The line numbers are included in case we need to refer to individual lines in the description or analysis of the algorithm.

```
1 Algorithm IntegerAddition.
2 Input: A = sum_0^{n-1} a_i β^i, B = sum_0^{n-1} b_i β^i, carry-in 0 ≤ d_in ≤ 1
3 Output: C := sum_0^{n-1} c_i β^i and 0 ≤ d ≤ 1 such that A + B + d_in = dβ^n + C
4 d ← d_in
5 for i from 0 to n − 1 do
6     s ← a_i + b_i + d
7     c_i ← s mod β
8     d ← s div β
9 Return C, d.
```

Let T be the number of different values taken by the data type represent-

ing the coefficients a_i, b_i. (Clearly $\beta \leq T$ but equality does not necessarily hold, e.g., $\beta = 10^9$ and $T = 2^{32}$.) At step 6, the value of s can be as large as $2\beta-1$, which is not representable if $\beta = T$. Several workarounds are possible: either use a machine instruction that gives the possible carry of $a_i + b_i$; or use the fact that, if a carry occurs in $a_i + b_i$, then the computed sum — if performed modulo T — equals $t := a_i + b_i - T < a_i$; thus comparing t and a_i will determine if a carry occurred. A third solution is to keep a bit in reserve, taking $\beta \leq \lceil T/2 \rceil$.

The subtraction code is very similar. Step 6 simply becomes $s \leftarrow a_i - b_i - d$, where $d \in \{0, 1\}$ is the *borrow* of the subtraction, and $-\beta \leq s < \beta$. The other steps are unchanged, with the invariant $A - B - d_{\text{in}} = -d\beta^n + C$.

Addition and subtraction of n-word integers costs $O(n)$, which is negligible compared to the multiplication cost. However, it is worth trying to reduce the constant factor implicit in this $O(n)$ cost; indeed, we shall see in §1.3 that "fast" multiplication algorithms are obtained by replacing multiplications by additions (usually more additions than the multiplications that they replace). Thus, the faster the additions are, the smaller the thresholds for changing over to the "fast" algorithms will be.

1.3 Multiplication

A nice application of large integer multiplication is the *Kronecker-Schönhage trick*, also called *segmentation* or *substitution* by some authors. Assume we want to multiply two polynomials $A(x)$ and $B(x)$ with non-negative integer coefficients (see Ex. 1.8.1 for negative coefficients). Assume both polynomials have degree less than n, and coefficients are bounded by ρ. Now take a power $X = \beta^k$ of the base β, $n\rho^2 < X$, and multiply the integers $a = A(X)$ and $b = B(X)$ obtained by evaluating A and B at $x = X$. If $C(x) = A(x)B(x) = \sum c_i x^i$, we clearly have $C(X) = \sum c_i X^i$. Now since the c_i are bounded by $n\rho^2 < X$, the coefficients c_i can be retrieved by simply "reading" blocks of k words in $C(X)$. Assume for example one wants to compute

$$(6x^5 + 6x^4 + 4x^3 + 9x^2 + x + 3)(7x^4 + x^3 + 2x^2 + x + 7),$$

with degree less than $n = 6$, and coefficients bounded by $\rho = 9$. One can thus take $X = 10^3 > n\rho^2$, and perform the integer multiplication:

$$6006004009001003 \times 7001002001007 = 42048046085072086042070010021,$$

from which one can read the product $42x^9 + 48x^8 + 46x^7 + 85x^6 + 72x^5 + 86x^4 + 42x^3 + 70x^2 + 10x + 21$.

Conversely, suppose we want to multiply two integers $a = \sum_{0 \le i < n} a_i \beta^i$ and $b = \sum_{0 \le j < n} b_j \beta^j$. Multiply the polynomials $A(x) = \sum_{0 \le i < n} a_i x^i$ and $B(x) = \sum_{0 \le j < n} b_j x^j$, obtaining a polynomial $C(x)$, then evaluate $C(x)$ at $x = \beta$ to obtain ab. Note that the coefficients of $C(x)$ may be larger than β, in fact they may be up to about $n\beta^2$. For example with $a = 123$ and $b = 456$ with $\beta = 10$, we obtain $A(x) = x^2 + 2x + 3$, $B(x) = 4x^2 + 5x + 6$, whose product is $C(x) = 4x^4 + 13x^3 + 28x^2 + 27x + 18$, and $C(10) = 56088$. These examples demonstrate the analogy between operations on polynomials and integers, and also show the limits of the analogy.

A common and very useful notation is to let $M(n)$ denote the time to multiply n-bit integers, or polynomials of degree $n-1$, depending on the context. In the polynomial case, we assume that the cost of multiplying coefficients is constant; this is known as the *arithmetic complexity* model, whereas the *bit complexity* model also takes into account the cost of multiplying coefficients, and thus their bit-size.

1.3.1 Naive Multiplication

```
1 Algorithm BasecaseMultiply.
2 Input: A = sum_0^{m-1} a_i β^i , B = sum_0^{n-1} b_j β^j
3 Output: C = AB := sum_0^{m+n-1} c_k β^k
4 C ← A · b_0
5 for j from 1 to n − 1 do
6     C ← C + β^j (A · b_j)
7 Return C.
```

Theorem 1.3.1 *Algorithm **BasecaseMultiply** computes the product AB correctly, and uses $\Theta(mn)$ word operations.*

The multiplication by β^j at step 6 is trivial with the chosen dense representation: it simply requires shifting by j words towards the most significant words. The main operation in algorithm **BasecaseMultiply** is the computation of $A \cdot b_j$ and its accumulation into C at step 6. Since all fast algorithms

rely on multiplication, the most important operation to optimize in multiple-precision software is thus the multiplication of an array of m words by one word, with accumulation of the result in another array of $m+1$ words.

Since multiplication with accumulation usually makes extensive use of the pipeline, it is best to give it arrays that are as long as possible, which means that A rather than B should be the operand of larger size (i.e., $m \geq n$).

1.3.2 Karatsuba's Algorithm

In the following, $n_0 \geq 2$ denotes the threshold between naive multiplication and Karatsuba's algorithm, which is used for n_0-word and larger inputs. The optimal "Karatsuba threshold" n_0 can vary from 10 to 100 words depending on the processor, and the relative efficiency of the word multiplication and addition (see Ex. 1.8.5).

```
1  Algorithm KaratsubaMultiply.
2  Input: A = Σ_0^{n-1} a_i β^i ,  B = Σ_0^{n-1} b_j β^j
3  Output: C = AB := Σ_0^{2n-1} c_k β^k
4  if n < n_0 then return BasecaseMultiply(A, B)
5  k ← ⌈n/2⌉
6  (A_0, B_0) := (A, B) mod β^k ,  (A_1, B_1) := (A, B) div β^k
7  s_A ← sign(A_0 − A_1) ,  s_B ← sign(B_0 − B_1)
8  C_0 ← KaratsubaMultiply(A_0, B_0)
9  C_1 ← KaratsubaMultiply(A_1, B_1)
10 C_2 ← KaratsubaMultiply(|A_0 − A_1|, |B_0 − B_1|)
11 Return C := C_0 + (C_0 + C_1 − s_A s_B C_2)β^k + C_1 β^{2k} .
```

Theorem 1.3.2 *Algorithm **KaratsubaMultiply** computes the product AB correctly, using $K(n) = O(n^\alpha)$ word multiplications, with $\alpha = \log_2 3 \approx 1.585$.*

Proof. Since $s_A|A_0 - A_1| = A_0 - A_1$, and similarly for B, $s_A s_B |A_0 - A_1||B_0 - B_1| = (A_0 - A_1)(B_0 - B_1)$, thus $C = A_0B_0 + (A_0B_1 + A_1B_0)\beta^k + A_1B_1\beta^{2k}$.

Since A_0, B_0, $|A_0 - A_1|$ and $|B_0 - B_1|$ have (at most) $\lceil n/2 \rceil$ words, and A_1 and B_1 have (at most) $\lfloor n/2 \rfloor$ words, the number $K(n)$ of word multiplications satisfies the recurrence $K(n) = n^2$ for $n < n_0$, and $K(n) = 2K(\lceil n/2 \rceil) + K(\lfloor n/2 \rfloor)$ for $n \geq n_0$. Assume $2^{\ell-1}n_0 < n \leq 2^\ell n_0$ with $\ell \geq 1$, then $K(n)$ is the sum of three $K(j)$ values with $j \leq 2^{l-1}n_0, \ldots$, thus of 3^l $K(j)$ with

$j \leq n_0$. Thus $K(n) \leq 3^l \max(K(n_0), (n_0-1)^2)$, which gives $K(n) \leq Cn^\alpha$ with $C = 3^{1-\log_2 n_0}\max(K(n_0), (n_0-1)^2)$. □

Different variants of Karatsuba's algorithm exist; this variant is known as the *subtractive* version. Another classical one is the *additive* version, which uses $A_0 + A_1$ and $B_0 + B_1$ instead of $|A_0 - A_1|$ and $|B_0 - B_1|$. However, the subtractive version is more convenient for integer arithmetic, since it avoids the possible carries in $A_0 + A_1$ and $B_0 + B_1$, which require either an extra word in those sums, or extra additions.

The efficiency of an implementation of Karatsuba's algorithm depends heavily on memory usage. It is quite important to avoid allocating memory for the intermediate results $|A_0 - A_1|$, $|B_0 - B_1|$, C_0, C_1, and C_2 at each step (although modern compilers are quite good at optimising code and removing unnecessary memory references). One possible solution is to allow a large temporary storage of m words, used both for the intermediate results and for the recursive calls. It can be shown that an auxiliary space of $m = 2n$ words — or even $m = n$ in the polynomial case — is sufficient (see Ex. 1.8.6).

Since the third product C_2 is used only once, it may be faster to have two auxiliary routines **KaratsubaAddmul** and **KaratsubaSubmul** that accumulate their result, calling themselves recursively, together with **KaratsubaMultiply** (see Ex. 1.8.8).

The above version uses $\sim 4n$ additions (or subtractions): $2\times\frac{n}{2}$ to compute $|A_0 - A_1|$ and $|B_0 - B_1|$, then n to add C_0 and C_1, again n to add or subtract C_2, and n to add $(C_0 + C_1 - s_A s_B C_2)\beta^k$ to $C_0 + C_1\beta^{2k}$. An improved scheme uses only $\sim \frac{7}{2}n$ additions (see Ex. 1.8.7).

Most fast multiplication algorithms can be viewed as evaluation/interpolation algorithms, from a polynomial point of view. Karatsuba's algorithm regards the inputs as polynomials $A_0 + A_1 x$ and $B_0 + B_1 x$ evaluated at $x = \beta^k$; since their product $C(x)$ is of degree 2, Lagrange's interpolation theorem says that it is sufficient to evaluate it at three points. The subtractive version evaluates $C(x)$ at $x = 0, -1, \infty$, whereas the additive version uses $x = 0, +1, \infty$.[1]

1.3.3 Toom-Cook Multiplication

Karatsuba's idea readily generalizes to what is known as Toom-Cook r-way multiplication. Write the inputs as $a_0 + \cdots + a_{r-1}x^{r-1}$ and $b_0 + \cdots + b_{r-1}x^{r-1}$,

[1] Evaluating $C(x)$ at ∞ means computing the product $A_1 B_1$ of the leading coefficients.

with $x = \beta^k$, and $k = \lceil n/r \rceil$. Since their product $C(x)$ is of degree $2r-2$, it suffices to evaluate it at $2r-1$ distinct points to be able to recover $C(x)$, and in particular $C(\beta^k)$.

Most references, when describing subquadratic multiplication algorithms, only describe Karatsuba and FFT-based algorithms. Nevertheless, the Toom-Cook algorithm is quite interesting in practice.

Toom-Cook r-way reduces one n-word product to $2r-1$ products of $\lceil n/r \rceil$ words. This gives an asymptotic complexity of $O(n^\nu)$ with $\nu = \frac{\log(2r-1)}{\log r}$. However, the constant hidden by the big-O notation depends strongly on the evaluation and interpolation formulæ, which in turn depend on the chosen points. One possibility is to take $-(r-1), \ldots, -1, 0, 1, \ldots, (r-1)$ as evaluation points.

The case $r = 2$ corresponds to Karatsuba's algorithm (§1.3.2). The case $r = 3$ is known as Toom-Cook 3-way, sometimes simply called "the Toom-Cook algorithm". The following algorithm uses evaluation points $0, 1, -1, 2, \infty$, and tries to optimize the evaluation and interpolation formulæ.

1 Algorithm **ToomCook3**.
2 **Input**: two integers $0 \le A, B < \beta^n$.
3 **Output**: $AB := c_0 + c_1\beta^k + c_2\beta^{2k} + c_3\beta^{3k} + c_4\beta^{4k}$ with $k = \lceil n/3 \rceil$.
4 **if** $n < 3$ **then** return KaratsubaMultiply(A, B)
5 Write $A = a_0 + a_1 x + a_2 x^2$, $B = b_0 + b_1 x + b_2 x^2$ with $x = \beta^k$.
6 $v_0 \leftarrow$ ToomCook3(a_0, b_0)
7 $v_1 \leftarrow$ ToomCook3$(a_{02} + a_1, b_{02} + b_1)$ where $a_{02} \leftarrow a_0 + a_2, b_{02} \leftarrow b_0 + b_2$
8 $v_{-1} \leftarrow$ ToomCook3$(a_{02} - a_1, b_{02} - b_1)$
9 $v_2 \leftarrow$ ToomCook3$(a_0 + 2a_1 + 4a_2, b_0 + 2b_1 + 4b_2)$
10 $v_\infty \leftarrow$ ToomCook3(a_2, b_2)
11 $t_1 \leftarrow (3v_0 + 2v_{-1} + v_2)/6 - 2v_\infty$, $t_2 \leftarrow (v_1 + v_{-1})/2$
12 $c_0 \leftarrow v_0$, $c_1 \leftarrow v_1 - t_1$, $c_2 \leftarrow t_2 - v_0 - v_\infty$, $c_3 \leftarrow t_1 - t_2$, $c_4 \leftarrow v_\infty$

The divisions at step 11 are exact; if β is a power of two, the division by 6 can be done using a division by 2 — which consists of a single shift — followed by a division by 3 (§1.4.7).

For higher order Toom-Cook implementations see [135], which considers the 4-way and 5-way variants, together with squaring. Toom-Cook r-way has to invert a $(2r-1) \times (2r-1)$ Vandermonde matrix with parameters the evaluation points; if one chooses consecutive integer points, the determinant

of that matrix contains all primes up to $2r-2$. This proves that the division by 3 can not be avoided for Toom-Cook 3-way with consecutive integer points (see Ex. 1.8.12 for a generalization of this result).

1.3.4 Fast Fourier Transform (FFT)

Most subquadratic multiplication algorithms can be seen as evaluation-interpolation algorithms. They mainly differ in the number of evaluation points, and the values of those points. However the evaluation and interpolation formulæ become intricate in Toom-Cook r-way for large r, since they involve $O(r^2)$ scalar operations. The Fast Fourier Transform (FFT) is a way to perform evaluation and interpolation in an efficient way for some special points (roots of unity) and special values of r. This explains why multiplication algorithms of the best asymptotic complexity are based on the Fast Fourier Transform.

There are different flavours of FFT multiplication, depending on the ring where the operations are performed. Schönhage-Strassen's algorithm [116], with a complexity of $O(n \log n \log\log n)$, works in the ring $\mathbb{Z}/(2^n+1)\mathbb{Z}$; since it is based on modular computations, we describe it in Chapter 2.

Other commonly used algorithms work with floating-point complex numbers [81, Section 4.3.3.C]; one drawback is that, due to the inexact nature of floating-point computations, a careful error analysis is required to guarantee the correctness of the implementation, assuming an underlying arithmetic with rigorous error bounds (*cf* Chapter 3).

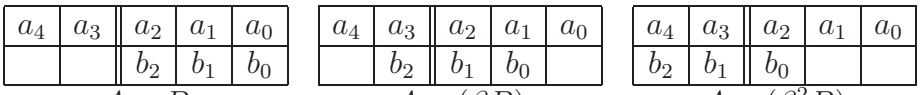
We say that multiplication is *in the FFT range* if n is large and the multiplication algorithm satisfies and $M(2n) \sim M(n)$. For example, this is true if the Schönhage-Strassen multiplication algorithm is used, but not if the classical algorithm or Karatsuba's algorithm is used.

1.3.5 Unbalanced Multiplication

The subquadratic algorithms considered so far (Karatsuba and Toom-Cook) work with equal-size operands. How do we efficiently multiply integers of different sizes with a subquadratic algorithm? This case is important in practice but is rarely considered in the literature. Assume the larger operand has size m, and the smaller has size $n \leq m$, and denote by $M(m,n)$ the corresponding multiplication cost.

When m is an exact multiple of n, say $m = kn$, a trivial strategy is to cut the larger operand into k pieces, giving $M(kn, n) = kM(n) + O(kn)$. However, this is not always the best strategy, see Ex. 1.8.14.

When m is not an exact multiple of n, different strategies are possible. Consider for example Karatsuba multiplication, and let $K(m, n)$ be the number of word-products for an $m \times n$ product. Take for example $m = 5$, $n = 3$. A natural idea is to pad the smallest operand to the size of the largest one. However there are several ways to perform this padding, as shown in the Figure, where the "Karatsuba cut" is represented by a double column:



The first strategy leads to two products of size 3, i.e., $2K(3, 3)$, the second one to $K(2, 1)+K(3, 2)+K(3, 3)$, and the third one to $K(2, 2)+K(3, 1)+K(3, 3)$, which give respectively 14, 15, 13 word products.

However, whenever $m/2 \le n \le m$, any such "padding strategy" will require $K(\lceil m/2 \rceil, \lceil m/2 \rceil)$ for the product of the differences (or sums) of the low and high parts from the operands, due to a "wrap around" effect when subtracting the parts from the smaller operand; this will ultimately lead to a cost similar to that of an $m \times m$ product. The "odd-even scheme" (see also Ex. 1.8.11) avoids this wrap around. Here is an example of Algorithm **OddEvenKaratsuba** for $m = 3$ and $n = 2$. Take $A = a_2x^2 + a_1x + a_0$

```
1  Algorithm OddEvenKaratsuba.
2  Input: A = Σ_0^{m-1} a_i x^i , B = Σ_0^{n-1} b_j x^j , m ≥ n
3  Output: A · B
4  if n = 1 then return Σ_0^{m-1} a_i b_0 x^i
5  k ← ⌈m/2⌉ , ℓ ← ⌈n/2⌋
6  Write A = A_0(x^2) + xA_1(x^2) , B = B_0(x^2) + xB_1(x^2)
7  A_0 ← A mod x^k , A_1 ← A div x^k
8  C_0 ← OddEvenKaratsuba(A_0, B_0)
9  C_1 ← OddEvenKaratsuba(A_0 + A_1, B_0 + B_1)
10 C_2 ← OddEvenKaratsuba(A_1, B_1)
11 Return C_0(x^2) + x(C_1 − C_0 − C_2)(x^2) + x^2 C_2(x^2) .
```

and $B = b_1x + b_0$. This yields $A_0 = a_2x + a_0$, $A_1 = a_1$, $B_0 = b_0$, $B_1 = b_1$,

thus $C_0 = (a_2x + a_0)b_0$, $C_1 = (a_2x + a_0 + a_1)(b_0 + b_1)$, $C_2 = a_1b_1$. We thus get $K(3,2) = 2K(2,1) + K(1) = 5$ with the odd-even scheme. The general recurrence for the odd-even scheme is:

$$K(m,n) = 2K(\lceil m/2\rceil, \lceil n/2\rceil) + K(\lfloor m/2\rfloor, \lfloor n/2\rfloor),$$

instead of

$$K(m,n) = 2K(\lceil m/2\rceil, \lceil m/2\rceil) + K(\lfloor m/2\rfloor, n - \lceil m/2\rceil)$$

for the classical strategy, assuming $n > m/2$. The second parameter in $K(\cdot,\cdot)$ only depend on the smaller size n for the odd-even scheme.

As for the classical strategy, there are several ways of padding with the odd-even scheme. Consider $m = 5$, $n = 3$, and write $A := a_4x^4+a_3x^3+a_2x^2+a_1x+a_0 = xA_1(x^2)+A_0(x^2)$, with $A_1(x) = a_3x+a_1$, $A_0(x) = a_4x^2+a_2x+a_0$; and $B := b_2x^2 + b_1x + b_0 = xB_1(x^2) + B_0(x^2)$, with $B_1(x) = b_1$, $B_0(x) = b_2x+b_0$. Without padding, we write $AB = x^2(A_1B_1)(x^2)+x((A_0+A_1)(B_0+B_1) - A_1B_1 - A_0B_0)(x^2) + (A_0B_0)(x^2)$, which gives $K(5,3) = K(2,1) + 2K(3,2) = 12$. With padding, we consider $xB = xB_1'(x^2) + B_0'(x^2)$, with $B_1'(x) = b_2x + b_0$, $B_0' = b_1x$. This gives $K(2,2) = 3$ for A_1B_1', $K(3,2) = 5$ for $(A_0 + A_1)(B_0' + B_1')$, and $K(3,1) = 3$ for A_0B_0' — taking into account the fact that B_0' has only one non-zero coefficient — thus a total of 11 only.

1.3.6 Squaring

In many applications, a significant proportion of the multiplications have equal operands, i.e., are squarings. Hence it is worth tuning a special squaring implementation as much as the implementation of multiplication itself, bearing in mind that the best possible speedup is two (see Ex. 1.8.15).

For naive multiplication, Algorithm **BasecaseMultiply** (§1.3.1) can be modified to obtain a theoretical speedup of two, since only about half of the products a_ib_j need to be computed.

Subquadratic algorithms like Karatsuba and Toom-Cook r-way can be specialized for squaring too. In general, the threshold obtained is larger than the corresponding multiplication threshold.

1.3.7 Multiplication by a Constant

It often happens that one integer is used in several consecutive multiplications, or is fixed for a complete calculation. If this constant multiplier is

small, i.e., less than the base β, not much speedup can be obtained compared to the usual product. We thus consider here a "large" constant multiplier.

When using evaluation-interpolation algorithms, like Karatsuba or Toom-Cook (see §1.3.2–1.3.3), one may store the results of the evaluation for that fixed multiplicand. If one assumes that an interpolation is as expensive as one evaluation, this may give a speedup of up to 3/2.

Special-purpose algorithms also exist. These algorithms differ from classical multiplication algorithms because they take into account the *value* of the given constant multiplier, and not only its size in bits or digits. They also differ in the model of complexity used. For example, Bernstein's algorithm [15], which is used by several compilers to compute addresses in data structure records, considers as basic operation $x, y \to 2^i x \pm y$, with a cost assumed to be independent of the integer i.

For example Bernstein's algorithm computes $20061x$ in five steps:

$$\begin{aligned} x_1 := 31x &= 2^5 x - x \\ x_2 := 93x &= 2^1 x_1 + x_1 \\ x_3 := 743x &= 2^3 x_2 - x \\ x_4 := 6687x &= 2^3 x_3 + x_3 \\ 20061x &= 2^1 x_4 + x_4. \end{aligned}$$

See [87] for a comparison of different algorithms for the problem of multiplication by an integer constant.

1.4 Division

Division is the next operation to consider after multiplication. Optimizing division is almost as important as optimizing multiplication, since division is usually more expensive, thus the speedup obtained on division will be more significant. (On the other hand, one usually performs more multiplications than divisions.) One strategy is to avoid divisions when possible, or replace them by multiplications. An example is when the same divisor is used for several consecutive operations; one can then precompute its inverse (see §2.3.1).

We distinguish several kinds of division: *full division* computes both quotient and remainder, while in some cases only the quotient (for example when dividing two floating-point mantissas) or remainder (when multiplying two

residues modulo n) is needed. Then we discuss exact division — when the remainder is known to be zero — and the problem of dividing by a constant.

1.4.1 Naive Division

In all division algorithms, we will consider *normalized* divisors. We say that $B := \sum_0^{n-1} b_j \beta^j$ is *normalized* when its most significant word b_{n-1} satisfies $b_{n-1} \geq \beta/2$. This is a stricter condition (for $\beta > 2$) than simply requiring that b_{n-1} be nonzero.

```
1  Algorithm BasecaseDivRem.
2  Input: A = Σ_0^{n+m-1} a_i β^i , B = Σ_0^{n-1} b_j β^j , B normalized
3  Output: quotient Q and remainder R of A divided by B.
4  if A ≥ β^m B then q_m ← 1, A ← A − β^m B else q_m ← 0
5  for j from m − 1 downto 0 do
6      q*_j ← ⌊(a_{n+j} β + a_{n+j-1})/b_{n-1}⌋   { quotient selection step }
7      q_j ← min(q*_j, β − 1)
8      A ← A − q_j β^j B
9      while A < 0 do
10         q_j ← q_j − 1
11         A ← A + β^j B
12 Return Q = Σ_0^m q_j β^j , R = A.
```

(Note: in the above algorithm, a_i denotes the *current* value of the i-th word of A, after the possible changes at steps 8 and 11.)

If B is not normalized, we can compute $A' = 2^k A$ and $B' = 2^k B$ so that B' is normalized, then divide A' by B' giving $A' = Q'B' + R'$; the quotient and remainder of the division of A by B are respectively $Q := Q'$ and $R := R'/2^k$, the latter division being exact.

Theorem 1.4.1 *Algorithm* ***BasecaseDivRem*** *correctly computes the quotient and remainder of the division of A by a normalized B, in $O(nm)$ word operations.*

Proof. First prove that the invariant $A < \beta^{j+1} B$ holds at step 5. This holds trivially for $j = m - 1$: B being normalized, $A < 2\beta^m B$ initially.

First consider the case $q_j = q_j^*$: then $q_j b_{n-1} \geq a_{n+j}\beta + a_{n+j-1} - b_{n-1} + 1$, thus

$$A - q_j \beta^j B \leq (b_{n-1} - 1)\beta^{n+j-1} + (A \bmod \beta^{n+j-1}),$$

which ensures that the new a_{n+j} vanishes, and $a_{n+j-1} < b_{n-1}$, thus $A < \beta^j B$ after step 8. Now A may become negative after step 8, but since $q_j b_{n-1} \leq a_{n+j}\beta + a_{n+j-1}$:

$$A - q_j\beta^j B > (a_{n+j}\beta + a_{n+j-1})\beta^{n+j-1} - q_j(b_{n-1}\beta^{n-1} + \beta^{n-1})\beta^j \geq -q_j\beta^{n+j-1}.$$

Therefore $A - q_j\beta^j B + 2\beta^j B \geq (2b_{n-1} - q_j)\beta^{n+j-1} > 0$, which proves that the while-loop at steps 9-11 is performed at most twice [81, Theorem 4.3.1.B]. When the while-loop is entered, A may increase only by $\beta^j B$ at a time, hence $A < \beta^j B$ at exit.

In the case $q_j \neq q_j^*$, i.e., $q_j^* \geq \beta$, we have before the while-loop: $A < \beta^{j+1}B - (\beta - 1)\beta^j B = \beta^j B$, thus the invariant holds. If the while-loop is entered, the same reasoning as above holds.

We conclude that when the for-loop ends, $0 \leq A < B$ holds, and since $(\sum_j^m q_j\beta^j)B + A$ is invariant through the algorithm, the quotient Q and remainder R are correct.

The most expensive step is step 8, which costs $O(n)$ operations for $q_j B$ (the multiplication by β^j is simply a word-shift); the total cost is $O(nm)$. □

Here is an example of algorithm **BasecaseDivRem** for the inputs $A = 766970544842443844$ and $B = 862664913$, with $\beta = 1000$: which gives quotient $Q = 889071217$ and remainder $R = 778334723$.

j	A	q_j	$A - q_j B\beta^j$	after correction
2	766 970 544 842 443 844	889	61 437 185 443 844	no change
1	61 437 185 443 844	071	187 976 620 844	no change
0	187 976 620 844	218	−84 330 190	778 334 723

Algorithm **BasecaseDivRem** simplifies when $A < \beta^m B$: remove step 4, and change m into $m - 1$ in the return value Q. However, the more general form we give is more convenient for a computer implementation, and will be used below.

A possible variant when $q_j^* \geq \beta$ is to let $q_j = \beta$; then $A - q_j\beta^j B$ at step 8 reduces to a single subtraction of B shifted by $j + 1$ words. However in this case the while-loop will be performed at least once, which corresponds to the identity $A - (\beta - 1)\beta^j B = A - \beta^{j+1}B + \beta^j B$.

If instead of having B normalized, i.e., $b_n \geq \beta/2$, one has $b_n \geq \beta/k$, there can be up to k iterations of the while-loop (and step 4 has to be modified).

A drawback of algorithm **BasecaseDivRem** is that the test $A < 0$ at line 9 is true with non-negligible probability, therefore branch prediction algorithms available on modern processors will fail, resulting in wasted cycles. A workaround is to compute a more accurate partial quotient, and therefore decrease the proportion of corrections to almost zero (see Ex. 1.8.18).

1.4.2 Divisor Preconditioning

Sometimes the quotient selection — step 6 of Algorithm **BasecaseDivRem** — is quite expensive compared to the total cost, especially for small sizes. Indeed, some processors do not have a machine instruction for the division of two words by one word; one way to compute q_j^* is then to precompute a one-word approximation of the inverse of b_{n-1}, and to multiply it by $a_{n+j}\beta + a_{n+j-1}$.

Svoboda's algorithm [123] makes the quotient selection trivial, after preconditioning the divisor. The main idea is that if b_{n-1} equals the base β in Algorithm **BasecaseDivRem**, then the quotient selection is easy, since it suffices to take $q_j^* = a_{n+j}$. (In addition, $q_j^* \le \beta - 1$ is then always fulfilled, thus step 7 of **BasecaseDivRem** can be avoided, and q_j^* replaced by q_j.)

```
1  Algorithm SvobodaDivision.
2  Input: A = sum_0^{n+m-1} a_i β^i , B = sum_0^{n-1} b_j β^j normalized , A < β^m B
3  Output: quotient Q and remainder R of A divided by B.
4  k ← ⌈β^{n+1}/B⌉
5  B' ← kB = β^{n+1} + sum_0^{n-1} b'_j β^j
6  for j from m − 1 downto 1 do
7      q_j ← a_{n+j}
8      A ← A − q_j β^{j−1} B'
9      if A < 0 do
10         q_j ← q_j − 1
11         A ← A + β^{j−1} B'
12 Q' = sum_1^{m-1} q_j β^j , R' = A
13 (q_0, R) ← (R' div B, R' mod B)
14 Return Q = kQ' + q_0 , R.
```

The division at step 13 can be performed with **BasecaseDivRem**; it gives a single word since A has $n + 1$ words.

With the example of section §1.4.1, Svoboda's algorithm would give $k = 1160$, $B' = 1000691299080$:

j	A	q_j	$A - q_j B' \beta^j$	after correction
2	766 970 544 842 443 844	766	441 009 747 163 844	no change
1	441 009 747 163 844	441	−295 115 730 436	705 575 568 644

We thus get $Q' = 766440$ and $R' = 705575568644$. The final division of Step 13 gives $R' = 817B + 778334723$, thus we get $Q = 1160 \cdot 766440 + 817 = 889071217$, and $R = 778334723$, as in §1.4.1.

Svoboda's algorithm is especially interesting when only the remainder is needed, since then one can avoid the "deconditioning" $Q = kQ' + q_0$. Note that when only the quotient is needed, dividing $A' = kA$ by $B' = kB$ yields it.

1.4.3 Divide and Conquer Division

The base-case division from §1.4.1 determines the quotient word by word. A natural idea is to try getting several words at a time, for example replacing the quotient selection step in Algorithm **BasecaseDivRem** by:

$$q_j^* \leftarrow \lfloor \frac{a_{n+j}\beta^3 + a_{n+j-1}\beta^2 + a_{n+j-2}\beta + a_{n+j-3}}{b_{n-1}\beta + b_{n-2}} \rfloor .$$

Since q_j^* has then two words, fast multiplication algorithms (§1.3) might speed up the computation of $q_j B$ at step 8 of Algorithm **BasecaseDivRem**.

More generally, the most significant half of the quotient — say Q_1, of k words — mainly depends on the k most significant words of the dividend and divisor. Once a good approximation to Q_1 is known, fast multiplication algorithms can be used to compute the partial remainder $A - Q_1 B$. The second idea of the divide and conquer division algorithm below is to compute the corresponding remainder together with the partial quotient Q_1; in such a way, one only has to subtract the product of Q_1 by the low part of the divisor, before computing the low part of the quotient.

In Algorithm **RecursiveDivRem**, one may replace the condition $m < 2$ at step 4 by $m < T$ for any integer $T \geq 2$. In practice, T is usually in the range 50 to 200.

One can not require $A < \beta^m B$ here, since this condition may not be satisfied in the recursive calls. Consider for example $A = 5517$, $B = 56$

1 Algorithm **RecursiveDivRem**.
2 **Input**: $A = \sum_0^{n+m-1} a_i\beta^i$, $B = \sum_0^{n-1} b_j\beta^j$, B normalized, $n \geq m$
3 **Output**: quotient Q and remainder R of A divided by B.
4 **if** $m < 2$ **then** return **BasecaseDivRem**(A, B)
5 $k \leftarrow \lfloor \frac{m}{2} \rfloor$, $B_1 \leftarrow B \bmod \beta^k$, $B_0 \leftarrow B \bmod \beta^k$
6 $(Q_1, R_1) \leftarrow$ **RecursiveDivRem**$(A \operatorname{div} \beta^{2k}, B_1)$
7 $A' \leftarrow R_1\beta^{2k} + (A \bmod \beta^{2k}) - Q_1B_0\beta^k$
8 **while** $A' < 0$ **do** $Q_1 \leftarrow Q_1 - 1$, $A' \leftarrow A' + \beta^k B$
9 $(Q_0, R_0) \leftarrow$ **RecursiveDivRem**$(A' \operatorname{div} \beta^k, B_1)$
10 $A'' \leftarrow R_0\beta^k + (A' \bmod \beta^k) - Q_0B_0$
11 **while** $A'' < 0$ **do** $Q_0 \leftarrow Q_0 - 1$, $A'' \leftarrow A'' + B$
12 Return $Q := Q_1\beta^k + Q_0$, $R := A''$.

with $\beta = 10$: the first recursive call will divide 55 by 5, which yields a two-digit quotient 11. Even $A \leq \beta^m B$ is not recursively fulfilled; consider $A = 55170000$ with $B = 5517$: the first recursive call will divide 5517 by 55. The weakest possible condition is that the n most significant words of A do not exceed those of B, i.e., $A < \beta^m(B + 1)$. In that case, the quotient is bounded by $\beta^m + \lfloor \frac{\beta^m - 1}{B} \rfloor$, which yields $\beta^m + 1$ in the case $n = m$ (compare Ex. 1.8.17). See also Ex. 1.8.20.

Theorem 1.4.2 *Algorithm* ***RecursiveDivRem*** *is correct, and uses* $D(n + m, n)$ *operations, where* $D(n + m, n) = 2D(n, n - m/2) + 2M(m/2) + O(n)$. *In particular* $D(n) := D(2n, n)$ *satisfies* $D(n) = 2D(n/2) + 2M(n/2) + O(n)$, *which gives* $D(n) \sim \frac{1}{2^{\alpha-1}-1} M(n)$ *for* $M(n) \sim n^\alpha$, $\alpha > 1$.

Proof. We first check the assumption for the recursive calls: B_1 is normalized since it has the same most significant word than B.

After step 6, we have $A = (Q_1B_1 + R_1)\beta^{2k} + (A \bmod \beta_{2k})$, thus after step 7: $A' = A - Q_1\beta^k B$, which still holds after step 8. After step 9, we have $A' = (Q_0B_1 + R_0)\beta^k + (A' \bmod \beta^k)$, thus after step 10: $A'' = A' - Q_0B$, which still holds after step 11. At step 12 we thus have $A = QB + R$.

$A \operatorname{div} \beta^{2k}$ has $m + n - 2k$ words, while B_1 has $n - k$ words, thus $0 \leq Q_1 < 2\beta^{m-k}$ and $0 \leq R_1 < B_1 < \beta^{n-k}$. Thus at step 7, $-2\beta^{m+k} < A' < \beta^k B$. Since B is normalized, the while-loop at step 8 is performed at most four times (this can happen only when $n = m$). At step 9 we have $0 \leq A' < \beta^k B$, thus $A' \operatorname{div} \beta^k$ has at most n words.

It follows $0 \leq Q_0 < 2\beta^k$ and $0 \leq R_0 < B_1 < \beta^{n-k}$. Hence at step 10, $-2\beta^{2k} < A'' < B$, and after at most four iterations at step 11, we have $0 \leq A'' < B$. □

Theorem 1.4.2 gives $D(n) \sim 2M(n)$ for Karatsuba multiplication, and $D(n) \sim 2.63\ldots M(n)$ for Toom-Cook 3-way; in the FFT range, see Ex. 1.8.21.

The same idea as in Ex. 1.8.18 applies: to decrease the probability that the estimated quotients Q_1 and Q_0 are too large, use one extra word of the truncated dividend and divisors in the recursive calls to **RecursiveDivRem**.

A graphical view of Algorithm **RecursiveDivRem** in the case $m = n$ is given in Fig. 1.1, which represents the multiplication $Q \cdot B$: one first computes the lower left corner in $D(n/2)$ (step 6), second the lower right corner in $M(n/2)$ (step 7), third the upper left corner in $D(n/2)$ (step 9), and finally the upper right corner in $M(n/2)$ (step 10).

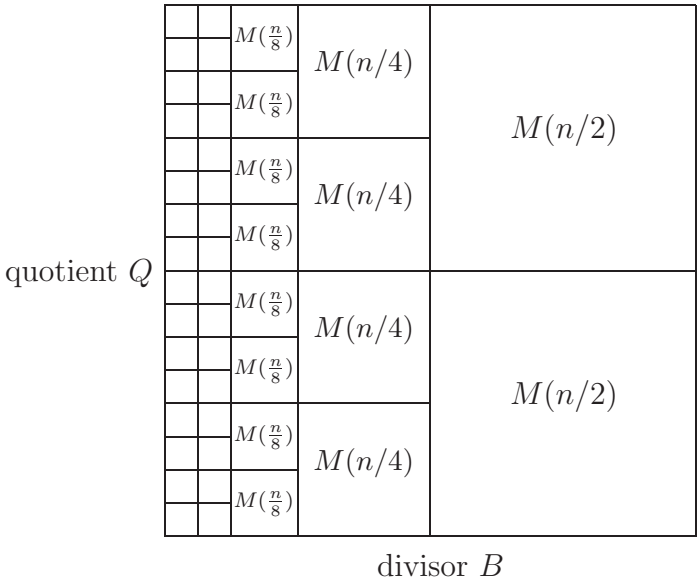


Figure 1.1: Divide and conquer division: a graphical view (most significant parts at the lower left corner).

Unbalanced Division

The condition $n \geq m$ in Algorithm **RecursiveDivRem** means that the dividend A is at most twice as large as the divisor B.

When A is more than twice as large as B ($m > n$ with the above notations), a possible strategy (see Ex. 1.8.22) computes n words of the quotient at a time. This reduces to the base-case algorithm, replacing β by β^n.

```
1  Algorithm UnbalancedDivision.
2  Input: A = ∑_0^{n+m-1} a_i β^i , B = ∑_0^{n-1} b_j β^j , B normalized , m > n.
3  Output: quotient Q and remainder R of A divided by B.
4  Q ← 0
5  while m > n do
6      (q, r) ← RecursiveDivRem(A div β^{m-n}, B) { 2n by n division }
7      Q ← Qβ^n + q
8      A ← rβ^{m-n} + A mod β^{m-n}
9      m ← m − n
10 (q, r) ← RecursiveDivRem(A, B)
11 Return Q := Qβ^m + q , R := r .
```

1.4.4 Newton's Method

Newton's iteration gives the division algorithm with best asymptotic complexity. One basic component of Newton's iteration is the computation of an approximate inverse. We refer here to Chapter 4. The p-adic version of Newton's method, also called Hensel lifting, is used in §1.4.5 for the exact division.

1.4.5 Exact Division

A division is *exact* when the remainder is zero. This happens for example when normalizing a fraction a/b: one divides both a and b by their greatest common divisor, and both divisions are exact. If the remainder is known *a priori* to be zero, this information is useful to speed up the computation of the quotient. Two strategies are possible:

- use classical MSB (most significant bits first) division algorithms, without computing the lower part of the remainder. Here, one has to take care of rounding errors, in order to guarantee the correctness of the final result; or

- use LSB (least significant bits first) algorithms. If the quotient is known to be less than β^n, computing $a/b \bmod \beta^n$ will reveal it.

In both strategies, subquadratic algorithms can be used too. We describe here the least significant bit algorithm, using Hensel lifting — which can be seen as a p-adic version of Newton's method:

```
1 Algorithm ExactDivision.
2 Input: A = sum_0^{n-1} a_i β^i ,  B = sum_0^{n-1} b_j β^j
3 Output: quotient Q = A/B mod β^n
4 C ← 1/b_0 mod β
5 for i from ⌈log_2 n⌉ − 1 downto 1 do
6     k ← ⌈n/2^i⌉
7     C ← C + C(1 − BC) mod β^k
8 Q ← AC mod β^k
9 Q ← Q + C(A − BQ) mod β^n
```

Algorithm **ExactDivision** uses the Karp-Markstein trick: lines 4-7 compute $1/B \bmod \beta^{\lceil n/2 \rceil}$, while the two last lines incorporate the dividend to obtain $A/B \bmod \beta^n$. Note that the *middle product* (§3.3.2) can be used in lines 7 and 9, to speed up the computation of $1-BC$ and $A-BQ$ respectively.

Finally, another gain is obtained using both strategies simultaneously: compute the most significant $n/2$ bits of the quotient using the MSB strategy, and the least $n/2$ bits using the LSB one. Since an exact division of size n is replaced by two exact divisions of size $n/2$, this gives a speedup up to 2 for quadratic algorithms (see Ex. 1.8.25).

1.4.6 Only Quotient or Remainder Wanted

When both the quotient and remainder of a division are needed, it is best to compute them simultaneously. This may seem to be a trivial statement, nevertheless some high-level languages provide both **div** and **mod**, but no single instruction to compute both quotient and remainder.

Once the quotient is known, the remainder can be recovered by a single multiplication as $a - qb$; on the other hand, when the remainder is known, the quotient can be recovered by an exact division as $(a - r)/b$ (§1.4.5).

However, it often happens that only one of the quotient or remainder is needed. For example, the division of two floating-point numbers reduces to

the quotient of their fractions (see Chapter 3). Conversely, the multiplication of two numbers modulo n reduces to the remainder of their product after division by n (see Chapter 2). In such cases, one may wonder if faster algorithms exist.

For a dividend of $2n$ words and a divisor of n words, a significant speedup — up to two for quadratic algorithms — can be obtained when only the quotient is needed, since one does not need to update the low n words of the current remainder (step 8 of Algorithm **BasecaseDivRem**).

It seems difficult to get a similar speedup when only the remainder is required. One possibility is to use Svoboda's algorithm, but this requires some precomputation, so is only useful when several divisions are performed with the same divisor. The idea is the following: precompute a multiple B_1 of B, having $3n/2$ words, the $n/2$ most significant words being $\beta^{n/2}$. Then reducing A mod B_1 reduces to a single $n/2 \times n$ multiplication. Once A is reduced into A_1 of $3n/2$ words by Svoboda's algorithm in $2M(n/2)$, use `RecursiveDivRem` on A_1 and B, which costs $D(n/2) + M(n/2)$. The total cost is thus $3M(n/2) + D(n/2)$, instead of $2M(n/2) + 2D(n/2)$ for a full division with `RecursiveDivRem`. This gives $\frac{5}{3}M(n)$ for Karatsuba and $2.04M(n)$ for Toom-Cook 3-way. A similar algorithm is described in §2.3.2 (Subquadratic Montgomery Reduction) with further optimizations.

1.4.7 Division by a Constant

As for multiplication, division by a constant c is an important special case. It arises for example in Toom-Cook multiplication, where one has to perform an exact division by 3 (§1.3.3). We assume here that we want to divide a multiprecision number by a one-word constant. One could of course use a classical division algorithm (§1.4.1). Algorithm **ConstantDivide** performs a modular division:

$$A + b\beta^n = cQ,$$

where the "carry" b will be zero when the division is exact.

Theorem 1.4.3 *The output of Algorithm* ***ConstantDivide*** *satisfies* $A + b\beta^n = cQ$.

Proof. We show that after step i, $0 \le i < n$, we have $A_i + b\beta^{i+1} = cQ_i$, where $A_i := \sum_{j=0}^{i} a_i\beta^i$ and $Q_i := \sum_{j=0}^{i} q_i\beta^i$. For $i = 0$, this is $a_0 + b\beta = cq_0$, which is exactly line 10: since $q_0 = a_0/c \bmod \beta$, $q_0c - a_0$ is divisible by β. Assume

```
1  Algorithm ConstantDivide.
2  Input: A = sum_0^{n-1} a_i β^i,  0 ≤ c < β.
3  Output: Q = sum_0^{n-1} q_i β^i  and  0 ≤ b < c  such that  A + bβ^n = cQ
4  d ← 1/c mod β
5  b ← 0
6  for i from 0 to n − 1 do
7      if b ≤ a_i then (x, b') ← (a_i − b, 0)
8      else (x, b') ← (a_i − b + β, 1)
9      q_i ← dx mod β
10     b'' ← (q_i c − x)/β
11     b ← b' + b''
12 Return sum_0^{n-1} q_i β^i,  b.
```

now that $A_{i-1} + b\beta^i = cQ_{i-1}$ holds for $1 \le i < n$. We have $a_i - b + b'\beta = x$, then $x + b''\beta = cq_i$, thus $A_i + (b' + b'')\beta^{i+1} = A_{i-1} + \beta^i(a_i + b'\beta + b''\beta) = cQ_{i-1} - b\beta^i + \beta^i(x + b - b'\beta + b'\beta + b''\beta) = cQ_{i-1} + \beta^i(x + b''\beta) = cQ_i$. □

Remark: at line 10, since $0 \le x < \beta$, b'' can also be obtained as $\lfloor q_i c/\beta \rfloor$.

Algorithm **ConstantDivide** is just a special case of Hensel's division, which is the topic of the next section.

1.4.8 Hensel's Division

Classical division consists in cancelling the most significant part of the dividend by a multiple of the divisor, while Hensel's division cancels the least significant part (Fig. 1.2). Given a dividend A of $2n$ words and a divisor B of n words, the classical or MSB (most significant bit) division computes a quotient Q and a remainder R such that $A = QB + R$, while Hensel's or LSB (least significant bit) division computes a LSB-quotient Q' and a LSB-remainder R' such that $A = Q'B + R'\beta^n$. While the MSB division requires the most significant bit of B to be set, the LSB division requires B to be relatively prime to the word base β, i.e., B to be odd for β a power of two.

The LSB-quotient is uniquely defined by $Q' = A/B \bmod \beta^n$, with $0 \le Q' < \beta^n$. This in turn uniquely defines the LSB-remainder $R' = (A - Q'B)\beta^{-n}$, with $-B < R' < \beta^n$.

Most MSB-division variants (naive, with preconditioning, divide and conquer, Newton's iteration) have their LSB-counterpart. For example the

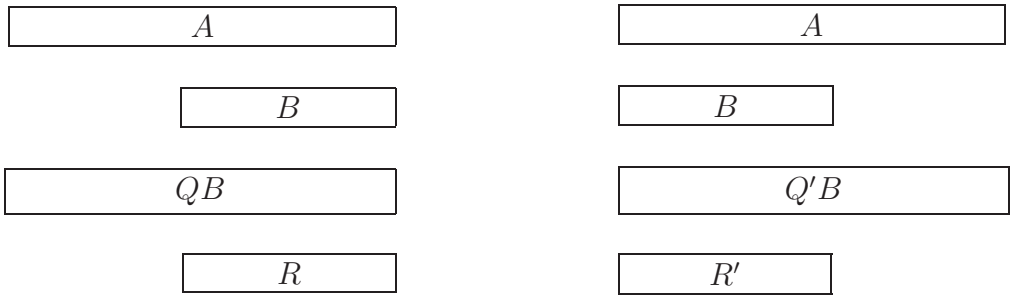


Figure 1.2: Classical/MSB division (left) vs Hensel/LSB division (right).

LSB preconditioning consists in using a multiple of the divisor such that $kB \equiv 1 \bmod \beta$, and Newton's iteration is called Hensel lifting in the LSB case. The exact division algorithm described at the end of §1.4.5 uses both MSB- and LSB-division simultaneously. One important difference is that LSB-division does not need any correction step, since the carries go in the direction opposite to the cancelled bits.

When only the remainder is wanted, Hensel's division is usually known as Montgomery reduction (see §2.3.2).

1.5 Roots

1.5.1 Square Root

The "paper and pencil" method once taught at school to extract square roots is very similar to "paper and pencil" division. It decomposes an integer m of the form $s^2 + r$, taking two digits of m at a time, and finding one digit of s for each two digits of m. It is based on the following idea: if $m = s^2 + r$ is the current decomposition, when taking two more digits of the root-end[2], we have a decomposition of the form $100m + r' = 100s^2 + 100r + r'$ with $0 \le r' < 100$. Since $(10s + t)^2 = 100s^2 + 20st + t^2$, a good approximation to the next digit t can be found by dividing $10r$ by $2s$.

Algorithm **SqrtRem** generalizes this idea to a power β^ℓ of the internal base close to $m^{1/4}$: one obtains a divide and conquer algorithm, which is in fact an error-free variant of Newton's method (*cf* Chapter 4):

[2]Input of the root operation, like the divid-end for division.

```
1  Algorithm SqrtRem.
2  Input: m = a_{n-1}β^{n-1} + ··· + a_1β + a_0 with a_{n-1} ≠ 0
3  Output: (s, r) such that s^2 ≤ m = s^2 + r < (s+1)^2
4  l ← ⌊(n-1)/4⌋
5  if l = 0 then return BasecaseSqrtRem(m)
6  write m = a_3β^{3l} + a_2β^{2l} + a_1β^l + a_0 with 0 ≤ a_2, a_1, a_0 < β^l
7  (s', r') ← SqrtRem(a_3β^l + a_2)
8  (q, u) ←DivRem(r'β^l + a_1, 2s')
9  s ← s'β^l + q
10 r ← uβ^l + a_0 − q^2
11 if r < 0 then
12     r ← r + 2s − 1
13     s ← s − 1
14 Return (s, r)
```

Theorem 1.5.1 *Algorithm **SqrtRem** correctly returns the integer square root s and remainder r of the input m, and has complexity $R(2n) \sim R(n) + D(n) + S(n)$ where $D(n)$ and $S(n)$ are the complexities of the division with remainder and squaring respectively. This gives $R(n) \sim \frac{1}{2}n^2$ with naive multiplication, $R(n) \sim \frac{4}{3}K(n)$ with Karatsuba's multiplication, assuming $S(n) \sim \frac{2}{3}M(n)$.*

As an example, assume Algorithm **SqrtRem** is called on $m = 123456789$ with $\beta = 10$. One has $n = 9$, $l = 2$, $a_3 = 123$, $a_2 = 45$, $a_1 = 67$, and $a_0 = 89$. The recursive call for $a_3\beta^l + a_2 = 12345$ yields $s' = 111$ and $r' = 24$. The **DivRem** call yields $q = 11$ and $u = 25$, which gives $s = 11111$ and $r = 2468$.

Another nice way to compute the integer square root of an integer n, i.e., $\lfloor n^{1/2} \rfloor$, is Algorithm **SqrtInt**, which is an all-integer version of Newton's method (§4.2).

Still with input 123456789, we successively get $s = 61728395$, 30864198, 15432100, 7716053, 3858034, 1929032, 964547, 482337, 241296, 120903, 60962, 31493, 17706, 12339, 11172, 11111, 11111. The convergence is slow because the initial value of s is much too large. However, any initial value greater or equal to $n^{1/2}$ works (see the proof of Algorithm **RootInt** below): starting from $s = 12000$, one gets $s = 11144$ then $s = 11111$.

```
1  Algorithm SqrtInt.
2  Input: an integer n ≥ 1.
3  Output: s = ⌊n^{1/2}⌋.
4  u ← n
5  repeat
6      s ← u
7      t ← s + ⌊n/s⌋
8      u ← ⌊t/2⌋
9  until u ≥ s
10 Return s
```

1.5.2 k-th Root

The idea of algorithm **SqrtRem** for the integer square root can be generalized to any power: if the current decomposition is $n = n'\beta^k + n''\beta^{k-1} + n'''$, first compute a k-th root of n', say $n' = s^k + r$, then divide $r\beta + n''$ by ks^{k-1} to get an approximation of the next root digit t, and correct it if needed. Unfortunately the computation of the remainder, which is easy for the square root, involves $O(k)$ terms for the k-th root, and this method may be slower than Newton's method with floating-point arithmetic (§4.2.3).

Similarly, algorithm **SqrtInt** can be generalized to the k-th root:

```
1  Algorithm RootInt.
2  Input: integers n ≥ 1, and k ≥ 2
3  Output: s = ⌊n^{1/k}⌋
4  u ← n
5  repeat
6      s ← u
7      t ← (k − 1)s + ⌊n/s^{k−1}⌋
8      u ← ⌊t/k⌋
9  until u ≥ s
10 Return s
```

Theorem 1.5.2 *Algorithm **RootInt** terminates and returns $\lfloor n^{1/k} \rfloor$.*

Proof. As long as $u < s$ in line 9, the sequence of s-values is decreasing, thus it suffices to consider what happens when $u \geq s$. First it is easy so see that $u \geq s$ implies $n \geq s^k$. Consider now the function $f(t) := [(k-1)t + n/t^{k-1}]/k$ for $t > 0$; its derivative is negative for $t < n^{1/k}$, and positive for $t > n^{1/k}$, thus $f(t) \geq f(n^{1/k}) = n^{1/k}$. This proves that $s \geq \lfloor n^{1/k} \rfloor$. Together with $s \leq n^{1/k}$, this proves that $s = \lfloor n^{1/k} \rfloor$ at the end of the algorithm. □

Note that any initial value $\geq \lfloor n^{1/k} \rfloor$ works. This also proves the correctness of Algorithm **SqrtInt** which is just the special case $k = 2$.

1.5.3 Exact Root

When a k-th root is known to be exact, there is of course no need to compute exactly the final remainder in "exact root" algorithms, which saves some computation time. However, one has to check that the remainder is sufficiently small that the computed root is correct.

When a root is known to be exact, one may also try to compute it starting from the least significant bits, as for exact division. Indeed, if $s^k = n$, then $s^k = n \bmod \beta^\ell$ for any integer ℓ. However, in the case of exact division, the equation $a = qb \bmod \beta^\ell$ has only one solution q as soon as b is relatively prime to β. Here, the equation $s^k = n \bmod \beta^\ell$ may have several solutions, so the lifting process is not unique. For example, $x^2 = 1 \bmod 2^3$ has four solutions $1, 3, 5, 7$.

Suppose we have $s^k = n \bmod \beta^\ell$, and we want to lift to $\beta^{\ell+1}$. This implies $(s + t\beta^\ell)^k = n + n'\beta^\ell \bmod \beta^{\ell+1}$ where $0 \leq t, n' < \beta$. Thus

$$kt = n' + \frac{n - s^k}{\beta^\ell} \bmod \beta.$$

This equation has a unique solution t when k is relatively prime to β. For example we can extract cube roots in this way for β a power of two. When k is relatively prime to β, we can also compute the root simultaneously from the most significant and least significant ends, as for the exact division.

Unknown Exponent

Assume now that one wants to check if a given integer n is an exact power, without knowing the corresponding exponent. For example, many factorization algorithms fail when given an exact power, therefore this case has to

be checked first. Algorithm **IsPower** detects exact powers, and returns the largest corresponding exponent. To quickly detect non-k-th powers at step 5,

```
1 Algorithm IsPower.
2 Input: a positive integer n.
3 Output: k ≥ 2 when n is an exact k–th power, 1 otherwise.
4 for k from ⌊log_2 n⌋ downto 2 do
5     if n is a k–th power, return k
6 Return 1.
```

one may use modular algorithms when k is relatively prime to the base β (see above).

REMARK: in Algorithm **IsPower**, one can limit the search to prime exponents k, but then the algorithm does not necessarily return the largest exponent, and we might have to call it again. For example, taking $n = 117649$, the algorithm will first return 3 because $117649 = 49^3$, and when called again with 49 it will return 2.

1.6 Greatest Common Divisor

Many algorithms for computing gcds may be found in the literature. We can distinguish between the following (non-exclusive) types:

- left-to-right (MSB) versus right-to-left (LSB) algorithms: in the former the actions depend on the most significant bits, while in the latter the actions depend on the least significant bits;
- naive algorithms: these $O(n^2)$ algorithms consider one word of each operand at a time, trying to guess from them the first quotients; we count in this class algorithms considering double-size words, namely Lehmer's algorithm and Sorenson's k-ary reduction in the left-to-right and right-to-left cases respectively; algorithms not in that class consider a number of words that depends on the input size n, and are often subquadratic;
- subtraction-only algorithms: these algorithms trade divisions for subtractions, at the cost of more iterations;

- plain versus extended algorithms: the former just compute the gcd of the inputs, while the latter express the gcd as a linear combination of the inputs.

1.6.1 Naive GCD

For completeness we mention Euclid's algorithm for finding the gcd of two non-negative integers u, v:

$$\text{while } v \neq 0 \text{ do } (u, v) \leftarrow (v, u \bmod v); \text{ Return } u.$$

Euclid's algorithm is discussed in many textbooks, e.g., [81], and we do not recommend it in its simplest form, except for testing purposes. Indeed, it is a slow way to compute a gcd, except for very small inputs.

Double-Digit Gcd. A first improvement comes from Lehmer's observation: the first few quotients in Euclid's algorithm usually can be determined from the two most significant words of the inputs. This avoids expensive divisions that give small quotients most of the time (see [81, §4.5.3]). Consider for example $a = 427,419,669,081$ and $b = 321,110,693,270$ with 3-digit words. The first quotients are $1, 3, 48, \ldots$ Now if we consider the most significant words, namely 427 and 321, we get the quotients $1, 3, 35, \ldots$. If we stop after the first two quotients, we see that we can replace the initial inputs by $a - b$ and $-3a + 4b$, which gives $106,308,975,811$ and $2,183,765,837$.

Lehmer's algorithm determines cofactors from the most significant words of the input integers. Those cofactors usually have size only half a word. The **DoubleDigitGcd** algorithm — which should be called "double-word" — uses the *two* most significant words instead, which gives cofactors t, u, v, w of one full-word each. This is optimal for the computation of the four products ta, ub, va, wb. With the above example, if we consider $427,419$ and $321,110$, we find that the first five quotients agree, so we can replace a, b by $-148a + 197b$ and $441a - 587b$, i.e., $695,550,202$ and $97,115,231$.

The subroutine `HalfBezout` takes as input two 2-word integers, performs Euclid's algorithm until the smallest remainder fits in one word, and returns the corresponding matrix $[t, u; v, w]$.

Binary Gcd. A better algorithm than Euclid's, though still having $O(n^2)$ complexity, is the *binary* algorithm. It differs from Euclid's algorithm in two

```
1 Algorithm DoubleDigitGcd.
2 Input: a := a_{n-1}β^{n-1} + ··· + a_0 , b := b_{m-1}β^{m-1} + ··· + b_0 .
3 Output: gcd(a, b).
4 if b = 0 then return a
5 if m < 2 then return BasecaseGcd(a, b)
6 if a < b or n > m then return DoubleDigitGcd(b, a mod b)
7 (t, u, v, w) ←HalfBezout(a_{n-1}β + a_{n-2}, b_{n-1}β + b_{n-2})
8 Return DoubleDigitGcd(|ta + ub|, |va + wb|).
```

ways: it consider least significant bits first, and it avoids divisions, except for divisions by two (which can be implemented as shifts on a binary computer).

```
1  Algorithm BinaryGcd.
2  Input: a, b > 0.
3  Output: gcd(a, b).
4  i ← 0
5  while a mod 2 = b mod 2 = 0 do
6      (i, a, b) ← (i + 1, a/2, b/2)
7  while a mod 2 = 0 do
8      a ← a/2
9  while b mod 2 = 0 do
10     b ← b/2
11 while a ≠ b do
12     (a, b) ← (|a − b|, min(a, b))
13     repeat a ← a/2 until a mod 2 ≠ 0
14 Return 2^i · a.
```

Sorenson's k-ary reduction

The binary algorithm is based on the fact that if a and b are both odd, then $a - b$ is even, and we can remove a factor of two since 2 does not divide $\gcd(a, b)$. Sorenson's k-ary reduction is a generalization of that idea: given a and b odd, we try to find small integers u, v such that $ua - vb$ is divisible by a large power of two.

Theorem 1.6.1 [130] *If $a, b > 0$ and $m > 1$ with $\gcd(a, m) = \gcd(b, m) = 1$, there exist u, v, $0 < |u|, v < \sqrt{m}$ such that $ua \equiv vb \bmod m$.*

Algorithm **ReducedRatMod** finds such a pair (u, v); it is a simple variation

```
1  Algorithm ReducedRatMod.
2  Input: a, b > 0, m > 1 with gcd(a, m) = gcd(b, m) = 1
3  Output: (u, v) such that 0 < |u|, v < √m and ua ≡ vb mod m
4  c ← a/b mod m
5  (u1, v1) ← (0, m)
6  (u2, v2) ← (1, c)
7  while v2 ≥ √m do
8      q ← ⌊v1/v2⌋
9      (u1, u2) ← (u2, u1 − qu2)
10     (v1, v2) ← (v2, v1 − qv2)
11 return (u2, v2).
```

of the extended Euclidean algorithm; indeed, the u_i are denominators from the continued fraction expansion of c/m.

When m is a prime power, the inversion $1/b \bmod m$ at line 4 of Algorithm **ReducedRatMod** can be performed efficiently using Hensel lifting (§2.4).

Given two integers a, b of say n words, Algorithm **ReducedRatMod** with $m = \beta^2$ will yields two integers u, v such that $vb - ua$ is a multiple of β^2. Since u, v have at most one-word each, $a' = (vb - ua)/\beta^2$ has at most $n - 1$ words — plus possible one bit — therefore with $b' = b \bmod a'$ one obtains $\gcd(a, b) = \gcd(a', b')$, where both a' and b' have one word less. This yields a LSB variant of the double-digit (MSB) algorithm.

1.6.2 Extended GCD

Algorithm **ExtendedGcd** (Table 1.6.2) solves the *extended* greatest common divisor problem: given two integers a and b, it computes their gcd g, and also two integers u and v (called *Bézout coefficients* or sometimes *cofactors* or *multipliers*) such that $g = ua+vb$. If a_0 and b_0 are the input numbers, and a, b the current values, the following invariants hold at the start of each iteration of the while loop (step 6) and after the while loop (step 12): $a = ua_0 + vb_0$, and $b = wa_0 + xb_0$.

An important special case is modular inversion (see Chapter 2): given an integer n, one wants to compute $1/a \bmod n$ for a relatively prime to n. One then simply runs algorithm **ExtendedGcd** with input a and $b = n$: this

```
1  Algorithm ExtendedGcd.
2  Input: integers a and b.
3  Output: integers (g,u,v) such that g = gcd(a,b) = ua + vb.
4  (u,w) ← (1,0)
5  (v,x) ← (0,1)
6  while b ≠ 0 do
7      (q,r) ← DivRem(a,b)
8      (a,b) ← (b,r)
9      (u,w) ← (w,u − qw)
10     (v,x) ← (x,v − qx)
11 Return (a,u,v).
```

yields u and v with $ua + vn = 1$, thus $1/a \equiv u \bmod n$. Since v is not needed here, we can simply avoid computing v and x, by removing lines 5 and 10.

It may also be worthwhile to compute only u in the general case, as the cofactor v can be recovered from $v = (g - ua)/b$; this division is exact (see §1.4.5).

All known algorithms for subquadratic gcd rely on an extended gcd subroutine, so we discuss the subquadratic extended gcd in the next section.

1.6.3 Half GCD, Divide and Conquer GCD

Designing a subquadratic integer gcd algorithm that is both mathematically correct and efficient in practice appears to be quite a challenging problem.

A first remark is that, starting from n-bit inputs, there are $O(n)$ terms in the remainder sequence $r_0 = a$, $r_1 = b$, ..., $r_{i+1} = r_{i-1} \bmod r_i$, ..., and the size of r_i decreases linearly with i. Thus computing all the partial remainders r_i leads to a quadratic cost, and a fast algorithm should avoid this.

However, the partial quotients $q_i = r_{i-1} \operatorname{div} r_i$ are usually small: the main idea is thus to compute them without computing the partial remainders. This can be seen as a generalization of the **DoubleDigitGcd** algorithm: instead of considering a fixed base β, adjust it so that the inputs have four "big words". The cofactor-matrix returned by the **HalfBezout** subroutine will then reduce the input size to about $3n/4$. A second call with the remaining two most significant "big words" of the new remainders will reduce their size to half the input size. This gives rise to the **HalfGcd** algorithm. Note that there are several possible definitions of the half-gcd. Given two n-bit

```
1  Algorithm HalfGcd.
2  Input: a ≥ b > 0
3  Output: a 2×2 matrix R and a',b' such that [a' b']^t = R [a b]^t
4  If a is small, use ExtendedGcd.
5  n ← nbits(a), k ← ⌊n/2⌋
6  a := a_1 2^k + a_0, b := b_1 2^k + b_0
7  S, a_2, b_2 ← HalfGcd(a_1, b_1)
8  a' ← a_2 2^k + S_11 a_0 + S_12 b_0
9  b' ← b_2 2^k + S_21 a_0 + S_22 b_0
10 ℓ ← ⌊k/2⌋
11 a' := a'_1 2^ℓ + a'_0, b' := b'_1 2^ℓ + b_0
12 T, a'_2, b'_2 ← HalfGcd(a'_1, b'_1)
13 a'' ← a'_2 2^ℓ + T_11 a'_0 + T_12 b'_0
14 b'' ← b'_2 2^ℓ + T_21 a'_0 + T_22 b'_0
15 Return S·T, a'', b''.
```

integers a and b, **HalfGcd** returns two consecutive elements a', b' of their remainder sequence with bit-size about $n/2$, (the different definitions differ in the exact stopping criterion). In some cases, it is convenient also to return the unimodular matrix R such that $[a'\ b']^t = R\,[a\ b]^t$, where R has entries of bit-size $n/2$.

Let $H(n)$ be the complexity of **HalfGcd** for inputs of n bits: a_1 and b_1 have $n/2$ bits, thus the coefficients of S and a_2, b_2 have $n/4$ bits. Thus a', b' have $3n/4$ bits, a'_1, b'_1 have $n/2$ bits, a'_0, b'_0 have $n/4$ bits, the coefficients of T and a'_2, b'_2 have $n/4$ bits, and a'', b'' have $n/2$ bits. We have $H(n) \sim 2H(n/2) + 4M(n/4, n/2) + 4M(n/4) + 8M(n/4)$, i.e., $H(n) \sim 2H(n/2) + 20M(n/4)$. If we do not need the final matrix $S \cdot T$, then we have $H^*(n) \sim H(n) - 8M(n/4)$. For the plain gcd, which simply calls **HalfGcd** until b is sufficiently small to call a naive algorithm, the corresponding cost $G(n)$ satisfies $G(n) = H^*(n) + G(n/2)$.

An application of the half gcd *per se* is the integer reconstruction problem. Assume one wants to compute a rational p/q where p and q are known to be bounded by some constant c. Instead of computing with rationals, one may perform all computations modulo some integer $n > c^2$. Hence one will end up with $p/q \equiv m \bmod n$, and the problem is now to find the unknown p and q from the known integer m. To do this, one starts an extended gcd from m and n, and one stops as soon as the current a and u are smaller than c: since

	naive	Karatsuba	Toom-Cook	FFT
$H(n)$	2.5	6.67	9.52	$5\log_2 n$
$H^*(n)$	2.0	5.78	8.48	$5\log_2 n$
$G(n)$	2.67	8.67	13.29	$10\log_2 n$

Table 1.1: Cost of **HalfGcd**: $H(n)$ with the cofactor matrix, $H^*(n)$ without the cofactor matrix, $G(n)$ the plain gcd, all in units of the multiplication cost $M(n)$, for naive multiplication, Karatsuba, Toom-Cook and FFT.

we have $a = um + vn$, this gives $m \equiv -a/u \bmod n$. This is exactly what is called a half-gcd; a subquadratic version is given above.

Subquadratic Binary GCD

The binary gcd can also be made fast, i.e., subquadratic in n. The basic idea is to mimic the left-to-right version, by defining an appropriate right-to-left division (Algorithm **BinaryDivide**).

```
1 Algorithm BinaryDivide.
2 Input: a, b ∈ ℤ with ν(b) − ν(a) = j > 0
3 Output: |q| < 2^j and r = a + q2^{−j}b such that ν(b) < ν(r)
4 b′ ← 2^{−j}b
5 q ← −a/b′ mod 2^{j+1}
6 if q ≥ 2^j then q ← q − 2^{j+1}
7 Return q, r = a + q2^{−j}b.
```

The integer q is the binary quotient of a and b, and r is the binary remainder.

This right-to-left division defines a right-to-left remainder sequence $a_0 = a$, $a_1 = b$, ..., where $a_{i+1} =$**BinaryRemainder**(a_{i-1}, a_i), and $\nu(a_{i+1}) < \nu(a_i)$. It can be shown that this sequence eventually reaches $a_{i+1} = 0$ for some index i. Assuming $\nu(a) = 0$, then $\gcd(a, b)$ is the odd part of a_i. Indeed, in Algorithm **BinaryDivide**, if some odd prime divides both a and b, it certainly divides $2^{-j}b$ which is an integer, and thus it divides $a + q2^{-j}b$. Reciprocally, if some odd prime divides both b and r, it divides also $2^{-j}b$, thus it divides $a = r - q2^{-j}b$; this shows that no spurious factor appears, unlike in other gcd algorithms.

EXAMPLE: let $a = 935$ and $b = 714$. We have $\nu(b) = \nu(a)+1$, thus Algorithm **BinaryDivide** computes $b' = 357$, $q = 1$, and $a_2 = a + q2^{-j}b = 1292$. The next step gives $a_3 = 1360$, then $a_4 = 1632$, $a_5 = 2176$, $a_6 = 0$. Since $2176 = 2^7 \cdot 17$, we conclude that the gcd of 935 and 714 is 17.

The corresponding gcd algorithm is quite similar to Algorithm **HalfGcd**, except it selects the low significant parts for the recursive calls, and uses **BinaryDivide** instead of the classical division. See the references in §1.9 for more details.

1.7 Base Conversion

Since computers usually work with binary numbers, and human prefer decimal representations, input/output base conversions are needed. In a typical computation, there will be only few conversions, compared to the total number of operations, thus optimizing conversions is less important. However, when working with huge numbers, naive conversion algorithms — which several software packages have — may slow down the whole computation.

In this section we consider that numbers are represented internally in base β — usually a power of 2 — and externally in base B — say a power of 10. When both bases are *commensurable*, i.e., both are powers of a common integer, like 8 and 16, conversions of n-digit numbers can be performed in $O(n)$ operations. We assume here that β and B are not commensurable.

One might think that only one algorithm is needed, since input and output are symmetric by exchanging bases β and B. Unfortunately, this is not true, since computations are done in base β only (see Ex. 1.8.29).

1.7.1 Quadratic Algorithms

Algorithms **IntegerInput** and **IntegerOutput** respectively read and write n-word integers, both with a complexity of $O(n^2)$.

1.7.2 Subquadratic Algorithms

Fast conversions routines are obtained using a "divide and conquer" strategy. For integer input, if the given string decomposes as $S = S_{\rm hi} \,||\, S_{\rm lo}$ where $S_{\rm lo}$ has k digits in base B, then

$$\mathrm{Input}(S, B) = \mathrm{Input}(S_{\rm hi}, B)B^k + \mathrm{Input}(S_{\rm lo}, B),$$

```
Algorithm IntegerInput.
Input: a string S = s_{m-1} ... s_1 s_0 of digits in base B
Output: the value A of the integer represented by S
A ← 0
for i from m − 1 downto 0 do
    A ← BA + val(s_i)
Return A.
```

```
Algorithm IntegerOutput.
Input: A = Σ_0^{n-1} a_i β^i
Output: a string S of characters, representing A in base B
m ← 0
while A ≠ 0
    s_m ← char(A mod B)
    A ← A div B
    m ← m + 1
Return S = s_{m-1} ... s_1 s_0.
```

where $\mathrm{Input}(S, B)$ is the value obtained when reading the string S in the external base B. Algorithm **FastIntegerInput** shows a possible way to implement this: If the output A has n words, algorithm **FastIntegerInput** has complexity $O(M(n)\log n)$, more precisely $\sim \frac{1}{2}M(n/2)\lg n$ for n a power of two in the FFT range (see Ex. 1.8.26).

For integer output, a similar algorithm can be designed, replacing multiplications by divisions. Namely, if $A = A_{\mathrm{hi}}B^k + A_{\mathrm{lo}}$, then

$$\mathrm{Output}(A, B) = \mathrm{Output}(A_{\mathrm{hi}}, B) \,||\, \mathrm{Output}(A_{\mathrm{lo}}, B),$$

where $\mathrm{Output}(A, B)$ is the string resulting from writing the integer A in the external base B, $S_1 \,||\, S_0$ denotes the concatenation of S_1 and S_0, and it is assumed that $\mathrm{Output}(A_{\mathrm{lo}}, B)$ has exactly k digits, after possibly padding with leading zeros.

If the input A has n words, algorithm **FastIntegerOutput** has complexity $O(M(n)\log n)$, more precisely $\sim \frac{1}{2}D(n/2)\lg n$ for n a power of two in the FFT range, where $D(n/2)$ is the cost of dividing an n-word integer by an $n/2$-word integer. Depending on the cost ratio between multiplication and division, integer output may thus be 2 to 5 times slower than integer input; see however Ex. 1.8.27.

```
Algorithm FastIntegerInput.
Input: a string S = s_{m-1} ... s_1 s_0 of digits in base B
Output: the value A of the integer represented by S
ℓ ← [val(s_0), val(s_1), ..., val(s_{m-1})]
(b, k) ← (B, m)
while k > 1 do
    if k even then ℓ ← [ℓ_1 + bℓ_2, ℓ_3 + bℓ_4, ..., ℓ_{k-1} + bℓ_k]
    else ℓ ← [ℓ_1 + bℓ_2, ℓ_3 + bℓ_4, ..., ℓ_k]
    (b, k) ← (b^2, ⌈k/2⌉)
Return ℓ_1.
```

```
Algorithm FastIntegerOutput.
Input: A = Σ_0^{n-1} a_i β^i
Output: a string S of characters, representing A in base B
if A < B then char(A)
else
    find k such that B^{2k-2} ≤ A < B^{2k}
    (Q, R) ← DivRem(A, B^k)
    r ← FastIntegerOutput(R)
    FastIntegerOutput(Q)||0^{k-len(r)}||r
```

1.8 Exercises

Exercise 1.8.1 Extend the Kronecker-Schönhage trick from the beginning of §1.3 to negative coefficients, assuming the coefficients are in the range $[-\rho, \rho]$.

Exercise 1.8.2 (Harvey [68]) For multiplying two polynomials of degree less than n, with non-negative integer coefficients bounded above by ρ, the Kronecker-Schönhage trick performs one integer multiplication of size about $2n \lg \rho$, assuming n is small compared to ρ. Show that it is possible to perform two integer multiplications of size $n \lg \rho$ instead, and even four integer multiplications of size $\frac{1}{2} n \lg \rho$.

Exercise 1.8.3 Assume your processor provides an instruction `fmaa`(a, b, c, d) returning h, l such that $ab + c + d = h\beta + l$ where $0 \leq a, b, c, d, l, h < \beta$. Rewrite Algorithm **BasecaseMultiply** using `fmaa`.

Exercise 1.8.4 (Hanrot) Prove that for $n_0 = 2$, the number $K(n)$ of word products in Karatsuba's algorithm as defined in Th. 1.3.2 is non-decreasing (caution:

this is no longer true with a larger threshold, for example with $n_0 = 8$ we have $K(7) = 49$ whereas $K(8) = 48$). Plot the graph of $\frac{K(n)}{n^{\log_2 3}}$ with a logarithmic scale for n, for $2^7 \leq n \leq 2^{10}$, and find experimentally where the maximum appears.

Exercise 1.8.5 (Ryde) Assume the basecase multiply costs $M(n) = an^2 + bn$, and that Karatsuba's algorithm costs $K(n) = 3K(n/2) + cn$. Show that dividing a by two increases the Karatsuba threshold n_0 by a factor of two, and on the contrary decreasing b and c decreases n_0.

Exercise 1.8.6 (Maeder [90], Thomé) Show that an auxiliary memory of $2n + o(n)$ words is enough to implement Karatsuba's algorithm in-place, for a $n \times n$ product. In the polynomial case, even prove that an auxiliary space of n coefficients is enough, in addition to the $n + n$ coefficients of the input polynomials, and the $2n - 1$ coefficients of the product. [It is allowed to use the $2n$ result words, but not to destroy the $n + n$ input words.]

Exercise 1.8.7 (Quercia, McLaughlin) Modify Alg. **KaratsubaMultiply** to use only $\sim \frac{7}{2}n$ additions/subtractions. [Hint: decompose C_0, C_1 and C_2 in two parts.]

Exercise 1.8.8 Design an in-place version of Algorithm **KaratsubaMultiply** (see Ex. 1.8.6) that accumulates the result in $c_0, \ldots, c_{2n-1}$, and returns a carry bit.

Exercise 1.8.9 (Vuillemin [129]) Design a program or circuit to compute a 3×2 product in 4 multiplications. Then use it to perform a 6×6 product in 16 multiplications. How does this compare asymptotically with Karatsuba and Toom-Cook 3-way?

Exercise 1.8.10 (Weimerskirch, Paar) Extend the Karatsuba trick to compute an $n \times n$ product in $\frac{n(n+1)}{2}$ multiplications and $\frac{(5n-2)(n-1)}{2}$ additions/subtractions. For which n does this win?

Exercise 1.8.11 (Hanrot) In Algorithm **OddEvenKaratsuba**, in case both m and n are odd, one combines the larger parts A_0 and B_0 together, and the smaller parts A_1 and B_1 together. Find a way to get instead:

$$K(m, n) = K(\lceil m/2 \rceil, \lfloor n/2 \rfloor) + K(\lfloor m/2 \rfloor, \lceil n/2 \rceil) + K(\lceil m/2 \rceil, \lceil n/2 \rceil).$$

Exercise 1.8.12 Prove that if 5 integer evaluation points are used for Toom-Cook 3-way, the division by 3 can not be avoided. Does this remain true if only 4 integer points are used together with ∞?

Exercise 1.8.13 (Quercia, Harvey) In Toom-Cook 3-way, take as evaluation point 2^w instead of 2 (§1.3.3), where w is the number of bits per word (usually $w = 32$ or 64). Which division is then needed? Same question for the evaluation point $2^{w/2}$.

Exercise 1.8.14 For multiplication of two numbers of size kn and n respectively, for an integer $k \geq 2$, show that the trivial strategy which performs k multiplications, each $n \times n$, is not always the best possible.

Exercise 1.8.15 (Karatsuba, Zuras [135]) Assuming the multiplication has superlinear cost, show that the speedup of squaring with respect to multiplication can not exceed 2.

Exercise 1.8.16 (Thomé, Quercia) Multiplication and the middle product are just special cases of linear forms programs: consider two set of inputs $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, and a set of outputs $c_1, \ldots, c_k$ that are sums of products of $a_i b_j$. For such a given problem, what is the least number of multiplies required? As an example, can we compute $x = au + cw, y = av + bw, z = bu + cv$ in less than 6 multiplies? Same question for $x = au - cw, y = av - bw, z = bu - cv$.

Exercise 1.8.17 In algorithm **BasecaseDivRem** (§1.4.1), prove that $q_j^* \leq \beta+1$. Can this bound be reached? In the case $q_j^* \geq \beta$, prove that the while-loop at steps 9-11 is executed at most once. Prove that the same holds for Svoboda's algorithm (§1.4.2), i.e., that $A \geq 0$ after step 11.

Exercise 1.8.18 (Granlund, Möller) In algorithm **BasecaseDivRem**, estimate the probability that $A < 0$ is true at line 9, assuming the remainder r_j from the division of $a_{n+j}\beta + a_{n+j-1}$ by b_{n-1} is uniformly distributed in $[0, b_{n-1} - 1]$, $A \bmod \beta^{n+j-1}$ is uniformly distributed in $[0, \beta^{n+j-1} - 1]$, and $B \bmod \beta^{n-1}$ is uniformly distributed in $[0, \beta^{n-1}-1]$. Then replace the computation of q_j^* by a division of the three most significant words of A by the two most significant words of B. Prove the algorithm is still correct; what is the maximal number of corrections, and the probability that $A < 0$ holds?

Exercise 1.8.19 (Montgomery [101]) Let $0 < b < \beta$, and $0 \leq a_4, \ldots, a_0 < \beta$. Prove that $a_4(\beta^4 \bmod b) + \cdots + a_1(\beta \bmod b) + a_0 < \beta^2$, as long as $b < \beta/3$. Use that fact to design an efficient algorithm dividing $A = a_{n-1}\beta^{n-1} + \cdots + a_0$ by b. Does that algorithm extend to division by the least significant digits?

Exercise 1.8.20 In Algorithm **RecursiveDivRem**, find inputs that require 1, 2, 3 or 4 corrections in step 11. [Hint: consider $\beta = 2$]. Prove that when $n = m$ and $A < \beta^m(B + 1)$, at most two corrections occur.

Exercise 1.8.21 Find the complexity of Algorithm **RecursiveDivRem** in the FFT range.

Exercise 1.8.22 Consider the division of A of kn words by B of n words, with integer $k \geq 3$, and the alternate strategy that consists in extending the divisor with zeros so that it has half the size of the dividend. Show this is always slower than Algorithm **UnbalancedDivision** [assuming the division has superlinear cost].

Exercise 1.8.23 An important special base of division is when the divisor is of the form b^k. This is useful for example for the output routine (§1.7). Can one design a fast algorithm for that case?

Exercise 1.8.24 (Sedoglavic) Does the Kronecker-Schönhage trick to reduce polynomial multiplication to integer multiplication (§1.3) also work — in an efficient way — for division? Assume for example you want to divide a degree-$2n$ polynomial $A(x)$ by a monic degree-n polynomial $B(x)$, both having integer coefficients bounded by ρ.

Exercise 1.8.25 Design an algorithm that performs an exact division of a $4n$-bit integer by a $2n$-bit integer, with a quotient of $2n$ bits, using the idea from the last paragraph of §1.4.5. Prove that your algorithm is correct.

Exercise 1.8.26 Find a formula $T(n)$ for the asymptotic complexity of Algorithm **FastIntegerInput** when $n = 2^k$ (§1.7.2), and show that, for general n, it is within a factor of two of $T(n)$. [Hint: consider the binary expansion of n]. Design another subquadratic algorithm that works top-down: is is faster?

Exercise 1.8.27 Show that asymptotically, the output routine can be made as fast as the input routine `FastIntegerInput`. [Hint: use Bernstein's scaled remainder tree and the middle product.] Experiment with it on your favorite multiple-precision software.

Exercise 1.8.28 If the internal base β and the external one B share a common divisor — as in the case $\beta = 2^l$ and $B = 10$ — show how one can exploit this to speed up the subquadratic input and output routines.

Exercise 1.8.29 Assume you are given two n-digit integers in base 10, but you have fast arithmetic in base 2 only. Can you multiply them in $O(M(n))$?

1.9 Notes and References

"Online" algorithms are considered in many books and papers, see for example the book by Borodin and El-Yaniv [18]. "Relaxed" algorithms were introduced by van der Hoeven. For references and a discussion of the differences between "lazy", "zealous" and "relaxed" algorithms, see [128].

An example of implementation with "guard bits" to avoid overflow problems in the addition (§1.2) is the block-wise modular arithmetic from Lenstra and Dixon on the MasPar [53], where they used $\beta = 2^{30}$ with 32-bit words.

The fact that polynomial multiplication reduces to integer multiplication is attributed to Kronecker, and was rediscovered by Schönhage [113]. Nice applications of the Kronecker-Schönhage trick are given in [120]. Very little is known about the *average* complexity of Karatsuba's algorithm. What is clear is that no simple asymptotic equivalent can be obtained, since the ratio $K(n)/n^\alpha$ does not converge. See Ex. 1.8.4.

A very good description of Toom-Cook algorithms can be found in [51, Section 9.5.1], in particular how to symbolically generate the evaluation and interpolation formulæ. Bodrato and Zanoni show that the Toom-Cook 3-way interpolation scheme from §1.3.3 is near from optimal — for the points $0, 1, -1, 2, \infty$; they also exhibit efficient 4-way and 5-way schemes [17].

The odd-even scheme is described in [66], and was independently discovered by Andreas Enge.

The exact division algorithm starting from least significant bits is due to Jebelean [74], who also invented with Krandick the "bidirectional" algorithm [82]. The Karp-Markstein trick to speed up Newton's iteration (or Hensel lifting over p-adic numbers) is described in [79]. The "recursive division" in §1.4.3 is from [41], although previous but not-so-detailed ideas can be found in [97] and [76]. The definition of Hensel's division used here is due to Shand and Vuillemin [117], who also point out the duality with Euclidean division.

Algorithm **SqrtRem** (§1.5.1) was first described in [134], and proven in [16]. Algorithm **SqrtInt** is described in [46]; its generalization to k-th roots (Algorithm **RootInt**) is due to Keith Briggs. The detection of exact powers is discussed in [46] and especially in [7, 13].

The classical (quadratic) Euclidean algorithm has been considered by many authors — a good reference is Knuth [81]. The binary gcd is almost as old as the classical Euclidean algorithm — Knuth [81] has traced it back to a first-century AD Chinese text, though it was rediscovered several times in the 20th century. The binary gcd has been analysed by Brent [24, 30],

Knuth [81], Maze [91] and Vallée [127]. A parallel (systolic) version that runs in $O(n)$ time using $O(n)$ processors was given by Brent and Kung [35].

The double-digit gcd is due to Jebelean [75]. The k-ary gcd reduction is due to Sorenson [119], and was improved and implemented in GNU MP by Weber, who also invented Algorithm **ReducedRatMod** [130].

The first subquadratic gcd algorithm was published by Knuth [80], but his analysis was suboptimal — he gave $O(n(\log n)^5(\log\log n))$ —, and the correct complexity was given by Schönhage [111]: for this reason the algorithm is sometimes called the Knuth-Schönhage algorithm. A description for the polynomial case can be found in [2], and a detailed but incorrect one for the integer case in [133]. The subquadratic binary gcd given here is due to Stehlé and Zimmermann [122].

Chapter 2

The FFT and Modular Arithmetic

In this Chapter our main topic is modular arithmetic, i.e., how to compute efficiently modulo a given integer N. In most applications, the modulus N is fixed, and special-purpose algorithms benefit from some precomputations depending on N only, that speed up arithmetic modulo N.

In this Chapter, unless explicitly stated, we consider that the modulus N occupies n words in the word-base β, i.e., $\beta^{n-1} \leq N < \beta^n$.

2.1 Representation

We consider in this section the different possible representations of residues modulo N. As in Chapter 1, we consider mainly dense representations.

2.1.1 Classical Representation

The classical representation stores a residue (class) a as an integer $0 \leq a < N$. Residues are thus always fully reduced, i.e., in *canonical* form.

Another non-redundant form consists in choosing a symmetric representation, say $-N/2 \leq a < N/2$. This form might save some reductions in additions or subtractions (see §2.2). Negative numbers might be stored either with a separate sign, or with a two's-complement representation.

Since N takes n words in base β, an alternative *redundant* representation consists in choosing $0 \leq a < \beta^n$ to represent a residue class. If the underlying arithmetic is word-based, this will yield no slowdown compared to the canonical form. An advantage of this representation is that when adding two residues, it suffices to compare their sum with β^n, and the result of this comparison is simply given by the carry bit of the addition (see Algorithm **IntegerAddition** in §1.2), instead of comparing the sum with N.

2.1.2 Montgomery's Form

Montgomery's form is another representation widely used when several modular operations have to be performed modulo the same integer N (additions, subtractions, modular multiplications). It implies a small overhead to convert — if needed — from the classical representation to Montgomery's and vice-versa, but this overhead is often more than compensated by the speedup obtained in the modular multiplication.

The main idea is to represent a residue a by $a' = aR \bmod N$, where $R = \beta^n$, and N takes n words in base β. Thus Montgomery is not concerned with the *physical* representation of a residue class, but with the *meaning* associated to a given physical representation. (As a consequence, the different choices mentioned above for the physical representation are all possible.) Addition and subtraction are unchanged, but (modular) multiplication translates to a different, much simpler, algorithm (§2.3.2).

In most applications using Montgomery's form, all inputs are first converted to Montgomery's form, using $a' = aR \bmod N$, then all computations are performed in Montgomery's form, and finally all outputs are converted back — if needed — to the classical form, using $a = a'/R \bmod N$. We need to assume that $(R, N) = 1$, or equivalently that $(\beta, N) = 1$, to ensure the existence of $1/R \bmod N$. This is not usually a problem because β is a power of two and N can be assumed to be odd.

2.1.3 Residue Number Systems

In a *Residue Number System*, a residue a is represented by a list of residues a_i modulo N_i, where the moduli N_i are coprime and their product is N. The integers a_i can be efficiently computed from a using a remainder tree, and the unique integer $0 \leq a < N = N_1 N_2 \cdots$ are computed from the a_i by an Explicit Chinese Remainder Theorem (§2.6). The residue number system is

classical (MSB)	p-adic (LSB)
Euclidean division	Hensel division, Montgomery reduction
Svoboda's algorithm	Montgomery-Svoboda
Euclidean gcd	binary gcd
Newton's method	Hensel lifting

Figure 2.1: Equivalence between LSB and MSB algorithms.

interesting since addition and multiplication can be performed in parallel on each small residue a_i. However this representation requires that N factors into convenient moduli $N_1, N_2, \ldots$, which is not always the case (see however §2.8).

2.1.4 MSB vs LSB Algorithms

Many classical (most significant bits first or MSB) algorithms have a p-adic (least significant bit first or LSB) equivalent form. Thus several algorithms in this Chapter are just LSB-variants of algorithms discussed in Chapter 1 (see Fig. 2.1).

2.1.5 Link with Polynomials

As in Chapter 1, a strong link exists between modular arithmetic and arithmetic on polynomials. One way of implementing finite fields $\mathbb{F}_q$ with $q = p^n$ is to work with polynomials in $\mathbb{F}_p[x]$, which are reduced modulo a monic irreducible polynomial $f(x) \in \mathbb{F}_p[x]$ of degree n. In this case modular reduction happens both at the coefficient level — i.e., in $\mathbb{F}_p$ — and at the polynomial level, i.e., modulo $f(x)$.

Some algorithms work in the ring $(\mathbb{Z}/nZ)[x]$, where n is a composite integer. An important case is Schönhage-Strassen's multiplication algorithm, where n has the form $2^\ell + 1$.

In both cases — $\mathbb{F}_p[x]$ and $(\mathbb{Z}/nZ)[x]$ —, the Kronecker-Schönhage trick (§1.3) can be applied efficiently. Indeed, since the coefficients are known to be bounded, by p and n respectively, and thus have a fixed size, the segmentation is quite efficient. If polynomials have degree d and coefficients are bounded by n, one obtains $O(M(d \log n))$ operations, instead of $M(d)M(\log n)$ with the classical approach. Also, the implementation is simpler, because we only

have to implement fast arithmetic for large integers instead of fast arithmetic at both the polynomial level and the coefficient level (see also Ex. 1.8.2).

2.2 Addition and Subtraction

The addition of two residues in classical representation is done as follows:

```
1 Algorithm ModularAdd.
2 Input: residues a,b with 0 ≤ a,b < N.
3 Output: c = a + b mod N.
4 c ← a + b
5 if c ≥ N then
6     c ← c − N
```

Assuming a and b are uniformly distributed in $[0, N-1]$, the subtraction $c \leftarrow c - N$ is performed with probability $(1 - 1/N)/2$. If we use instead a symmetric representation in $[-N/2, N/2)$, the probability that we need to add or subtract N drops to $1/4 + O(1/N^2)$at the cost of an additional test. This extra test might be expensive for small N — say one or two words — but will be relatively cheap as long as N uses say a dozen words.

2.3 Multiplication

Modular multiplication consists in computing $A \cdot B \bmod N$, where A and B are residues modulo N, which has n words in base β. We assume here that A and B have at most n words, and in some cases that they are fully reduced, i.e., $0 \le A, B < N$.

The naive algorithm first computes $C = AB$, which has at most $2n$ words, and then reduces C modulo N. When the modulus N is not fixed, the best known algorithms are those presented in Chapter 1 (§1.4.6). We consider here better algorithms that benefit from an invariant modulus. These algorithms are allowed to perform precomputations depending on N only. The cost of the precomputations is not taken into account: it is assumed negligible for many modular reductions. However, we assume that the amount of precomputed data uses only linear space, i.e., $O(\log N)$ memory.

Algorithms with precomputations include Barrett's algorithm (§2.3.1), which computes an approximation of the inverse of the modulus, thus trading

division for multiplication; Montgomery's algorithm, which corresponds to Hensel's division with remainder only (§1.4.8), and its subquadratic variant, which is the MSB-variant of Barrett's algorithm; and finally Mihailescu's algorithm (§2.3.3).

2.3.1 Barrett's Algorithm

Barrett's algorithm [6] is interesting when many divisions have to be made with the same divisor; this is the case when one performs computations modulo a fixed integer. The idea is to precompute an approximation of the divisor inverse. In such a way, an approximation of the quotient is obtained with just one multiplication, and the corresponding remainder after a second multiplication. A small number of corrections suffice to convert those approximations into exact values. For sake of simplicity, we describe Barrett's algorithm in base β; however, β might be replaced by any integer, in particular 2^n or β^n.

```
1 Algorithm BarrettDivRem.
2 Input: integers A, B with 0 ≤ A < β², β/2 < B < β.
3 Output: quotient Q and remainder R of A divided by B.
4 I ← ⌊β²/B⌋ [precomputation]
5 Q ← ⌊A₁I/β⌋ where A = A₁β + A₀ with 0 ≤ A₀ < β
6 R ← A − QB
7 while R ≥ B do
8     (Q, R) ← (Q + 1, R − B)
9 Return (Q, R).
```

Theorem 2.3.1 *Algorithm* ***BarrettDivRem*** *is correct and step 8 is performed at most* 3 *times.*

Proof. Since $A = QB+R$ is invariant in the algorithm, we just need to prove that $0 \le R < B$ at the end. We first consider the value of Q, R before the while-loop. Since $\beta/2 < B < \beta$, we have $\beta < \beta^2/B < 2\beta$, thus $\beta \le I < 2\beta$. We have $Q \le A_1 I/\beta \le A_1\beta/B \le A/B$. This ensures that R is nonnegative. Now $I > \beta^2/B - 1$, which gives

$$IB > \beta^2 - B. \tag{2.1}$$

Similarly, $Q > A_1 I/\beta - 1$ gives

$$\beta Q > A_1 I - \beta. \tag{2.2}$$

This yields $\beta QB > A_1IB - \beta B > A_1(\beta^2 - B) - \beta B = \beta(A - A_0) - B(\beta + A_1) > \beta A - 4\beta B$ since $A_0 < \beta < 2B$ and $A_1 < \beta$. We conclude that $A < B(Q+4)$, thus at most 3 corrections are needed. □

The bound of 3 corrections is tight: it is obtained for $A = 1980$, $B = 36$, $n = 6$. For this example $I = 113$, $A_1 = 30$, $Q = 52$, $R = 108 = 3B$.

The multiplications at steps 5 and 6 may be replaced by short products, more precisely that of step 5 by a high short product, and that of step 6 by a low short product (see §3.3).

Complexity of Barrett's Algorithm

If the multiplications at steps 5 and 6 are performed using full products, Barrett's algorithm costs $2M(n)$ for a divisor of size n. In the FFT range, this costs might be lowered to $1.5M(n)$ using the "wrap-around trick" (§3.4.1); moreover, if the forward transforms of I and B are stored, the cost decreases to $M(n)$, assuming one FFT equals three transforms.

2.3.2 Montgomery's Multiplication

Montgomery's algorithm is very efficient to perform modular arithmetic modulo a fixed modulus N. The main idea is to replace a residue $A \bmod N$ by $A' = \lambda A \bmod N$, where A' is the "Montgomery form" corresponding to the residue A. Addition and subtraction are unchanged, since $\lambda A + \lambda B \equiv \lambda(A + B) \bmod N$. The multiplication of two residues in Montgomery form does not give exactly what we want: $(\lambda A)(\lambda B) \neq \lambda(AB) \bmod N$. The trick is to replace the classical modular multiplication by "Montgomery's multiplication":

$$\mathbf{MontgomeryMul}(A', B') = \frac{A'B'}{\lambda} \bmod N.$$

For some values of λ, `MontgomeryMul`(A', B') can be easily computed, in particular for $\lambda = \beta^n$, where N uses n words in base β. Fig. 2.2 presents a quadratic algorithm (**REDC**) to compute `MontgomeryMul`(A', B') in that case, and a subquadratic reduction (**FastREDC**) is given in Fig. 2.3.

Another view of Montgomery's algorithm for $\lambda = \beta^n$ is to consider that it computes the remainder of Hensel's division (§1.4.8).

Theorem 2.3.2 *Algorithm **REDC** is correct.*

```
1 Algorithm REDC.
2 Input: 0 ≤ C < β^{2n}, N < β^n, μ ← −N^{−1} mod β, (β, N) = 1
3 Output: 0 ≤ R < β^n such that R = Cβ^{−n} mod N
4 for i from 0 to n − 1 do
5     q_i ← μc_i mod β
6     C ← C + q_i N β^i
7 R ← Cβ^{−n}
8 if R ≥ β^n then return R − N else return R.
```

Figure 2.2: Montgomery multiplication (quadratic non-interleaved version). The c_i form the current base-β decomposition of C, i.e., the c_i are defined by $C = \sum_0^{2n-1} c_i \beta^i$.

Proof. We first prove that $R \equiv C\beta^{-n} \bmod N$: C is only modified in line 6, which does not change $C \bmod N$, thus at line 7 we have $R \equiv C\beta^{-n} \bmod N$, which remains true in the last line.

Assume that for a given i, we have $C \equiv 0 \bmod \beta^i$ when entering step 5. Then since $q_i \equiv -c_i/N \bmod \beta$, we have $C + q_i N\beta^i \equiv 0 \bmod \beta^{i+1}$ at the next step, so $c_i = 0$. Thus when one exits the for-loop, C is a multiple of β^n, thus R is an integer at step 7.

Still at step 7, we have $C < \beta^{2n} + (\beta-1)N(1 + \beta + \cdots + \beta^{n-1}) = \beta^{2n} + N(\beta^n - 1)$ thus $R < \beta^n + N$, and $R - N < \beta^n$. □

Compared to classical division (Algorithm **BasecaseDivRem**, §1.4.1), Montgomery's algorithm has two significant advantages: the quotient selection is performed by a multiplication modulo the word base β, which is more efficient than a division by the most significant word b_{n-1} of the divisor as in **BasecaseDivRem**; and there is no repair step *inside* the for-loop — there is only one at the very end.

For example, with inputs $C = 766970544842443844$, $N = 862664913$, and $\beta = 1000$, Algorithm REDC precomputes $\mu = 23$, then we have $q_0 = 412$, which yields $C = C + 412N = 766970900260388000$; then $q_1 = 924$, which yields $C = C + 924N\beta = 767768002640000000$; then $q_2 = 720$, which yields $C = C + 720N\beta^2 = 1388886740000000000$. At step 7, $R = 1388886740$, and since $R \geq \beta^3$, one returns $R - N = 526221827$.

Since Montgomery's algorithm — i.e., Hensel's division with remainder only — can be viewed as an LSB variant of the classical division, Svoboda's

Algorithm **FastREDC**.
Input: $0 \le C < \beta^2$, $N < \beta$, $\mu \leftarrow -1/N \bmod \beta$
Output: $0 \le R < \beta$ such that $R = C/\beta \bmod N$
$Q \leftarrow \mu C \bmod \beta$
$R \leftarrow (C + QN)/\beta$
if $R \ge \beta$ **then** return $R - N$ **else** return R.

Figure 2.3: Subquadratic Montgomery multiplication.

divisor preconditioning (§1.4.2) also translates to the LSB context. More precisely, in Algorithm `REDC`, one wants to modify the divisor N so that the "quotient selection" $q \leftarrow \mu c_i \bmod \beta$ at step 5 becomes trivial. A natural choice is to have $\mu = 1$, which corresponds to $N \equiv -1 \bmod \beta$. The multiplier k used in Svoboda division is thus here simply the parameter μ in `REDC`. The Montgomery-Svoboda algorithm obtained works as follows:

1. first compute $N' = \mu N$, with $N' < \beta^{n+1}$;

2. then perform the $n - 1$ first loops of `REDC`, replacing μ by 1, and N by N';

3. perform a last classical loop with μ and N, and the last steps from `REDC`.

The quotient selection with Montgomery-Svoboda's algorithm thus simply consists in "reading" the word of weight β^i in the divisor C.

For the above example, one gets $N' = 19841292999$, q_0 is the least significant word of C, i.e., $q_0 = 844$, then $C = C + 844N' = 766987290893735000$; then $q_1 = 735$ and $C = C + 735N'\beta = 781570641248000000$; the last step gives $q_2 = 704$, and $C = C + 704N\beta^2 = 1388886740000000000$, which is the same value that we found above after the for-loop.

Subquadratic Montgomery Reduction

A subquadratic version of REDC is obtained by taking $n = 1$ in Algorithm **REDC**, and considering β as a "giant base" (alternatively, replace β by β^n below):

This is exactly the 2-adic counterpart of Barrett's subquadratic algorithm: steps 4-5 might be performed by a low short product and a high short product respectively.

When combined with Karatsuba's multiplication , assuming the products of steps 4-5 are full products, the reduction requires 2 multiplications of size n, i.e., 6 multiplications of size $n/2$ (n denotes the size of N, β being a giant base).

With some additional precomputation, the reduction might be performed with 5 multiplications of size $n/2$, assuming n is even. This is simply Montgomery-Svoboda's algorithm with N having two big words in base $\beta^{n/2}$:

1 Algorithm **MontgomerySvoboda2**.
2 **Input**: $0 \le C < \beta^{2n}$, $N < \beta^n$, $\mu \leftarrow -1/N \bmod \beta^{n/2}$, $N' = \mu N$
3 **Output**: $0 \le R < \beta^n$ such that $R = C/\beta^n \bmod N$
4 $q_0 \leftarrow C \bmod \beta^{n/2}$
5 $C \leftarrow (C + q_0 N')/\beta^{n/2}$
6 $q_1 \leftarrow \mu C \bmod \beta^{n/2}$
7 $R \leftarrow (C + q_1 N)/\beta^{n/2}$
8 **if** $R \ge \beta^n$ **then** return $R - N$ **else** return R.

The cost of the algorithm is $M(n, n/2)$ to compute $q_0 N'$ (even if N' has in principle $3n/2$ words, we know $N' = H\beta^{n/2} - 1$ with $H < \beta^n$, thus it suffices to multiply q_0 by H), $M(n/2)$ to compute $\mu C \bmod \beta^{n/2}$, and again $M(n, n/2)$ to compute $q_1 N$, thus a total of $5M(n/2)$ if each $n \times (n/2)$ product is realized by two $(n/2) \times (n/2)$ products.

The algorithm is quite similar to the one described at the end of §1.4.6, where the cost was $3M(n/2) + D(n/2)$ for a division of $2n$ by n with remainder only. The main difference here is that, thanks to Montgomery's form, the last classical division $D(n/2)$ in Svoboda's algorithm is replaced by multiplications of total cost $2M(n/2)$, which is usually faster.

Algorithm **MontgomerySvoboda2** can be extended as follows. The value C obtained after step 5 has $3n/2$ words, i.e., an excess of $n/2$ words. Instead of reducing that excess with `REDC`, one could reduce it using Svoboda's technique with $\mu' = -1/N \bmod \beta^{n/4}$, and $N'' = \mu' N$. This would reduce the low $n/4$ words from C at the cost of $M(n, n/4)$, and a last `REDC` step would reduce the final excess of $n/4$, which would give $D(2n, n) = M(n, n/2) + M(n, n/4) + M(n/4) + M(n, n/4)$. This "folding" process can be generalized to $D(2n, n) = M(n, n/2) + \cdots + M(n, n/2^k) + M(n/2^k) + M(n, n/2^k)$. If

Algorithm	Karatsuba	Toom-Cook 3-way	Toom-Cook 4-way
$D(n)$	$2M(n)$	$2.63M(n)$	$3.10M(n)$
1-folding	$1.67M(n)$	$1.81M(n)$	$1.89M(n)$
2-folding	$1.67M(n)$	$1.91M(n)$	$2.04M(n)$
3-folding	$1.74M(n)$	$2.06M(n)$	$2.25M(n)$

Figure 2.4: Theoretical complexity of subquadratic REDC with 1-, 2- and 3−folding, for different multiplication algorithms.

$M(n, n/2^k)$ reduces to $2^k M(n/2^k)$, this gives:

$$D(n) = 2M(n/2) + 4M(n/4) + \cdots + 2^{k-1}M(n/2^{k-1}) + (2^{k+1}+1)M(n/2^k).$$

Unfortunately, the resulting multiplications are more and more unbalanced, moreover one needs to store k precomputed multiples $\mathbb{N}', \mathbb{N}'', \ldots$ of N, each requiring at least n words (if one does not store the low part of those constants which all equal -1). Fig. 2.4 shows that the single-folding algorithm is the best one.

Exercise 2.7.3 discusses further possible improvements in Montgomery-Svoboda's algorithm, which achieve $D(n) = 1.58M(n)$ in the case of Karatsuba multiplication.

2.3.3 Mihailescu's Algorithm

Mihailescu's algorithm assumes one can perform fast multiplication modulo both $2^n - 1$ and $2^n + 1$, for sufficiently many values of n. This assumption is true for example with Schönhage-Strassen's algorithm: the original version multiplies two numbers modulo $2^n + 1$, but discarding the "twist" operations before and after the Fourier transforms computes their product modulo $2^n - 1$. (This has to be done at the top level only: the recursive operations compute modulo $2^{n'} + 1$ in both cases.)

The key idea in Mihailescu's algorithm is to perform a Montgomery reduction AB/R with R not being equal to a power β^n of the word base, but with $R = \beta^n - 1$. Modulo the latter value, efficient convolution can be performed using FFT.

Theorem 2.3.3 *Algorithm* ***MultiplyMihailescu*** *computes* $AB\beta^{-n} \bmod N$ *correctly, in* $\approx 1.5M(n)$ *operations, assuming multiplication modulo* $2^n \pm 1$ *costs* $\frac{1}{2}M(n)$*, or* 3 *Fourier transforms of size* n*.*

1 Algorithm **MultiplyMihailescu**.
2 **Input**: A, B with $0 \leq A, B < N < \beta^n$
3 **Output**: $AB/(2^n - 1) \bmod N$
4 Assumes: $\mu = N^{-1} \bmod (2^n - 1)$ is precomputed
5 $c \leftarrow AB \bmod (2^n - 1)$
6 $d \leftarrow AB \bmod (2^n + 1)$
7 $\gamma \leftarrow c\mu \bmod (2^n - 1)$
8 $\delta \leftarrow \gamma N \bmod (2^n + 1)$
9 Return $(\delta - d)/2 \bmod (2^n + 1)$

Proof. Let $AB = h(2^n-1)+\ell$. We want $AB/(2^n-1) = h+\ell/(2^n-1) \bmod N$. Then step 5 gets $c = \ell$ and step 6 computes $d = \ell - 2h \bmod (2^n + 1)$. Now $\gamma = \ell/N \bmod (2^n - 1)$, thus $\gamma N = \ell + (2^n - 1)\tau$ for $\tau \geq 0$, which gives $\tau \equiv -\ell/(2^n-1) \bmod N$. This yields $\delta = \ell-2\tau$, and finally $(\delta-d)/2 = h-\tau$.

The cost of the algorithm is mainly that of the four convolutions $AB \bmod (2^n \pm 1)$, $c\mu \bmod (2^n-1)$ and $\gamma N \bmod (2^n+1)$, which cost $\frac{4}{2}M(n)$ altogether. However in $c\mu \bmod (2^n - 1)$ and $\gamma N \bmod (2^n + 1)$, the operands μ and N are invariant, therefore their Fourier transform can be precomputed, which saves $\frac{1}{3}M(n)$. A further saving of $\frac{1}{6}M(n)$ is obtained by keeping d and δ in Fourier space in steps 6 and 8, performing the subtraction $d - \delta$ in Fourier space, and performing only one inverse transform for step 9. This finally gives $(2 - \frac{1}{3} - \frac{1}{6})M(n) = 1.5M(n)$. □

The $1.5M(n)$ cost of Mihailescu's algorithm is quite surprising, since it means that a modular multiplication can be performed faster than two multiplications. In other words, since a modular multiplication is basically a multiplication followed by a division, this means that the "division" can be performed for the cost of half a multiplication!

2.3.4 Special Moduli

For special moduli N faster algorithms may exist. Ideal is $N = \beta^n \pm 1$. This is precisely the kind of modulus used in the Schönhage-Strassen algorithm based on the Fast Fourier Transform (FFT). In the FFT range, a multiplication modulo $\beta^n \pm 1$ is used to perform the product of two integers of at most $n/2$ words, and a multiplication modulo $\beta^n \pm 1$ costs $M(n/2) \leq \frac{1}{2}M(n)$.

However in most applications the modulus cannot be chosen, and there is no reason for it to have a special form. We refer to §2.8 for further information about special moduli.

2.3.5 Fast Multiplication Over GF(2)[x]

The finite field GF(2) is quite important in cryptography and number theory. An alternative notation for GF(2) is F_2. Usually one considers polynomials over GF(2), i.e., elements of GF(2)[x], also called *binary polynomials*:

$$A = \sum_{i=0}^{d-1} a_i x^i,$$

where $a_i \in \mathrm{GF}(2)$ can be represented by a bit, either 0 or 1. The computer representation of a binary polynomial of degree $d-1$ is thus similar to that of an integer of d bits. It is natural to ask how fast one can multiply such binary polynomials. Since multiplication over GF(2)[x] is essentially the same as integer multiplication, but without the carries, multiplying two degree-d binary polynomials *should* be faster than multiplying two d-bit integers. In practice, this is not the case, the main reason being that modern processors do not provide a multiply instruction over GF(2)[x] at the word level. We describe here fast multiplication algorithms over GF(2)[x]: a base-case algorithm, a Toom-Cook 3-way variant, and an algorithm due to Schönhage using the Fast Fourier Transform.

The Base Case

If the multiplication over GF(2)[x] at the word level is not provided by the processor, one should implement it in software. Consider a 64-bit computer for example. One needs a routine multiplying two 64-bit words — interpreted as two binary polynomials of degree at most 63 — and returning the product as two 64-bit words, being the lower and upper part of the product. We show here how to efficiently implement such a routine using word-level operations.

For an integer i, write $\pi(i)$ the binary polynomial corresponding to i, and for a polynomial p, write $\nu(p)$ the integer corresponding to p. We have for example $\pi(0) = 0$, $\pi(1) = 1$, $\pi(2) = x$, $\pi(3) = x + 1$, $\pi(4) = x^2$, and $\nu(x^4 + 1) = 17$, ...

```
1  Algorithm GF2MulBaseCase.
2  Input: nonnegative integers 0 ≤ a, b < 2^w = β (word base).
3  Output: nonnegative integers ℓ, h with π(a)π(b) = π(h)x^w + π(ℓ).
4  Parameter: a slice size k, and n = ⌈w/k⌉.
5  Compute A_s = ν(π(a)π(s) mod x^w) for all 0 ≤ s < 2^k.
6  Write b in base 2^k: b = b_0 + b_1 2^k + ··· + b_{n-1} 2^{(n-1)k} where 0 ≤ b_j < 2^k.
7  h ← 0, ℓ ← A_{b_{n-1}}
8  for i from n − 2 downto 0 do
9      h2^w + ℓ ← (h2^w + ℓ) ≪ k
10     ℓ ← ℓ ⊕ A_{b_i}
11 u = b, m ← (2^{k-1} − 1) (2^{nk}−1)/(2^k−1) mod 2^w
12 for j from 1 to k − 1 do { repair step }
13     u ← (u ≫ 1) ∧ m
14     if bit w − j of a is set, then h ← h ⊕ u
```

(Line 9 means that $h2^w + \ell$ is shifted by k bits to the left, the low w bits are stored in ℓ, and the upper bits in h.)

Consider for example the multiplication of $a = (110010110)_2$ — which represents the polynomial $x^8 + x^7 + x^4 + x^2 + x$ — by $b = (100000101)_2$ — which represents the polynomial $x^8 + x^2 + 1$ — with $w = 9$ and $k = 3$. Step 5 will compute $A_0 = 0$, $A_1 = a = (110010110)_2$, $A_2 = xa \bmod x^9 = x^8+x^5+x^3+x^2$, $A_3 = (x+1)a \bmod x^9 = x^7+x^5+x^4+x^3+x$, $A_4 = x^6+x^4+x^3$, $A_5 = x^8+x^7+x^6+x^3+x^2+x$, $A_6 = x^8+x^6+x^5+x^4+x^2$, $A_7 = x^7+x^6+x^5+x$.

The decomposition of b in step 6 gives $b_0 = 5$, $b_1 = 0$, $b_2 = 4$. We thus have $n = 3, h = (000000000)_2, \ell = A_4 = (001011000)_2$. After step 9 for $i = 1$, we have $h = (000000001)_2, \ell = (011000000)_2$, and after step 10, ℓ is unchanged since $b_1 = 0$ and $A_0 = 0$. Now for $i = 0$, after step 9 we have $h = (000001011)_2, \ell = (000000000)_2$, and after step 10, $\ell = A_5 = (111001110)_2$.

Steps 12 to 14 form a "repair" loop. This is necessary because some upper bits from a are discarded during creation of the table A. More precisely, bit $w - j$ of a is discarded in A_s whenever $w - j + \deg(\pi(s)) \geq w$, i.e., $\deg(\pi(s)) \geq j$; in the above example the most significant bit of a is discarded in A_2 to A_7, and the second most significant bit is discarded in A_4 to A_7. Thus if bit $w-j$ of a is set, for each monomial x^t in some $\pi(s)$ with $t \geq j$, or equivalently for each set bit 2^t in some b_i, one should add the corresponding bit product. This can be performed in parallel for all concerned bits of b using a binary mask. Step 11 computes $u = (100000101)_2$, $m = (011011011)_2$. (The mask m prevents bits from a given set of k consecutive bits from carrying

Algorithm **GF2ToomCook3**.
Input: binary polynomials $A = \sum_{i=0}^{n-1} a_i x^i$, $B = \sum_{i=0}^{n-1} b_i x^i$
Output: $C = AB = c_0 + c_1 X + c_2 X^2 + c_3 X^3 + c_4 X^4$
Let $X = x^k$ where $k = \lceil n/3 \rceil$
Write $A = a_0 + a_1 X + a_2 X^2$, $B = b_0 + b_1 X + b_2 X^2$, $\deg(a_i), \deg(b_i) < k$
$c_0 \leftarrow a_0 b_0$
$c_4 \leftarrow a_2 b_2$
$v_1 \leftarrow (a_0 + a_1 + a_2)(b_0 + b_1 + b_2)$
$v_x \leftarrow (a_0 + a_1 x + a_2 x^2)(b_0 + b_1 x + b_2 x^2)$
$v_{1/x} \leftarrow (a_0 x^2 + a_1 x + a_2)(b_0 x^2 + b_1 x + b_2)$
$w_1 \leftarrow v_1 - c_0 - c_4$
$w_x \leftarrow (v_x - c_0 - c_4 x^4)/x$
$w_{1/x} \leftarrow (v_{1/x} - c_0 x^4 - c_4)/x$
$t \leftarrow (w_x + w_{1/x})/(1 + x^2)$
$c_2 \leftarrow w_1 + t$
$c_1 \leftarrow (w_{1/x} + c_2 x + t)/(1 + x^2)$
$c_3 \leftarrow t + c_1$

over to the next set.) Then for $j = 1$ in step 13 we have $u = (010000010)_2$, and since bit 8 of a was set, h becomes $h \oplus u = (010001001)_2$. For $j = 2$ we have $u = (001000001)_2$, and since bit 7 of a is also set, h becomes $h \oplus u = (011001000)_2$. This corresponds to the correct product $\pi(a)\pi(b) = x^{16} + x^{15} + x^{12} + x^8 + x^7 + x^6 + x^3 + x^2 + x$.

The complication of repair steps could be avoided by the use of double-length registers, but this would be significantly slower.

Toom-Cook 3-Way

Karatsuba's algorithm requires only 3 evaluation points, for which one usually chooses 0, 1, and ∞. Since Toom-Cook 3-way requires 5 evaluation points, one may think at first glance that it is not possible to find so many points in GF(2). The idea to overcome this difficulty is to use the transcendental variable x — and combinations of it — as evaluation points. An example is given in Algorithm **GF2ToomCook3**. With $C = c_0 + c_1 X + c_2 X^2 + c_3 X^3 + c_4 X^4$, we have $v_1 = A(1)B(1) = C(1) = c_0 + c_1 + c_2 + c_3 + c_4$, thus $w_1 = c_1 + c_2 + c_3$. Similarly, we have $v_x = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4$, thus $w_x = c_1 + c_2 x + c_3 x^2$, and $w_{1/x} = c_1 x^2 + c_2 x + c_3$. Then $t = c_1 + c_3$,

which gives c_2 as $w_1 + t$; $(w_{1/x} + c_2 x + t)/(1 + x^2)$ gives c_1, and $t + c_1$ gives c_3.

The exact divisions by $1 + x^2$ can be implemented efficiently by Hensel's division (§1.4.8), since we know the remainder is zero. More precisely, if $T(x)$ has degree $< n$, Hensel's division yields:

$$T(x) = Q(x)(1 + x^2) + R(x)x^n, \tag{2.3}$$

where $Q(x)$ of degree less than n is defined uniquely by $Q(x) = T(x)/(1 + x^2) \bmod x^n$, and $\deg R(x) < 2$. This division can be performed word-by-word, from the least to the most significant word of $T(x)$ (as in Algorithm **ConstantDivide** from §1.4.7).

Assume that the number w of bits per word is a power of 2, say $w = 32$. One uses the identity:

$$\frac{1}{1 + x^2} = (1 - x^2)(1 - x^4)(1 - x^8)(1 - x^{16}) \bmod x^{32}.$$

(Of course, the "$-$" signs in this identity can be replaced by "$+$" signs when the base field is GF(2), and we will do this from now on.) Hensel's division of a word $t(x)$ of $T(x)$ by $1 + x^2$ writes

$$t(x) = q(x)(1 + x^2) + r(x)x^w, \tag{2.4}$$

where $\deg(q) < w$ and $\deg(r) < 2$, then $(1 + x^2)(1 + x^4)(1 + x^8)(1 + x^{16}) \cdots t(x) = q(x) \bmod x^w$. The remainder $r(x)$ is simply formed by the two most significant bits of $q(x)$, since dividing (2.4) by x^w yields $q(x) \operatorname{div} x^{w-2} = r(x)$. This remainder is then added to the following word of $T(x)$, and the last remainder is known to be zero.

Using the Fast Fourier Transform

The algorithm described in this section multiplies two polynomials of degree n from GF(2)$[x]$ in time $O(n \log n \log\log n)$, using the Fast Fourier Transform (FFT).

The normal FFT requires 2^k-th roots of unity, so does not work in GF(2)$[x]$ where these roots of unity fail to exist. The solution is to replace powers of 2 by powers of 3.

Assume we want to multiply two polynomials of GF(2)$[x]/(x^N + 1)$, where $N = KM$, and K is a power of 3. Write the polynomials a, b (of degree less

than N) that are to be multiplied as follows:

$$a = \sum_{i=0}^{K-1} a_i x^{iM}, \qquad b = \sum_{i=0}^{K-1} b_i x^{iM},$$

where the a_i, b_i are polynomials of degree less than M.

Consider the a_i as elements of $R_{N'} = \mathrm{GF}(2)[x]/(x^{2N'} + x^{N'} + 1)$, for an integer $N' = KM'/3$. Let $\omega = x^{M'} \in R_{N'}$. Clearly we have $1+\omega^{K/3}+\omega^{2K/3} = 0$, and $\omega^K = 1$ in $R_{N'}$.

The Fourier transform of $(a_0, a_1, \ldots, a_{K-1})$ is $(\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{K-1})$ where

$$\hat{a}_i = \sum_{j=0}^{K-1} \omega^{ij} a_j,$$

and similarly for $(b_0, b_1, \ldots, b_{K-1})$. The pointwise product of $(\hat{a}_0, \ldots, \hat{a}_{K-1})$ by $(\hat{b}_0, \ldots, \hat{b}_{K-1})$ gives $(\hat{c}_0, \ldots, \hat{c}_{K-1})$ where

$$\hat{c}_i = \left(\sum_{j=0}^{K-1} \omega^{ij} a_j\right) \left(\sum_{k=0}^{K-1} \omega^{ik} b_k\right).$$

Now the inverse transform of $(\hat{c}_0, \ldots, \hat{c}_{K-1})$ is defined as $(c_0, \ldots, c_{K-1})$ where:

$$c_\ell = \sum_{i=0}^{K-1} \omega^{-\ell i} \hat{c}_i.$$

(Note that since $\omega^K = 1$, we have $\omega^{-\lambda} = \omega^{-\lambda \bmod K}$, or $\omega^{-1} = \omega^{K-1} = x^{M'(K-1)}$.) This gives:

$$\begin{aligned} c_\ell &= \sum_{i=0}^{K-1} \omega^{-\ell i} \left(\sum_{j=0}^{K-1} \omega^{ij} a_j\right) \left(\sum_{k=0}^{K-1} \omega^{ik} b_k\right) \\ &= \sum_{j=0}^{K-1} \sum_{k=0}^{K-1} a_j b_k \sum_{i=0}^{K-1} \omega^{i(j+k-\ell)}. \end{aligned}$$

Write $t := j + k - \ell$. If $t \neq 0 \bmod K$, then $\sum_{i=0}^{K-1} \omega^{it} = \frac{\omega^{Kt}+1}{\omega^t+1} = 0$ since $\omega^K = 1$ but $\omega^t \neq 1$. If $t = 0 \bmod K$, then $\omega^{it} = 1$; since $-K < j+k-\ell < 2K$, this happens only for $j+k-\ell \in \{0, K\}$. In that case the sum $\sum_{i=0}^{K-1} \omega^{i(j+k-\ell)}$ equals K, i.e., 1 modulo 2 (remember K is a power of 3).

We thus have for $0 \le \ell \le K-1$:

$$c_\ell = \left[\sum_{j+k=\ell} a_j b_k + \sum_{j+k=\ell+K} a_j b_k\right] \bmod (x^{2N'} + x^{N'} + 1).$$

If $N' \ge M$, $c_\ell = \sum_{j+k=\ell} a_j b_k + \sum_{j+k=\ell+K} a_j b_k$, and the polynomial $c = \sum_{\ell=0}^{K-1} c_\ell x^{\ell M}$ thus equals $ab \bmod (x^N + 1)$.

Like Schönhage-Strassen's integer multiplication, this algorithm can be used recursively. Indeed, the pointwise products mod $x^{2N'} + x^{N'} + 1$ can be performed mod $x^{3N'} + 1$, therefore a multiplication mod $x^N + 1$ reduces to about K products mod $x^{3N/K} + 1$, assuming $N' = M$. This yields the asymptotic complexity of $O(N \log N \log\log N)$.

In practice usually the algorithm is used only once: a multiplication of two degree-$N/2$ polynomials can be performed with K products of degree $\approx 2N/K$ polynomials.

Example: consider $N = 3 \cdot 5$, with $K = 3$, $M = 5$, $N' = 5$, $M' = 5$,

$$a = x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^8 + x^6 + x^5 + x^4 + x^3 + x^2 + 1,$$

$$b = x^{13} + x^{11} + x^8 + x^7 + x^6 + x^2.$$

We have $(a_0, a_1, a_2) = (x^4 + x^3 + x^2 + 1, x^3 + x + 1, x^4 + x^3 + x^2 + x + 1)$, $(b_0, b_1, b_2) = (x^2, x^3 + x^2 + x, x^3 + x)$. In the Fourier domain, we perform computations modulo $x^{10} + x^5 + 1$, and $\omega = x^5$, which gives

$$\begin{aligned}
(\hat{a}_0, \hat{a}_1, \hat{a}_2) &= (x^3 + 1, x^9 + x^7 + x, x^9 + x^7 + x^4 + x^2 + x),\\
(\hat{b}_0, \hat{b}_1, \hat{b}_2) &= (0, x^7 + x^3 + x^2 + x, x^7 + x^3 + x),\\
(\hat{c}_0, \hat{c}_1, \hat{c}_2) &= (0, x^7 + x^6 + x^3, x^6 + x^3),\\
(c_0, c_1, c_2) &= (x^7, x^6 + x^3 + x^2, x^7 + x^6 + x^3 + x^2),
\end{aligned}$$

and

$$c_0 + x^M c_1 + x^{2M} c_2 = x^{13} + x^{12} + x^{11} + x^8 + x^2 + x \bmod (x^{15} + 1).$$

We describe how the Fast Fourier Transform is performed. We want to compute efficiently

$$\hat{a}_i = \sum_{j=0}^{K-1} \omega^{ij} a_j,$$

where $K = 3^k$. Define $\mathrm{FFT}(\omega, (a_0, \ldots, a_{K-1})) = (\hat{a}_0, \ldots, \hat{a}_{K-1})$. For $0 \le \ell < K/3$ we define

$$a' = \mathrm{FFT}(\omega^3, (a_0, a_3, \ldots, a_{K/3-3})),$$
$$a'' = \mathrm{FFT}(\omega^3, (a_1, a_4, \ldots, a_{K/3-2})),$$
$$a''' = \mathrm{FFT}(\omega^3, (a_2, a_5, \ldots, a_{K/3-1})),$$

i.e.,

$$a'_\ell = \sum_{j=0}^{K/3-1} \omega^{3\ell j} a_{3j}, \quad a''_\ell = \sum_{j=0}^{K/3-1} \omega^{3\ell j} a_{3j+1}, \quad a'''_\ell = \sum_{j=0}^{K/3-1} \omega^{3\ell j} a_{3j+2}$$

thus we have for $0 \le \ell < K/3$:

$$\begin{aligned} \hat{a}_\ell &= a'_\ell + \omega^\ell a''_\ell + \omega^{2\ell} a'''_\ell, \\ \hat{a}_{\ell+K/3} &= a'_\ell + \omega^{\ell+K/3} a''_\ell + \omega^{2\ell+2K/3} a'''_\ell, \\ \hat{a}_{\ell+2K/3} &= a'_\ell + \omega^{\ell+2K/3} a''_\ell + \omega^{2\ell+4K/3} a'''_\ell. \end{aligned}$$

This group of operations is the equivalent of the "butterfly" operation $(a \leftarrow a + b, b \leftarrow a + \omega b)$ in the classical FFT.

2.4 Division and Inversion

We have seen above that modular multiplication reduces to integer division, since to compute $ab \bmod N$, the classical method consists in dividing ab by N as $ab = qN + r$, then $ab \equiv r \bmod N$. In the same vein, modular division reduces to an integer (extended) gcd. More precisely, the division $a/b \bmod N$ is usually computed as $a \cdot (1/b) \bmod N$, thus a modular inverse is followed by a modular multiplication. We therefore concentrate on modular inversion in this section.

We have seen in Chapter 1 that computing an extended gcd is expensive, both for small sizes where it usually costs several multiplications, or for large sizes where it costs $O(M(n) \log n)$. Therefore modular inversions should be avoided if possible; we explain at the end of this section how this can be done.

Table 2.4 gives Algorithm **ModularInverse**, which is just Algorithm **ExtendedGcd** (§1.6.2) with $(a, b) \to (b, N)$ and the lines computing the cofactors of N omitted. Algorithm **ModularInverse** is the naive version of

```
Algorithm ModularInverse.
Input: integers b and N, b prime to N.
Output: integer u = 1/b mod N.
(u, w) ← (1, 0), c ← N
while c ≠ 0 do
    (q, r) ←DivRem(b, c)
    (b, c) ← (c, r)
    (u, w) ← (w, u − qw)
Return u.
```

modular inversion, with complexity $O(n^2)$ if N takes n words in base β. The subquadratic $O(M(n)\log n)$ algorithm is based on the **HalfGcd** algorithm (§1.6.3).

When the modulus N has a special form, faster algorithms may exist. In particular for $N = p^k$, $O(M(n)$ algorithms exist, based on Hensel's lifting, which can be seen as the p-adic variant of Newton's method (§4.2). To compute $1/b \bmod N$, we use the iteration of (4.4):

$$x_{j+1} = x_j + x_j(1 - bx_j) \bmod p^k.$$

Assume x_j approximates $1/b$ to "p-adic precision" ℓ, i.e., $bx_j \equiv 1 + \varepsilon p^\ell$. Then $bx_{j+1} = bx_j(2 - bx_j) = (1 + \varepsilon p^\ell)(1 - \varepsilon p^\ell) = 1 - \varepsilon^2 p^{2\ell}$. Therefore x_{k+1} is an approximation of $1/b$ to double precision (in the p-adic sense).

As an example, assume one wants to compute the inverse of an odd integer b modulo 2^{32}. The initial approximation $x_0 = 1$ satisfies $x_0 = 1/b \bmod 2$, thus five iterations are enough. The first iteration is $x_1 \leftarrow x_0 + x_0(1 - bx_0) \bmod 2^2$, which simplifies to $x_1 \leftarrow 2 - b \bmod 4$ since $x_0 = 1$. Now whether $b \equiv 1 \bmod 4$ or $b \equiv 3 \bmod 4$, we have $2 - b \equiv b \bmod 4$, thus one can start directly the second iteration with $x_1 = b$:

$$\begin{aligned}
x_2 &\leftarrow b(2 - b^2) \bmod 2^4\\
x_3 &\leftarrow x_2(2 - bx_2) \bmod 2^8\\
x_4 &\leftarrow x_3(2 - bx_3) \bmod 2^{16}\\
x_5 &\leftarrow x_4(2 - bx_4) \bmod 2^{32}
\end{aligned}$$

Consider for example $b = 17$. The above algorithm yields $x_2 = 1$, $x_3 = 241$, $x_4 = 61681$ and $x_5 = 4042322161$. Of course, any computation mod p^ℓ might be computed modulo p^k for $k \geq \ell$, in particular all above computations might

be performed modulo 2^{32}. On a 32-bit computer, arithmetic on basic integer types is usually performed modulo 2^{32}, thus the reduction comes for free, and one will usually write in the C language (using `unsigned` variables):

```
x2 = b  * (2 - b * b);   x3 = x2 * (2 - b * x2);
x4 = x3 * (2 - b * x3);  x5 = x4 * (2 - b * x4);
```

Another way to perform modular division when the modulus has a special form is Hensel's division (§1.4.8). For a modulus $N = \beta^n$, given two integers A, B, we compute Q and R such that

$$A = QB + R\beta^n.$$

Therefore we have $A/B \equiv Q \bmod \beta^n$. While Montgomery's modular multiplication only needs the remainder R of Hensel's division, modular division needs the quotient Q, thus Hensel's division plays a central role in modular arithmetic modulo β^n.

2.4.1 Several Inversions at Once

A modular inversion, which reduces to an extended gcd (§1.6.2), is usually much more expensive than a multiplication. This is true not only in the FFT range, where a gcd takes time $\Theta(M(n)\log n)$, but also for smaller numbers. When several inversions are to be performed modulo the same number, the following algorithm is usually faster:

```
1  Algorithm MultipleInversion.
2  Input: 0 < x_1, ..., x_k < N
3  Output: y_1 = 1/x_1 mod N, ..., y_k = 1/x_k mod N
4  z_1 ← x_1
5  for i from 2 to k do
6      z_i ← z_{i-1} x_i mod N
7  q ← 1/z_k mod N
8  for i from k downto 2 do
9      y_i ← q z_{i-1} mod N
10     q ← q x_i mod N
11 y_1 ← q
```

Proof. We have $z_i \equiv x_1 x_2 \dots x_i \bmod N$, thus at the beginning of step i (line 8), $q \equiv (x_1 \dots x_i)^{-1} \bmod N$, which indeed gives $y_i \equiv 1/x_i \bmod N$. □

This algorithm uses only one modular inversion, and $3(k-1)$ modular multiplications. Thus it is faster than k inversions when a modular inversion is more than three times as expensive as a product. Fig. 2.5 shows a recursive variant of the algorithm, with the same number of modular multiplications: one for each internal node when going up the (product) tree, and two for each internal node when going down the (remainder) tree. This recursive variant might be performed in parallel in $O(\log k)$ operations using k processors, however the total memory usage is $\Theta(k \log k)$ residues, instead of $O(k)$ for the linear algorithm **MultipleInversion**.

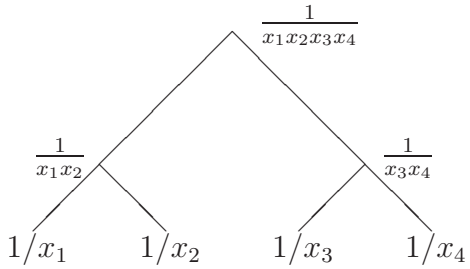


Figure 2.5: A recursive variant of Algorithm **MultipleInversion**. First go up the tree, building $x_1x_2 \bmod N$ from x_1 and x_2 in the left branch, $x_3x_4 \bmod N$ in the right branch, and $x_1x_2x_3x_4 \bmod N$ at the root of the tree. Then invert the root of the tree. Finally go down the tree, multiplying $\frac{1}{x_1x_2x_3x_4}$ by the stored value x_3x_4 to get $\frac{1}{x_1x_2}$, and so on.

A dual case is when there are several moduli but the number to invert is fixed. Say we want to compute $1/x \bmod N_1, \ldots, 1/x \bmod N_k$. We illustrate a possible algorithm in the case $k = 4$. First compute $N = N_1 \ldots N_k$ using a product tree like that in Fig. 2.5, for example first compute N_1N_2 and N_3N_4, then multiply both to get $N = (N_1N_2)(N_3N_4)$. Then compute $y = 1/x \bmod N$, and go down the tree, while reducing the residue at each node. In our example we compute $z = y \bmod (N_1N_2)$ in the left branch, then $z \bmod N_1$ yields $1/x \bmod N_1$. An important difference between this algorithm and the algorithm illustrated in Fig. 2.5 is that here, the numbers grow while going up the tree. Thus, depending on the sizes of x and the N_j, this algorithm might be of theoretical interest only.

2.5 Exponentiation

Modular exponentiation is the most time-consuming mathematical operation in several cryptographic algorithms. The well-known RSA algorithm is based on the fact that computing

$$c = a^e \bmod N \tag{2.5}$$

is relatively easy, but recovering a from c, e and N is difficult, especially when N has at least two large prime factors. The discrete logarithm problem is similar: here c, a and N are given, and one looks for e satisfying (2.5).

When the exponent e is fixed (or known to be small), an optimal sequence of squarings and multiplications might be computed in advance. This is related to the classical *addition chain* problem: What is the smallest chain of additions to reach the integer e, starting from 1? For example if $e = 15$, a possible chain is:

$$1, 1+1=2, 1+2=3, 1+3=4, 3+4=7, 7+7=14, 1+14=15.$$

The length of a chain is defined to be the number of additions needed to compute it. Thus this chain has length 6 (not 7). An addition chain readily translates to an exponentiation chain:

$$a, a\cdot a = a^2, a\cdot a^2 = a^3, a\cdot a^3 = a^4, a^3\cdot a^4 = a^7, a^7\cdot a^7 = a^{14}, a\cdot a^{14} = a^{15}.$$

A shorter chain for $e = 15$ is:

$$1, 1+1=2, 2+2=3, 2+3=5, 5+5=10, 5+10=15.$$

This chain is the shortest possible for $e = 15$, so we write $\sigma(15) = 5$, where in general $\sigma(e)$ denotes the length of the shortest addition chain for e. In the case where e is small, and an addition chain of shortest length $\sigma(e)$ is known for e, computing $a^e \bmod N$ may be performed in $\sigma(e)$ modular multiplications.

When e is large and $(a, N) = 1$, then e might be reduced modulo $\phi(N)$, where $\phi(N)$ is Euler's totient function, i.e., the number of integers in $[1, N]$ which are relatively prime to N. This is because $a^{\phi(N)} \equiv 1 \bmod N$ whenever $(a, N) = 1$ (Fermat's little theorem).

Since $\phi(N)$ is a multiplicative function, it is easy to compute $\phi(N)$ if we know the prime factorisation of N. For example,

$$\phi(1001) = \phi(7\cdot 11\cdot 13) = (7-1)(11-1)(13-1) = 720,$$

and $2009 \equiv 569 \bmod 720$, so $17^{2009} \equiv 17^{569} \bmod 1001$.

Assume now that e is smaller than $\phi(N)$. Since a lower bound on the length $\sigma(e)$ of the addition chain for e is $\lg e$, this yields a lower bound $(\lg e)M(n)$ for the modular exponentiation, where n is the size of N. When e is of size n, a modular exponentiation costs $O(nM(n))$, which is much more than the cost of operations considered in Chapter 1, with $O(M(n)\log n)$ for the more expensive ones there. The different algorithms presented in this section save only a constant factor compared to binary exponentiation.

2.5.1 Binary Exponentiation

A simple (and not far from optimal) algorithm for modular exponentiation is "binary exponentiation". Two variants exist: left-to-right and right-to-left. We give the former in Algorithm **LeftToRightBinaryExp** and leave the latter as an exercise for the reader.

```
1 Algorithm LeftToRightBinaryExp.
2 Input: a, e, N positive integers
3 Output: x = a^e mod N
4 Let (e_l e_{l-1} ... e_1 e_0) be the binary representation of e
5 x ← a
6 for i = l − 1 downto 0 do
7     x ← x^2 mod N
8     if e_i = 1 then x ← ax mod N
```

Left-to-right binary exponentiation has two advantages over right-to-left exponentiation:

- it requires only one auxiliary variable, instead of two for the right-to-left exponentiation: one to store successive values of a^{2^i}, and one to store the result;
- in the case where a is small, the multiplications ax at step 8 always involve a small operand.

If e is a random integer of n bits, step 8 will be performed on average $n/2$ times, thus the average cost of Algorithm **LeftToRightBinaryExp** is $\frac{3}{2}\ell M(n)$.

EXAMPLE: for the exponent $e = 3499211612$, which is

$$(11010000100100011011101101011100)_2$$

in binary, Algorithm **LeftToRightBinaryExp** performs 31 squarings and 15 multiplications (one for each 1-bit, except the most significant one).

2.5.2 Base 2^k Exponentiation

Compared to binary exponentiation, base 2^k exponentiation reduces the number of multiplications $ax \bmod N$ (Algorithm **LeftToRightBinaryExp**, step 8). The idea is to precompute small powers of $a \bmod N$:

```
1 Algorithm BaseKExp.
2 Input: a, e, N positive integers
3 Output: x = a^e mod N
4 Precompute a^2 then t[i] := a^i mod N for 1 ≤ i < 2^k
5 Let (e_ℓ e_{ℓ−1} ... e_1 e_0) be the base 2^k representation of e
6 x ← t[e_ℓ]
7 for i = ℓ − 1 downto 0 do
8     x ← x^{2^k} mod N
9     if e_i ≠ 0 then x ← t[e_i]x mod N
```

The precomputation cost is $(2^k-2)M(n)$, and if the digits e_i are random and uniformly distributed in $[0, 2^k-1]$, then step 9 is performed with probability $1-2^{-k}$. If e has n bits, the number of loops is about n/k. Ignoring the squares at step 8 whose total cost depends on $k\ell \approx n$ (independent of k), the total expected cost in terms of multiplications modulo N is:

$$2^k - 2 + \frac{n}{k}(1-2^{-k}).$$

For $k=1$ this formula gives $n/2$; for $k=2$ it gives $\frac{3}{8}n+2$, which is faster for $n \geq 16$; for $k=3$ it gives $\frac{7}{24}n+6$, which is faster than the $k=2$ formula for $n \geq 48$. When n is large, the optimal value of k is when $k^2 2^k \approx n/\log 2$. A disadvantage of this algorithm is its memory usage, since $\Theta(2^k)$ precomputed entries have to be stored.

Example: consider the exponent $e = 3499211612$. Algorithm **BaseKExp** performs 31 squarings independently of k, thus we count multiplications only. For $k=2$, we have $e = (3100210123231130)_4$: and Algorithm **BaseKExp** performs one multiplication to precompute a^2, and 11 multiplications for the non-zero digits of e in base 4 (except the leading one). For $k=3$, we have $e = (32044335534)_8$, and the algorithm performs 6 multiplications to precompute $a^2, a^3, \ldots, a^7$, and 9 multiplications in step 9.

This example shows two facts. First, if some digits — here 6 and 7 — do not appear in the base-2^k representation of e, we do not need to precompute the corresponding powers of a. Second, when a digit is even, say $e_i = 2$, instead of doing 3 squarings and multiplying by a^2, one could do 2 squarings, multiply by a, and perform a last squaring. This leads to the following algorithm:

```
1  Algorithm BaseKExpOdd.
2  Input: a, e, N positive integers
3  Output: x = a^e mod N
4  Precompute t[i] := a^i mod N for i odd, 1 ≤ i < 2^k,
5  Let (e_ℓ e_{ℓ-1} ... e_1 e_0) be the base 2^k representation of e
6  x ← t[e_ℓ]
7  for i = ℓ − 1 downto 0 do
8      write e_i = 2^m d with d odd (if e_i = 0 then m = d = 0)
9      x ← x^{2^{k-m}} mod N
10     if e_i ≠ 0 then x ← t[d]x mod N
11     x ← x^{2^m} mod N
```

The correctness of steps 9-11 follows from:

$$x^{2^k} a^{2^m d} = (x^{2^{k-m}} a^d)^{2^m}.$$

On our example, with $k = 3$, this algorithm performs only 4 multiplications to precompute a^2 then a^3, a^5, a^7, and 9 multiplications in step 10.

2.5.3 Sliding Window and Redundant Representation

The "sliding window" algorithm is a straightforward generalization of Algorithm **BaseKExpOdd**. Instead of cutting the exponent into fixed parts of k bits each, the idea is to divide it into windows, where two adjacent windows might be separated by a block of zero or more 0-bits. The decomposition starts from the least significant bits, for example with $e = 3499211612$, in binary:

$$\underbrace{1}_{e_8}\underbrace{101}_{e_7}00\underbrace{001}_{e_6}\underbrace{001}_{e_5}00\underbrace{011}_{e_4}\underbrace{011}_{e_3}\underbrace{101}_{e_2}\underbrace{101}_{e_1}0\underbrace{111}_{e_0}00.$$

Here there are 9 windows (indicated by $e_8, ..., e_0$ above) and we perform only 8 multiplications, an improvement of one multiplication over Algorithm **BaseKExpOdd**. On average, the sliding window algorithm leads to about

$\lceil \frac{n}{k+1} \rceil$ windows instead of $\lceil \frac{n}{k} \rceil$ with (fixed windows and) base-2^k exponentiation.

Another possible improvement may be feasible when division is possible (and cheap) in the underlying group. For example, if we encounter three consecutive ones, say 111, in the binary representation of e, we may replace some bits by -1, denoted by $\bar{1}$, as in $100\bar{1}$. We have thus replaced three multiplications by one multiplication and one division, in other words $x^7 = x^8 \cdot x^{-1}$. On our running example, this gives:

$$e = 1101000010010010010\bar{0}00\bar{1}00\bar{1}0\bar{1}00\bar{1}00,$$

which has only 10 non-zero digits, apart from the leading one, instead of 15 with bits 0 and 1 only. The redundant representation with bits 0, 1 and $\bar{1}$ is called the Booth representation. It is a special case of the Avizienis signed-digit redundant representation. Signed-digit representations exist in any base.

For simplicity we have not distinguished between the cost of multiplication and the cost of squaring (when the two operands in the multiplication are known to be equal), but this distinction is significant in some applications (e.g., elliptic curve cryptography). Note that, when the underlying group operation is denoted by addition rather than multiplication, as is usually the case for groups defined over elliptic curves, then the discussion above applies with "multiplication" replaced by "addition", "division" by "subtraction", and "squaring" by "doubling".

2.6 Chinese Remainder Theorem

In applications where integer or rational results are expected, it is often worthwhile to use a "residue number system" (as in §2.1.3) and perform all computations modulo several small primes. The final result is then recovered via the Chinese Remainder Theorem (CRT) For such applications, it is important to have fast conversion routines from integer to modular representation, and vice versa.

The integer to modular conversion problem is the following: given an integer x, and several prime moduli m_i, $1 \le i \le n$, how to efficiently compute $x_i = x \bmod m_i$, for $1 \le i \le n$? This is exactly the remainder tree problem, which is discussed in §2.4.1 (see also Ex. 1.8.27).

The converse *CRT reconstruction* problem is the following: given the x_i, how to efficiently reconstruct the unique integer x, $0 \le x < M = m_1 m_2 \cdots m_k$, such that $x = x_i \bmod m_i$, for $1 \le i \le k$? It suffices to solve this problem for $k = 2$, and use the solution recursively. Assume that $x \equiv a_1 \bmod m_1$, and $x \equiv a_2 \bmod m_2$. Write $x = \lambda m_1 + \mu m_2$. Then $\mu m_2 \equiv a_1 \bmod m_1$, and $\lambda m_2 \equiv a_2 \bmod m_2$. Assume we have computed an extended gcd of m_1 and m_2, i.e., we know integers u, v such that $um_1 + vm_2 = 1$. We deduce $\mu \equiv va_1 \bmod m_1$, and $\lambda \equiv ua_2 \bmod m_2$. Thus

$$x \leftarrow (ua_2 \bmod m_2)m_1 + (va_1 \bmod m_1)m_2.$$

This gives $x < 2m_1m_2$. If $x \ge m_1m_2$ then set $x \leftarrow x - m_1m_2$ to ensure that $0 \le x < m_1m_2$.

2.7 Exercises

Exercise 2.7.1 Show that, if a symmetric representation in $[-N/2, N/2)$ is used in Algorithm **ModularAdd** (§2.2), then the probability that we need to add or subtract N is $1/4$ if N is even, and $(1 - 1/N^2)/4$ if N is odd (assuming in both cases that a and b are uniformly distributed).

Exercise 2.7.2 Modify Algorithm **GF2MulBaseCase** (§2.3.5, The Base Case) to use a table of size $2^{\lceil k/2 \rceil}$ instead of 2^k, with only a small slowdown.

Exercise 2.7.3 Write down the complexity of Montgomery-Svoboda's algorithm (§2.3.2) for k steps. For $k = 3$, use relaxed Karatsuba multiplication [128] to save one $M(n/3)$ product.

Exercise 2.7.4 Assume you have an FFT algorithm computing products modulo $2^n + 1$. Prove that, with some preconditioning, you can perform a division of a $2n$-bit integer by an n-bit integer as fast as 1.5 multiplications of n bits by n bits.

Exercise 2.7.5 Assume you know $p(x) \bmod (x^{n_1} + 1)$ and $p(x) \bmod (x^{n_2} + 1)$, where $p(x) \in \mathrm{GF}(2)[x]$ has degree $n - 1$, and $n_1 > n_2$. Up to which value of n can you uniquely reconstruct p? Design a corresponding algorithm.

Exercise 2.7.6 Analyze the complexity of the algorithm outlined at the end of §2.4.1 to compute $1/x \bmod N_1, \ldots, 1/x \bmod N_k$, when all the N_i have size n, and x has size ℓ. For which values of n, ℓ is it faster than the naive algorithm which computes all modular inverses separately? [Assume $M(n)$ is quasi-linear, and neglect multiplication constants.]

Exercise 2.7.7 Write a **RightToLeftBinaryExp** algorithm and compare it with Algorithm **LeftToRightBinaryExp** of §2.5.1.

Exercise 2.7.8 Analyze the complexity of the CRT reconstruction algorithm outlined in §2.6 for $M = m_1 m_2 \cdots m_k$, with M having size n, and the m_i of size n/k each. [Assume $M(n) \approx n \log n$ for simplicity.]

2.8 Notes and References

Several number theoretic algorithms make heavy use of modular arithmetic, in particular integer factorization algorithms (for example: Pollard's ρ algorithm and the elliptic curve method).

Another important application of modular arithmetic in computer algebra is computing the roots of a univariate polynomial over a finite field, which requires efficient arithmetic over $\mathbb{F}_p[x]$.

We say in §2.1.3 that residue number systems can only be used when N factors into $N_1 N_2 \ldots$; this is not quite true, since Bernstein shows in [11] how to perform modular arithmetic using a residue number system.

Some examples of efficient use of the Kronecker-Schönhage trick are given by Steel [120].

The basecase multiplication algorithm from (§2.3.5, The Base Case) (with repair step) is a generalization of an algorithm published in NTL (Number Theory Library) by Shoup. The Toom-Cook 3-way algorithm from §2.3.5 was designed by the second author, after discussions with Michel Quercia, who proposed the exact division by $1 + x^2$; an implementation in NTL has been available on the web page of the second author since 2004. The Fast Fourier Transform multiplication algorithm is due to Schönhage [112].

The original description of Montgomery's REDC algorithm is [100]. It is now widely used in several applications. However only a few authors considered using a reduction factor which is not of the form β^n, among them McLaughlin [92] and Mihailescu [96]. The folding optimization of REDC described in §2.3.2 (Subquadratic Montgomery Reduction) is an LSB-extension of the algorithm described in the context of Barrett's algorithm by Hasenplaugh, Gaubatz and Gopal [69].

The description of Mihailescu's algorithm [96] in §2.3.3 is inspired by David Harvey.

Many authors have proposed FFT algorithms, or improvements of such algorithms. Some references are Aho, Hopcroft and Ullman [2]; Borodin and

Munro [19], who describe the polynomial approach; Van Loan [89] for the linear algebra approach; and Pollard [107] for the FFT over finite fields. In Bernstein [10, §23] the reader will find some historical remarks and several nice applications of the FFT.

Recently Fürer [59] has proposed an integer multiplication algorithm that is asymptotically faster than the Schönhage-Strassen algorithm. Fürer's algorithm almost achieves the conjecture best possible $\Theta(n \log n)$ running time.

Concerning special moduli, Percival considers in [106] the case $N = a \pm b$ where both a and b are highly composite; this is a generalization of the case $N = \beta^n \pm 1$.

The statement in §2.3.5 that modern processors do not provide a multiply instruction over GF(2)[x] was correct when written in 2008, but the situation may soon change, as Intel plans to introduce such an instruction `(PCMULQDQ)` on its "Westmere" chips scheduled for release in 2009.

The description in §2.3.5 of fast multiplication algorithms over GF(2)[x] is based on the paper [33], which gives an extensive study of algorithms for fast arithmetic over GF(2)[x]. For an application to factorisation of polynomials over GF(2), see [39].

The FFT algorithm described in §2.3.5 (Using the Fast Fourier Transform) is due to Schönhage [112] and is implemented in the *gf2x* package [32]. Many FFT algorithms and variants are described by Arndt in [4]: Walsh transform, Haar transform, Hartley transform, number theoretic transform (NTT).

Algorithm **MultipleInversion** is due to Montgomery [99].

Modular exponentiation algorithms are described in much detail in the *Handbook of Applied Cryptography* by Menezes, van Oorschot and Vanstone [93, Chapter 14]. A detailed description of the best theoretical algorithms, with due credit, can be found in [12].

A quadratic algorithm for CRT reconstruction is discussed in [46]; Möller gives some improvements in the case of a small number of small moduli known in advance [98]. The explicit Chinese Remainder Theorem and its applications to modular exponentiation are discussed by Bernstein and Sorenson in [14].

Chapter 3

Floating-Point Arithmetic

This Chapter discusses the basic operations — addition, subtraction, multiplication, division, square root, conversion — on arbitrary precision floating-point numbers, as Chapter 1 does for arbitrary precision integers. More advanced functions like elementary and special functions are covered in Chapter 4. This Chapter largely follows the IEEE 754 standard, and extends it in a natural way to arbitrary precision; deviations from IEEE 754 are explicitly mentioned. Topics not discussed here include: hardware implementations, fixed-precision implementations, special representations.

3.1 Representation

The classical non-redundant representation of a floating-point number x in radix $\beta > 1$ is the following (other representations are discussed in §3.8):

$$x = (-1)^s \cdot m \cdot \beta^e, \tag{3.1}$$

where $(-1)^s$, $s \in \{0, 1\}$, is the *sign*, $m \geq 0$ is the *significand*, and the integer e is the *exponent* of x. In addition, a positive integer n defines the *precision* of x, which means that the significand m contains at most n significant digits in radix β.

An important special case is $m = 0$ representing zero x. In this case the sign s and exponent e are irrelevant and may be used to encode other information (see for example §3.1.3).

For $m \neq 0$, several semantics are possible; the most common ones are:

- $\beta^{-1} \leq m < 1$, then $\beta^{e-1} \leq |x| < \beta^e$. In this case m is an integer multiple of β^{-n}. We say that the *unit in the last place* of x is β^{e-n}, and we write $\mathrm{ulp}(x) = \beta^{e-n}$. For example $x = 3.1416$ with radix $\beta = 10$ is encoded by $m = 0.31416$ and $e = 1$. This is the convention we will use in this Chapter;

- $1 \leq m < \beta$, then $\beta \leq |x| < \beta^{e+1}$, and $\mathrm{ulp}(x) = \beta^{e+1-n}$. With radix ten the number $x = 3.1416$ is encoded by $m = 3.1416$ and $e = 0$. This is the convention adopted in the IEEE 754 standard.

- we can also use an integer significand $\beta^{n-1} \leq m < \beta^n$, then $\beta^{e+n-1} \leq |x| < \beta^{e+n}$, and $\mathrm{ulp}(x) = \beta^e$. With radix ten the number $x = 3.1416$ is encoded by $m = 31416$ and $e = -4$.

Note that in the above three cases, there is only one possible representation of a non-zero floating-point number: we have a *canonical* representation. In some applications, it is useful to remove the lower bound on nonzero m, which in the three cases above gives respectively $0 < m < 1$, $0 < m < \beta$, and $0 < m < \beta^n$, with m an integer multiple of β^{e-n}, β^{e+1-n}, and 1 respectively. In this case, there is no longer a canonical representation. For example, with an integer significand and a precision of 5 digits, the number 3.1400 is encoded by $(m = 31400, e = -4)$, $(m = 03140, e = -3)$, and $(m = 00314, e = -2)$ in the three cases. However, this non-canonical representation has the drawback that the most significant non-zero digit of the significand is not known in advance. The unique encoding with a non-zero most significant digit, i.e., $(m = 31400, e = -4)$ here, is called the *normalised* — or simply *normal* — encoding.

The significand is also called *mantissa* or *fraction*. The above examples demonstrate that the different significand semantics correspond to different positions of the decimal (or radix β) point, or equivalently to different *biases* of the exponent. We assume in this Chapter that both the radix β and the significand semantics are implicit for a given implementation, thus are not physically encoded.

3.1.1 Radix Choice

Most floating-point implementations use radix $\beta = 2$ or a power of two, because this is convenient and efficient on binary computers. For a radix β

which is not a power of 2, two choices are possible:

- store the significand in base β, or more generally in base β^k for an integer $k \geq 1$. Each digit in base β^k requires $\lceil k \lg \beta \rceil$ bits. With such a choice, individual digits can be accessed easily. With $\beta = 10$ and $k = 1$, this is the "Binary Coded Decimal" or BCD encoding: each decimal digit is represented by 4 bits, with a memory loss of about 17% (since $\lg(10)/4 \approx 0.83$). A more compact choice is radix 10^3, where 3 decimal digits are stored in 10 bits, instead of in 12 bits with the BCD format. This yields a memory loss of only 0.34% (since $\lg(1000)/10 \approx 0.9966$);

- store the significand in binary. This idea is used in Intel's Binary-Integer Decimal (BID) encoding, and in one of the two decimal encodings in the revision of the IEEE 754 standard. Individual digits cannot be directly accessed, but one can use efficient binary hardware or software to perform operations on the significand.

In arbitrary precision, a drawback of the binary encoding is that, during the addition of two numbers, it is not easy to detect if the significand exceeds the maximum value $\beta^n - 1$ (when considered as an integer) and thus if rounding is required. Either β^n is precomputed, which is only realistic if all computations involve the same precision n, or it is computed on the fly, which might result in $O(M(n) \log n)$ complexity (see Chapter 1).

3.1.2 Exponent Range

In principle, one might consider an unbounded exponent. In other words, the exponent e might be encoded by an arbitrary precision integer (Chapter 1). This has the great advantage that no underflow or overflow would occur (see below). However, in most applications, an exponent encoded in 32 bits is more than enough: this enables one to represent values up to about $10^{646456993}$. A result exceeding this value most probably corresponds to an error in the algorithm or the implementation. In addition, using arbitrary precision integers for the exponent induces an extra overhead that slows down the implementation in the average case, and it requires more memory to store each number.

Thus, in practice the exponent usually has a limited range $e_{\min} \leq e \leq e_{\max}$. We say that a floating-point number is *representable* if it can represented in the form $(-1)^s \cdot m \cdot \beta^e$ with $e_{\min} \leq e \leq e_{\max}$. The set of representable

numbers clearly depends on the significand semantics. For the convention we use here, i.e., $\beta^{-1} \leq m < 1$, the smallest positive floating-point number is $\beta^{e_{\min}-1}$, and the largest is $\beta^{e_{\max}}(1-\beta^{-n})$.

Other conventions for the significand yield different exponent ranges. For example the IEEE 754 double precision format — called `binary64` in the IEEE 754 revision — has $e_{\min} = -1022$, $e_{\max} = 1023$ for a significand in $[1, 2)$; this corresponds to $e_{\min} = -1021$, $e_{\max} = 1024$ for a significand in $[1/2, 1)$, and $e_{\min} = -1074$, $e_{\max} = 971$ for an integer significand in $[2^{52}, 2^{53})$.

3.1.3 Special Values

With a bounded exponent range, if one wants a complete arithmetic, one needs some special values to represent very large and very small values. Very small values are naturally flushed to zero, which is a special number in the sense that its significand is $m = 0$, which is not normalised. For very large values, it is natural to introduce two special values $-\infty$ and $+\infty$, which encode large non-representable values. Since we have two infinities, it is natural to have two zeros -0 and $+0$, for example $1/(-\infty) = -0$ and $1/(+\infty) = +0$. This is the IEEE 754 choice. Another possibility would be to have only one infinity ∞ and one zero 0, forgetting the sign in both cases.

An additional special value is *Not a Number* (NaN), which either represents an uninitialised value, or is the result of an *invalid* operation like $\sqrt{-1}$ or $(+\infty)-(+\infty)$. Some implementations distinguish between different kinds of NaN, in particular IEEE 754 defines *signalling* and *quiet* NaNs.

3.1.4 Subnormal Numbers

Subnormal numbers are required by the IEEE 754 standard, to allow what is called *gradual underflow* between the smallest (in absolute value) non-zero normalised numbers and zero. We first explain what subnormal numbers are; then we will see why they are not necessary in arbitrary precision.

Assume we have an integer significand in $[\beta^{n-1}, \beta^n)$ where n is the precision, and an exponent in $[e_{\min}, e_{\max}]$. Write $\eta = \beta^{e_{\min}}$. The two smallest positive normalised numbers are $x = \beta^{n-1}\eta$ and $y = (\beta^{n-1}+1)\eta$. The difference $y - x$ equals η, which is tiny compared to the difference between 0 and x, which is $\beta^{n-1}\eta$. In particular, $y - x$ cannot be represented exactly as a normalised number, and will be rounded to zero in rounded to nearest mode. This has the unfortunate consequence that instructions like:

```
if (y <> x) then
   z = 1.0 / (y - x);
```

will produce a "division by zero" error within `1.0 / (y - x)`.

Subnormal numbers solve this problem. The idea is to relax the condition $\beta^{n-1} \leq m$ for the exponent $e_{\min}$. In other words, we include all numbers of the form $m \cdot \beta^{e_{\min}}$ for $1 \leq m < \beta^{n-1}$ in the set of valid floating-point numbers. One could also permit $m = 0$, and then zero would be a subnormal number, but we continue to regard zero as a special case.

Subnormal numbers are all integer multiples of η, with a multiplier $1 \leq m < \beta^{n-1}$. The difference between $x = \beta^{n-1}\eta$ and $y = (\beta^{n-1} + 1)\eta$ is now representable, since it equals η, the smallest positive subnormal number. More generally, all floating-point numbers are multiples of η, likewise for their sum or difference (in other words. operations in the subnormal domain correspond to fixed-point arithmetic). If the sum or difference is non-zero, it has magnitude at least η, thus cannot be rounded to zero. Thus the "division by zero" problem mentioned above does not occur with subnormal numbers.

In the IEEE 754 double precision format — called `binary64` in the IEEE 754 revision — the smallest positive normal number is 2^{-1022}, and the smallest positive subnormal number is 2^{-1074}. In arbitrary precision, subnormal numbers seldom occur, since usually the exponent range is huge compared to the expected exponents in a given application. Thus the only reason for implementing subnormal numbers in arbitrary precision is to provide an extension of IEEE 754 arithmetic. Of course, if the exponent range is unbounded, then there is absolutely no need for subnormal numbers, because any nonzero floating-point number can be normalised.

3.1.5 Encoding

The *encoding* of a floating-point number $x = (-1)^s \cdot m \cdot \beta^e$ is the way the values s, m and e are stored in the computer. Remember that β is implicit, i.e., is considered fixed for a given implementation; as a consequence, we do not consider here *mixed radix* operations involving numbers with different radices β and β'.

We have already seen that there are several ways to encode the significand m when β is not a power of two: in base-β^k or in binary. For normal numbers in radix 2, i.e., $2^{n-1} \leq m < 2^n$, the leading bit of the significand is necessarily 1, thus one might choose not the encode it in memory, to gain an extra bit of

precision. This is called the *implicit leading bit*, and it is the choice made in the IEEE 754 formats. For example the double precision format has a sign bit, an exponent field of 11 bits, and a significand of 53 bits, with only 52 bits stored, which gives a total of 64 stored bits:

sign (1 bit)	exponent (11 bits)	significand (52 bits, plus implicit leading bit)

A nice consequence of this particular encoding is the following. Let x be a double precision number, neither subnormal, $\pm\infty$, NaN, nor the largest normal number. Consider the 64-bit encoding of x as a 64-bit integer, with the sign bit in the most significant bit, the exponent bits in the next significant bits, and the explicit part of the significand in the low significant bits. Adding 1 to this 64-bit integer yields the next double precision number to x, away from zero. Indeed, if the significand m is smaller than $2^{53}-1$, m becomes $m+1$ which is smaller than 2^{53}. If $m = 2^{53}-1$, then the lowest 52 bits are all set, and a carry occurs between the significand field and the exponent field. Since the significand field becomes zero, the new significand is 2^{52}, taking into account the implicit leading bit. This corresponds to a change from $(2^{53}-1)\cdot 2^e$ to $2^{52}\cdot 2^{e+1}$, which is exactly the next number away from zero. Thanks to this consequence of the encoding, an integer comparison of two words (ignoring the actual type of the operands) should give the same result as a floating-point comparison, so it is possible to sort normal floating-point numbers as if they were integers of the same length (64-bit for double precision).

In arbitrary precision, saving one bit is not as crucial as in fixed (small) precision, where one is constrained by the word size (usually 32 or 64 bits). Thus, in arbitrary precision, it is easier and preferable to encode the whole significand. Also, note that having an "implicit bit" is not possible in radix $\beta > 2$, since for a normal number the most significant digit might take several values, from 1 to $\beta - 1$.

When the significand occupies several words, it can be stored in a linked list, or in an array (with a separate size field). Lists are easier to extend, but accessing arrays is usually more efficient because fewer memory references are required in the inner loops.

The sign s is most easily encoded as a separate bit field, with a non-negative significand. Other possibilities are to have a signed significand, using either 1's complement or 2's complement, but in the latter case a special encoding is required for zero, if it is desired to distinguish +0 from

-0. Finally, the exponent might be encoded as a signed word (for example, type `long` in the C language).

3.1.6 Precision: Local, Global, Operation, Operand

The different operands of a given operation might have different precisions, and the result of that operation might be desired with yet another precision. There are several ways to address this issue.

- The precision, say n is attached to a given operation. In this case, operands with a smaller precision are automatically converted to precision n. Operands with a larger precision might either be left unchanged, or rounded to precision n. In the former case, the code implementing the operation must be able to handle operands with different precisions. In the latter case, the rounding mode to shorten the operands must be specified. Note that this rounding mode might differ from that of the operation itself, and that operand rounding might yield large errors. Consider for example $a = 1.345$ and $b = 1.234567$ with a precision of 4 digits. If b is taken as exact, the exact value of $a - b$ equals 0.110433, which when rounded to nearest becomes 0.1104. If b is first rounded to nearest to 4 digits, we get $b' = 1.235$, and $a - b' = 0.1100$ is rounded to itself.

- The precision n is attached to each variable. Here again two cases may occur. If the operation destination is part of the operation inputs, as in `sub (c, a, b)`, which means $c \leftarrow \circ(a - b)$, then the precision of the result operand c is known, thus the rounding precision is known in advance. Alternatively, if no precision is given for the result, one might choose the maximal (or minimal) precision from the input operands, or use a global variable, or request an extra precision parameter for the operation, as in `c = sub (a, b, n)`.

Of course, all these different semantics are non-equivalent, and may yield different results. In the following, we consider the case where each variable, including the destination variable, has its own precision, and no pre-rounding or post-rounding occurs. In other words, the operands are considered exact to their full precision.

Rounding is considered in detail in §3.1.9. Here we define what we mean by the *correct rounding* of a function.

Definition 3.1.1 *Let $a, b, \ldots$ be floating-point numbers, f be a mathematical function, $n \geq 1$ be an integer, and $\circ$ a rounding mode. We say that c is the* correct rounding *of $f(a, b, \ldots)$, and we write $c = \circ(f(a, b, \ldots))$, if c is the floating-point number closest from $f(a, b, \ldots)$ according to the given rounding mode. (In case several numbers are at the same distance of $f(a, b, \ldots)$, the rounding mode must define in a deterministic way which one is "the closest".)*

3.1.7 Link to Integers

Most floating-point operations reduce to arithmetic on the significands, which can be considered as integers as seen in the beginning of this section. Therefore efficient arbitrary precision floating-point arithmetic requires efficient underlying integer arithmetic (see Chapter 1).

Conversely, floating-point numbers might be useful for the implementation of arbitrary precision integer arithmetic. For example, one might use hardware floating-point numbers to represent an arbitrary precision integer. Indeed, since a double precision floating-point has 53 bits of precision, it can represent an integer up to $2^{53} - 1$, and an integer A can be represented as: $A = a_{n-1}\beta^{n-1} + \cdots + a_i\beta^i + \cdots + a_1\beta + a_0$, where $\beta = 2^{53}$, and the a_i are stored in double precision numbers. Such an encoding was popular when most processors were 32-bit, and some had relatively slow integer operations in hardware. Now that most computers are 64-bit, this encoding is obsolete.

Floating-point *expansions* are a variant of the above. Instead of storing a_i and having β^i implicit, the idea is to directly store $a_i\beta^i$. Of course, this only works for relatively small i, i.e., whenever $a_i\beta^i$ does not exceed the format range. For example, for IEEE 754 double precision and $\beta = 2^{53}$, the maximal precision is 1024 bits. (Alternatively, one might represent an integer as a multiple of the smallest positive number 2^{-1074}, with a corresponding maximal precision of 2098 bits.)

Hardware floating-point numbers might also be used to implement the Fast Fourier Transform (FFT), using complex numbers with floating-point real and imaginary part (see §3.3.1).

3.1.8 Ziv's Algorithm and Error Analysis

A *rounding boundary* is a point at which the rounding function $\circ(x)$ is discontinuous.

In fixed precision, for basic arithmetic operations, it is sometimes possible to design one-pass algorithms that directly compute a correct rounding. However, in arbitrary precision, or for elementary or special functions, the classical method is to use Ziv's algorithm:

1. we are given an input x, a target precision n, and a rounding mode;

2. compute an approximation y with precision $m > n$, and a corresponding error bound ε such that $|y - f(x)| \leq \varepsilon$;

3. if $[y-\varepsilon, y+\varepsilon]$ contains a rounding boundary, increase m and go to Step 2;

4. output the rounding of y, according to the given mode.

The error bound ε at Step 2 might be computed either *a priori*, i.e., from x and n only, or *dynamically*, i.e., from the different intermediate values computed by the algorithm. A dynamic bound will usually be tighter, but will require extra computations (however, those computations might be done in low precision).

Depending on the mathematical function to be implemented, one might prefer an absolute or a relative error analysis. When computing a relative error bound, at least two techniques are available: one might express the errors in terms of units in the last place (ulps), or one might express them in terms of true relative error. It is of course possible in a given analysis to mix both kinds of errors, but in general one loses a constant factor — the radix β — when converting from one kind of relative error to the other kind.

Another important distinction is *forward* vs *backward* error analysis. Assume we want to compute $y = f(x)$. Because the input is rounded, and/or because of rounding errors during the computation, we might actually compute $y' \approx f(x')$. Forward error analysis will bound $|y' - y|$ if we have a bound on $|x' - x|$ and on the rounding errors that occur during the computation.

Backward error analysis works in the other direction. If the computed value is y', then backward error analysis will give us a number δ such that, for *some* x' in the ball $|x' - x| \leq \delta$, we have $y' = f(x')$. This means that the error is *no worse* than might have been caused by an error of δ in the input value. Note that, if the problem is ill-conditioned, δ might be small even if $|y' - y|$ is large.

In our error analyses, we assume that no overflow or underflow occurs, or equivalently that the exponent range is unbounded, unless the contrary is explicitly stated.

3.1.9 Rounding

There are several possible definitions of rounding. For example *probabilistic rounding* — also called *stochastic rounding* — chooses at random a rounding towards $+\infty$ or $-\infty$ for each operation. The IEEE 754 standard defines four rounding modes: towards zero, $+\infty$, $-\infty$ and to nearest (with ties broken to even). Another useful mode is "rounding away", which rounds in the opposite direction from zero: a positive number is rounded towards $+\infty$, and a negative number towards $-\infty$. If the sign of the result is known, all IEEE 754 rounding modes might be converted to either rounding to nearest, rounding towards zero, or rounding away.

Theorem 3.1.1 *Consider a binary floating-point system with radix β and precision n. Let u be the rounding to nearest of some real x, then the following inequalities hold:*

$$\begin{aligned} |u - x| &\leq \frac{1}{2}\,\mathrm{ulp}(u) \\ |u - x| &\leq \frac{1}{2}\beta^{1-n}|u| \\ |u - x| &\leq \frac{1}{2}\beta^{1-n}|x|. \end{aligned}$$

Proof. For $x = 0$, necessarily $u = 0$, and the statement holds. Without loss of generality, we can assume u and x positive. The first inequality is the definition of rounding to nearest, and the second one follows from $\mathrm{ulp}(u) \leq \beta^{1-n}u$. (In the case $\beta = 2$, it gives $|u - x| \leq 2^{-n}|u|$.) For the last inequality, we distinguish two cases: if $u \leq x$, it follows from the second inequality. If $x < u$, then if x and u have the same exponent, i.e., $\beta^{e-1} \leq x < u < \beta^e$, then $\frac{1}{2}\,\mathrm{ulp}(u) = \frac{1}{2}\beta^{e-n} \leq \frac{1}{2}\beta^{1-n}x$. The only remaining case is $\beta^{e-1} \leq x < u = \beta^e$. Since the floating-point number preceding β^e is $\beta^e(1 - \beta^{-n})$, and x was rounded to nearest, we have $|u - x| \leq \frac{1}{2}\beta^{e-n}$ here too. □

In order to round according to a given rounding mode, one proceeds as follows:

1. first round as if the exponent range was unbounded, with the given rounding mode;

2. if the rounded result is within the exponent range, return this value;

3. otherwise raise the "underflow" or "overflow" exception, and return ± 0 or $\pm\infty$ accordingly.

For example, assume radix 10 with precision 4, $e_{\max} = 3$, with $x = 0.9234 \cdot 10^3$, $y = 0.7656 \cdot 10^2$. The exact sum $x + y$ equals $0.99996 \cdot 10^3$. With rounding towards zero, we obtain $0.9999 \cdot 10^3$, which is representable, so there is no overflow. With rounding to nearest, $x + y$ rounds to $0.1000 \cdot 10^4$ with an unbounded exponent range, which exceeds $e_{\max} = 3$, thus we get $+\infty$ as result, with an overflow. In this model, the overflow depends not only on the operands, but also on the rounding mode. This is consistent with IEEE 754, which requires that a number larger or equal to $(1 - 2^{-n})2^{e_{\max}}$ rounds to $+\infty$.

The "round to nearest" mode from IEEE 754 rounds the result of an operation to the nearest representable number. In case the result of an operation is exactly in the middle of two consecutive numbers, the one with its least significant bit zero is chosen (remember IEEE 754 only considers radix 2). For example 1.1011_2 is rounded with a precision of 4 bits to 1.110_2, as is 1.1101_2. However this rule does not readily extend to an arbitrary radix. Consider for example radix $\rho = 3$, a precision of 4 digits, and the number $1212.111\ldots_3$. Both $1212._3$ and $1220._3$ end in an even digit. The natural extension is to require the whole significand to be even, when interpreted as an integer in $[\rho^{n-1}, \beta^n - 1]$. In this setting, $(1212.111\ldots)_3$ rounds to $(1212)_3 = 50_{10}$. (Note that ρ^n is an odd number here.)

Assume we want to correctly round to n bits a real number whose binary expansion is $2^e \cdot 0.1b_1 \ldots b_n b_{n+1} \ldots$ It is enough to know the values of $r = b_{n+1}$ — called the *round bit* — and that of the *sticky bit* s, which is 0 when $b_{n+2}b_{n+3}\ldots$ is identically zero, and 1 otherwise. Table 3.1 shows how to correctly round given r, s, and the given rounding mode; rounding to $\pm\infty$ being converted to rounding to zero or away, according to the sign of the number. The entry "b_n" is for round to nearest in the case of a tie: if $b_n = 0$ it will be unchanged, but if $b_n = 1$ we add 1 (thus changing b_n to 0).

In general, we do not have an infinite expansion, but a finite approximation y of an unknown real value x. For example, y might be the result of an arithmetic operation such as division, or an approximation to the value of a

r	s	to zero	to nearest	away
0	0	0	0	0
0	1	0	0	1
1	0	0	b_n	1
1	1	0	1	1

Table 3.1: Rounding rules according to the round bit r and the sticky bit s: a "0" entry means truncate (round to zero), a "1" means round away from zero (add 1 to the truncated significand).

transcendental function such as exp. The following problem arises: given the approximation y, and a bound on the error $|y-x|$, is it possible to determine the correct rounding of x? Algorithm **RoundingPossible** returns *true* if and only if it is possible.

```
1 Algorithm RoundingPossible.
2 Input: a floating-point number y = 0.1y_2 ... y_m, a precision n ≤ m,
3        an error bound ε = 2^{-k}, a rounding mode ∘
4 Output: true when ∘_n(x) can be determined for |y - x| < ε
5 If k ≤ n + 1 then return false
6 If ∘ is to nearest then ℓ ← n + 1 else ℓ ← n
7 If ∘ is to nearest and y_ℓ = y_{ℓ+1} then return true
8 If y_{ℓ+1} = y_{ℓ+2} = ... = y_k then return false
9 Return true.
```

Proof. Since rounding is monotonic, it is possible to determine $\circ(x)$ exactly when $\circ(y-2^{-k}) = \circ(y+2^{-k})$, or in other words when the interval $[y-2^{-k}, y+2^{-k}]$ contains no rounding boundary. The rounding boundaries for rounding to nearest in precision n are those for directed rounding in precision $n+1$.

If $k \leq n+1$, then the interval $[-2^{-k}, 2^{-k}]$ has width at least 2^{-n}, thus contains one rounding boundary: it is not possible to round correctly. In case of rounding to nearest, if the round bit and the following bit are equal — thus 00 or 11 — and the error is after the round bit ($k > n+1$), it is possible to round correctly. Otherwise it is only possible when $y_{\ell+1}, y_{\ell+2}, \ldots, y_k$ are not all identical. □

The Double Rounding Problem

When a given real value x is first rounded to precision m, then to precision $n < m$, we say that a "double rounding" occurs. The "double rounding problem" happens when this value differs from the direct rounding of x to the smaller precision n, assuming the same rounding mode is used in all cases:

$$\circ_n(\circ_m(x)) \neq \circ_n(x).$$

The double rounding problem does not occur for the directed rounding modes. For those rounding modes, the rounding boundaries at the larger precision m refine those at the smaller precision n, thus all real values x that round to the same value y at precision m also round to the same value at precision n, namely $\circ_n(y)$.

Consider the decimal value $x = 3.14251$. Rounding to nearest to 5 digits, one gets $y = 3.1425$; rounding y to nearest-even to 4 digits, one gets 3.142, whereas the direct rounding of x would give 3.143.

With rounding to nearest mode, the double rounding problem only occurs when the second rounding involves the even-rule, i.e., the value $y = \circ_m(x)$ is a rounding boundary at precision n. Indeed, otherwise y is at distance at least one ulp (in precision m) of a rounding boundary at precision n, and since $|y - x|$ is bounded by half an ulp (in precision m), all possible values for x round to the same value in precision n.

Note that the double rounding problem does not occur with all ways of breaking ties for rounding to nearest (Ex. 3.7.2).

3.1.10 Strategies

To determine correct rounding of $f(x)$ with n bits of precision, the best strategy is usually to first compute an approximation y of $f(x)$ with a working precision of $m = n + h$ bits, with h relatively small. Several strategies are possible in Ziv's algorithm (§3.1.8) when this first approximation y is not accurate enough, or too close to a rounding boundary.

- Compute the exact value of $f(x)$, and round it to the target precision n. This is possible for a basic operation, for example $f(x) = x^2$, or more generally $f(x, y) = x+y$ or $x \times y$. Some elementary functions may yield an exact representable output too, for example $\sqrt{2.25} = 1.5$. An "exact result" test after the first approximation avoids possibly unnecessary further computations.

- Repeat the computation with a larger working precision $m' = n + h'$. Assuming that the digits of $f(x)$ behave "randomly" and that $|f'(x)/f(x)|$ is not too large, using $h' \approx \lg n$ is enough to guarantee that rounding is possible with probability $1 - O(\frac{1}{n})$. If rounding is still not possible, because the h' last digits of the approximation encode 0 or $2^{h'} - 1$, one can increase the working precision and try again. A check for exact results guarantees that this process will eventually terminate, provided the algorithm used has the property that it gives the exact result if this result is representable and the working precision is high enough. For example, the square root algorithm should return the exact result if it is representable (see Algorithm **FPSqrt** in § 3.5, and also exercise 3.7.3).

3.2 Addition, Subtraction, Comparison

Addition and subtraction of floating-point numbers operate from the most significant digits, whereas the integer addition and subtraction start from the least significant digits. Thus completely different algorithms are involved. In addition, in the floating-point case, part or all of the inputs might have no impact on the output, except in the rounding phase.

In summary, floating-point addition and subtraction are more difficult to implement than integer addition/subtraction for two reasons:

- scaling due to the exponents requires shifting the significands before adding or subtracting them. In principle one could perform all operations using only integer operations, but this would require huge integers, for example when adding 1 and 2^{-1000}.

- as the carries are propagated from least to most significant digits, one may have to look at arbitrarily low input digits to guarantee correct rounding.

In this section, we distinguish between "addition", where both operands to be added have the same sign, and "subtraction", where the operands to be added have different signs (we assume a sign-magnitude representation). The case of one or both operands zero is treated separately; in the description below we assume that all operands are nonzero.

3.2.1 Floating-Point Addition

Algorithm **FPadd** adds two binary floating-point numbers b and c of the same sign. More precisely, it computes the correct rounding of $b + c$, with respect to the given rounding mode $\circ$. For the sake of simplicity, we assume b and c are positive, $b \geq c > 0$. It will also be convenient to scale b and c so that $2^{n-1} \leq b < 2^n$ and $2^{m-1} \leq c < 2^m$, where n is the desired precision of the output, and $m \leq n$. Of course, if the inputs$b$ and c to Algorithm **FPadd** are scaled by 2^k, then the output must be scaled by 2^{-k}. We assume that the rounding mode is to nearest, towards zero, or away from zero (rounding to $\pm\infty$ reduces to rounding towards zero or away from zero, depending on the sign of the operands).

```
1  Algorithm FPadd.
2  Input: b ≥ c > 0 two binary floating−point numbers,
3         a precision n such that 2^{n−1} ≤ b < 2^n,
4         and a rounding mode ∘.
5  Output: a floating-point number a of precision n and scale e
6         such that a · 2^e = ∘(b + c)
7  Split b into b_h + b_ℓ where b_h contains the n most significant
8        bits of b.
9  Split c into c_h + c_ℓ where c_h contains the most
10       significant bits of c, and ulp(c_h) = ulp(b_h).
11 a_h ← b_h + c_h,   e ← 0
12 (c, r, s) ← b_ℓ + c_ℓ
13 (a, t) ← a_h + c + round(∘, r, s)
14 e ← 0
15 if a ≥ 2^n then
16    a ← round2(∘, a mod 2, t), \quad e ← e + 1
17    if a = 2^n then (a, e) ← (a/2, e + 1)
18 Return (a, e).
```

The values of round$(\circ, r, s)$ and round2$(\circ, a \bmod 2, t)$ are given in Table 3.2. At step 12, the notation $(c, r, s) \leftarrow b_l + c_l$ means that c is the carry bit of $b_l + c_l$, r the round bit, and s the sticky bit: c, r, and s are in $\{0, 1\}$. For rounding to nearest, t is a ternary value, which is respectively positive, zero, or negative when a is smaller than, equal to, or larger than the exact sum $b + c$.

$\circ$	r	s	round$(\circ, r, s)$	$t := \text{sign}(b+c-a)$
zero	any	any	0	
away	any	any	0 if $r = s = 0$, 1 otherwise	
nearest	0	any	0	$+s$
nearest	1	0	0/1 (even rounding)	$+1/-1$
nearest	1	$\neq 0$	1	-1

$\circ$	$a \bmod 2$	t	round2$(\circ, a \bmod 2, t)$
any	0	any	$a/2$
zero	1		$(a-1)/2$
away	1		$(a+1)/2$
nearest	1	0	$(a-1)/2$ if even, $(a+1)/2$ otherwise
nearest	1	± 1	$(a+t)/2$

Table 3.2: Rounding rules for addition.

Theorem 3.2.1 *Algorithm **FPadd** is correct.*

Proof. Without loss of generality, we can assume that $2^{n-1} \leq b < 2^n$ and $2^{m-1} \leq c < 2^m$, with $m \leq n$. With this assumption, b_h and c_h are the integer parts of b and c, b_l and c_l their fractional parts. Since $b \geq c$, we have $c_h \leq b_h$ and $2^{n-1} \leq b_h \leq 2^n - 1$, thus $2^{n-1} \leq a_h \leq 2^{n+1} - 2$, and at step 13, $2^{n-1} \leq a \leq 2^{n+1}$. If $a < 2^n$, a is the correct rounding of $b+c$. Otherwise, we face the "double rounding" problem: rounding a down to n bits will give the correct result, except when a is odd and rounding is to nearest. In that case, we need to know if the first rounding was exact, and if not in which direction it was rounded; this information is represented by the ternary value t. After the second rounding, we have $2^{n-1} \leq a \leq 2^n$. □

Note that the exponent e_a of the result lies between e_b (the exponent of b, here we considered the case $e_b = n$) and $e_b + 2$. Thus no underflow can occur in an addition. The case $e_a = e_b + 2$ can occur only when the destination precision is less than that of the operands.

3.2.2 Floating-Point Subtraction

Floating-point subtraction is very similar to addition; with the difference that *cancellation* can occur. Consider for example the subtraction $6.77823 -$

5.98771. The most significant digit of both operands disappeared in the result 0.79052. This cancellation can be dramatic, as in $6.7782357934 - 6.7782298731 = 0.0000059203$, where six digits were cancelled.

Two approaches are possible, assuming n result digits are wanted, and the exponent difference between the inputs is d.

- Subtract from the most n significant digits of the larger operand (in absolute value) the $n - d$ significant digits of the smaller operand. If the result has $n - e$ digits with $e > 0$, restart with $n + e$ digits from the larger operand and $(n + e) - d$ from the smaller one.

- Alternatively, predict the number e of cancelled digits in the subtraction, and directly subtract the $(n+e)-d$ most significant digits of the smaller operand from the $n+e$ most significant digits of the larger one.

Note that in the first case, we might have $e = n$, i.e., all most significant digits cancel, thus the process might need to be repeated several times.

The first step in the second phase is usually called *leading zero detection*. Note that the number e of cancelled digits might depend on the rounding mode. For example $6.778 - 5.7781$ with a 3-digit result yields 0.999 with rounding toward zero, and 1.00 with rounding to nearest. Therefore in a real implementation, the exact definition of e has to be made more precise.

Finally, in practice one will consider $n+g$ and $(n+g)-d$ digits instead of n and $n - d$, where the g "guard digits" will prove useful (i) either to decide of the final rounding, (ii) and/or to avoid another loop in case $e \leq g$.

Sterbenz's Theorem

Sterbenz's Theorem is an important result concerning floating-point subtraction (of operands of same sign). It states that the rounding error is zero in some important cases. More precisely:

Theorem 3.2.2 (Sterbenz) If x and y are two floating-point numbers of same precision n, such that y lies in the interval $[x/2, 2x]$, then $y-x$ is exactly representable in precision n.

Proof. The case $x = y = 0$ is trivial, so assume that $x \neq 0$. Since y lies in $[x/2, 2x]$, x and y must have the same sign, We assume without loss of generality that x and y are positive.

Assume $x \le y \le 2x$ (the same reasoning applies for $x/2 \le y < x$, i.e., $y \le x \le 2y$, which is symmetric in x, y). Then since $x \le y$, we have $\mathrm{ulp}(x) \le \mathrm{ulp}(y)$, thus y is an integer multiple of $\mathrm{ulp}(x)$. It follows that $y-x$ is an integer multiple of $\mathrm{ulp}(x)$ too, and since $0 \le y-x \le x$, $y-x$ is necessarily representable with the precision of x. □

It is important to note that Sterbenz's Theorem applies for any radix β; the constant 2 in $[x/2, 2x]$ has nothing to do with the radix.

3.3 Multiplication

Multiplication of floating-point numbers is called a *short product*. This reflects the fact that in some cases, the low part of the full product of the significands has no impact — except maybe for the rounding — on the final result. Consider the multiplication xy, where $x = \ell \cdot \beta^e$, and $y = m \cdot \beta^f$. Then $\circ(x \cdot y) = \circ(\ell \cdot m)\beta^{e+f}$, thus it suffices to consider the case where $x = \ell$ and $y = m$ are integers, and the product is rounded at some weight β^g for $g \ge 0$. Either the integer product $\ell \cdot m$ is computed exactly, using one of the algorithms from Chapter 1, and then rounded; or the upper part is computed directly using a "short product algorithm", with correct rounding. The different cases that can occur are depicted in Fig. 3.1.

An interesting question is: how many consecutive identical bits can occur after the round bit? Without loss of generality, we can rephrase this question as follows. Given two odd integers of at most n bits, what is the longest run of identical bits in their product? (In the case of an even significand, one might write it $m = \ell \cdot 2^e$ with ℓ odd.) There is no a priori bound except the trivial one of $2n-2$ for the number of zeros, and $2n-1$ for the number of ones. Consider with a precision 5 bits for example, $27 \times 19 = (1000000001)_2$. More generally such a case corresponds to a factorisation of $2^{2n-1}+1$ into two integers of n bits, for example $258513 \times 132913 = 2^{35}+1$. For consecutive ones, the value $2n$ is not possible since $2^{2n}-1$ cannot factor into two integers of at most n bits. Therefore the maximal runs have $2n-1$ ones, for example $217 \times 151 = (111111111111111)_2$ for $n = 8$. A larger example is $849583 \times 647089 = 2^{39}-1$.

The exact product of two floating-point numbers $m \cdot \beta^e$ and $m' \cdot \beta^{e'}$ is $(mm') \cdot \beta^{e+e'}$. Therefore, if no underflow or overflow occurs, the problem reduces to the multiplication of the significands m and m'. See Algorithm **FPmultiply**.

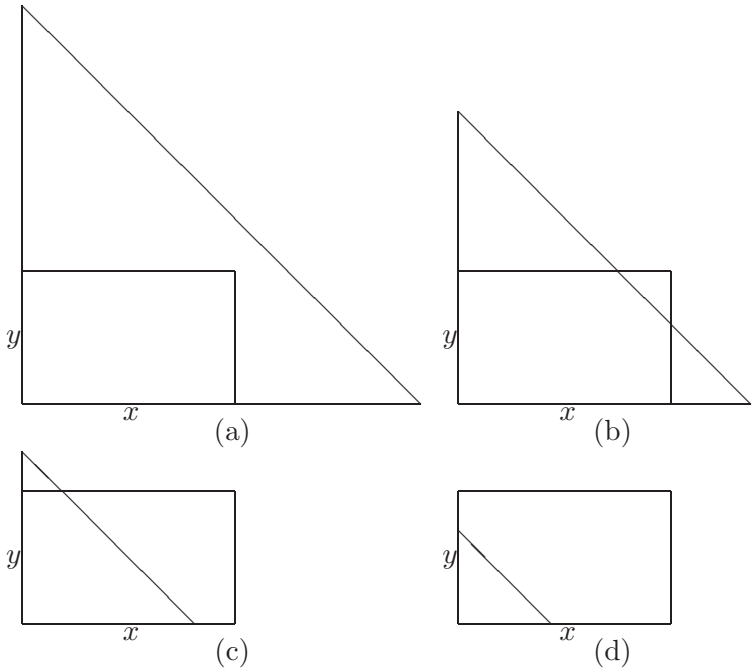


Figure 3.1: Different multiplication scenarios, according to the input and output precisions. The rectangle corresponds to the full product of the inputs x and y (most significand digits bottom left), the triangle to the wanted short product. Case (a), no rounding is necessary, the product being exact; case (b): the full product needs to be rounded, but the inputs should not be; case (c): the input with the larger precision might be truncated before performing a short product; case (d): both inputs might be truncated.

The product at step 4 of **FPmultiply** is a *short product*, i.e., a product whose most significant part only is wanted, as discussed at the start of this Section. In the quadratic range, it can be computed in about half the time of a full product. In the Karatsuba and Toom-Cook ranges, Mulders' algorithm can gain 10% to 20%; however due to carries, using this algorithm for floating-point computations is tricky. Lastly, in the FFT range, no better algorithm is known than computing the full product mm', and then rounding it.

Hence our advice is to perform a full product of m and m', possibly after truncating them to $n + g$ digits if they have more than $n + g$ digits. Here g (the number of *guard digits*) should be positive (see Exercise 3.7.4).

It seems wasteful to multiply n-bit operands, producing a $2n$-bit product,

```
Algorithm FPmultiply.
Input: x = m·β^e, x' = m'·β^{e'}, a precision n, a rounding mode ∘
Output: ∘(xx') rounded to precision n
m'' ← ∘(mm') rounded to precision n
Return m''·β^{e+e'}.
```

only to discard the low-order n bits. Algorithm **ShortProduct** computes an approximation to the short product without computing the $2n$-bit full product.

Error analysis of the short product. Consider two n-word normalised significands A and B that we multiply using a short product algorithm, where the notation **FullProduct**(A, B, n) means the full integer product $A \cdot B$.

```
Algorithm ShortProduct.
Input: integers A, B, and n, with 0 ≤ A, B < β^n
Output: an approximation of AB div β^n
if n ≤ n_0 then return FullProduct(A, B) div β^n
choose k ≥ n/2, ℓ ← n − k
C_1 ← FullProduct(A div β^ℓ, B div β^ℓ) div β^{k−ℓ}
C_2 ← ShortProduct(A mod β^ℓ, B div β^k, ℓ)
C_3 ← ShortProduct(A div β^k, B mod β^ℓ, ℓ)
Return C_1 + C_2 + C_3.
```

Theorem 3.3.1 *The value C' returned by Algorithm **ShortProduct** differs from the exact short product $C = AB$ div β^n by at most $3(n-1)$:*

$$C' \le C \le C' + 3(n-1).$$

Proof. First since A, B are nonnegative, and all roundings are truncations, the inequality $C' \le C$ easily follows.

Let $A = \sum_i a_i \beta^i$ and $B = \sum_j b_j \beta^j$, where $0 \le a_i, b_j < \beta$. The possible errors come from: (i) the neglected $a_i b_j$ terms, i.e., parts C'_2, C'_3, C_4 of Fig. 3.2, (ii) the truncation while computing C_1, (iii) the error in the recursive calls for C_2 and C_3.

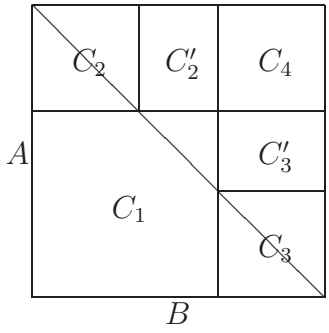


Figure 3.2: Graphical view of Algorithm **ShortProduct**: the computed parts are C_1, C_2, C_3, and the neglected parts are C_2', C_3', C_4 (most significant part bottom left).

We first prove that the algorithm accumulates all products $a_i b_j$ with $i + j \geq n - 1$. This corresponds to all terms on and below the diagonal in Fig. 3.2. The most significant neglected terms are the bottom-left terms from C_2 and C_3, respectively $a_{\ell-1}b_{k-1}$ and $a_{k-1}b_{\ell-1}$. Their contribution is at most $2(\beta-1)^2\beta^{n-2}$. The neglected terms from the next diagonal contribute to at most $4(\beta-1)^2\beta^{n-3}$, and so on. The total contribution of neglected terms is thus bounded by:

$$(\beta-1)^2\beta^n[2\beta^{-2} + 4\beta^{-3} + 6\beta^{-4} + \cdots] < 2\beta^n$$

(the inequality is strict since the sum is finite).

The truncation error in C_1 is at most β^n, thus the maximal difference $\varepsilon(n)$ between C and C' satisfies:

$$\varepsilon(n) < 3 + 2\varepsilon(\lfloor n/2 \rfloor),$$

which gives $\varepsilon(n) < 3(n-1)$, since $\varepsilon(1) = 0$. □

Question: is the upper bound $C' + (n-1)$ attained? Can the theorem be improved?

Remark: if one of the operands was truncated before applying algorithm `ShortProduct`, simply add one unit to the upper bound (the truncated part is less than 1, thus its product by the other operand is bounded by β^n).

3.3.1 Integer Multiplication via Complex FFT

To multiply n-bit integers, it may be advantageous to use the Fast Fourier Tranform (FFT for short, see §1.3.4). Note that the FFT computes the cyclic convolution $z = x * y$ defined by

$$z_k = \sum_{0 \le j < N} x_j y_{k-j \bmod N} \text{ for } 0 \le k < N.$$

In order to use the FFT for integer multiplication, we have to pad the input vectors with zeros, thus increasing the length of the transform from N to $2N$.

FFT algorithms fall into two classes: those using number theoretical properties, and those based on complex floating-point computations. The latter, while not always giving the best known asymptotic complexity, have good practical behaviour, because they take advantage of the efficiency of floating-point hardware. The drawback of the complex floating-point FFT (complex FFT for short) is that, being based on floating-point computations, it requires a rigorous error analysis. However, in some contexts where occasional errors are not disastrous, one may accept a small probability of error if this speeds up the computation. For example, in the context of integer factorisation, a small probability of error is acceptable because the result (a purported factorisation) can easily be checked and discarded if incorrect.

The following theorem provides a tight error analysis of the complex FFT:

Theorem 3.3.2 *The FFT allows computation of the cyclic convolution $z = x * y$ of two vectors of length $N = 2^n$ of complex values such that*

$$||z' - z||_\infty < ||x|| \cdot ||y|| \cdot ((1+\varepsilon)^{3n}(1+\varepsilon\sqrt{5})^{3n+1}(1+\mu)^{3n} - 1), \qquad (3.2)$$

where $||\cdot||$ and $||\cdot||_\infty$ denote the Euclidean and infinity norms respectively, ε is such that $|(a \pm b)' - (a \pm b)| < \varepsilon|a \pm b|$, $|(ab)' - (ab)| < \varepsilon|ab|$ for all machine floats a, b, $\mu > |(w^k)' - (w^k)|$, $0 \le k < N$, $w = e^{\frac{2\pi i}{N}}$, and $(\cdot)'$ refers to the computed (stored) value of $\cdot$ for each expression.

For the IEEE 754 double precision format, with rounding to nearest, we have $\varepsilon = 2^{-53}$, and if the w^k are correctly rounded, we can take $\mu = \varepsilon/\sqrt{2}$. For a fixed FFT size $N = 2^n$, Eq. (3.2) enables one to compute a bound B on the coefficients of x and y, such that $||z' - z||_\infty < 1/2$, which enables one to uniquely round the coefficients of z' to an integer. Table 3.3 gives

n	b	m
2	24	48
3	23	92
4	22	176
5	22	352
6	21	672
7	20	1280
8	20	2560
9	19	4864
10	19	9728

n	b	m
12	17	34816
13	17	69632
14	16	131072
15	16	262144
16	15	491520
17	15	983040
18	14	1835008
19	14	3670016
20	13	6815744

Table 3.3: Maximal number b of bits per IEEE 754 double-precision floating-point number `binary64` (53-bit significand), and maximal m for a plain $m \times m$ bit integer product, for a given FFT size 2^n, with signed coefficients.

$b = \lg B$, the number of bits that can be used in a 64-bit floating-point word, if we wish to perform m-bit multiplication exactly (in fact $m = 2^{n-1}b$). It is assumed that the FFT is performed with signed coefficients in the range $[-2^{b-1}, +2^{b-1})$ – see [50, pg. 161].

Note that Theorem 3.3.2 is a worst-case result; with rounding to nearest we expect the error to be smaller due to cancellation – see Exercise 3.7.9.

3.3.2 The Middle Product

Given two integers of $2n$ and n bits respectively, their "middle product" consists of the n middle bits of their product (see Fig 3.3). The middle product

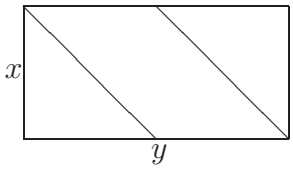


Figure 3.3: The middle product of y of $2n$ bits and x of n bits is the middle region.

might be computed using two short products, one (low) short product between the high part of y and x, and one (high) short product between the low

part of y and x. However there are algorithms to compute a $2n \times n$ middle product with the same $M(n)$ complexity as an $n \times n$ full product (see §3.8).

Several applications may benefit from an efficient middle product. One of those applications is Newton's method (§4.2). Consider for example the reciprocal iteration (§4.2.2) $x_{j+1} = x_j + x_j(1 - x_j y)$. If x_j has n bits, to get $2n$ accurate bits in x_{j+1}, one has to consider $2n$ bits from y. The product $x_j y$ then has $3n$ bits, but if x_j is accurate to n bits, the n most significant bits of $x_j y$ cancel with 1, and the n least significant bits can be ignored in this iteration. Thus what is wanted is exactly the middle product of x_j and y.

Payne and Hanek Argument Reduction

Another application of the middle product is Payne and Hanek argument reduction. Assume $x = m \cdot 2^e$ is a floating-point number with a significand $\frac{1}{2} \leq m < 1$ of n bits and a large exponent e (say $n = 53$ and $e = 1024$ to fix the ideas). We want to compute $\sin x$ with a precision of n bits. The classical argument reduction works as follows: first compute $k = \lfloor x/\pi \rceil$, then compute the reduced argument

$$x' = x - k\pi. \tag{3.3}$$

About e bits will be cancelled in the subtraction $x - (k\pi)$, thus we need to compute $k\pi$ with a precision of at least $e + n$ bits to get an accuracy of at least n bits for x'. Assuming $1/\pi$ has been precomputed to precision e, the computation of k costs $M(e, n)$, and the multiplication $k\pi$ costs $M(e + n)$, thus the total cost is about $M(e)$ when $e \gg n$. The key idea of Payne and

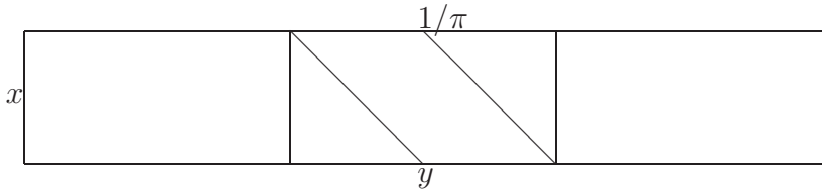


Figure 3.4: A graphical view of Payne and Hanek algorithm.

Hanek algorithm is to rewrite Eq. (3.3) as follows:

$$x' = \pi(\frac{x}{\pi} - k). \tag{3.4}$$

If the significand of x has $n \ll e$ bits, only about $2n$ bits from the expansion of $1/\pi$ will effectively contribute to the n most significant bits of x', namely

the bits of weight 2^{-e-n} to 2^{-e+n}. Let y be the corresponding $2n$-bit part of $1/\pi$. Payne and Hanek's algorithm works as follows: first multiply the n-bit significand of x by y, keep the n middle bits, and multiply by a n-bit approximation of π. The total cost is $M(2n, n) + M(n)$, or even $2M(n)$ if the middle product is performed in time $M(n)$.

3.4 Reciprocal and Division

As for integer operations (§1.4), one should try as much as possible to trade floating-point divisions for multiplications, since the cost of a floating-point multiplication is theoretically smaller than the cost for a division by a constant factor (usually 2 up to 5 depending on the algorithm used). In practice, the ratio might not even be constant, as some implementations provide division with cost $\Theta(M(n) \log n)$ or $\Theta(n^2)$.

When several divisions have to be performed with the same divisor, a well-known trick is to first compute the reciprocal of the divisor (§3.4.1); then each division reduces to a multiplications by the reciprocal. A small drawback is that each division incurs two rounding errors (one for the reciprocal and one for multiplication by the reciprocal) instead of one, so we can no longer guarantee a correctly rounded result. For example, in base ten with six digits, $3.0/3.0$ might evaluate to $0.999999 = 3.0 \times 0.333333$.

The cases of a single division, or several divisions with a varying divisor, are considered in § 3.4.2.

3.4.1 Reciprocal

We describe here algorithms that compute an approximate reciprocal of a positive floating-point number a, using integer-only operations (see Chapter 1). Those integer operations simulate floating-point computations, but all roundings are made explicit. The number a is represented by an integer A of n words in radix β: $a = \beta^{-n}A$, and we assume $\beta^n/2 \leq A$, such that $1/2 \leq a < 1$. (This does not cover all cases for $\beta \geq 3$, however if $\beta^{n-1} \leq A < \beta^n/2$, multiplying A by some appropriate integer $k < \beta$ will reduce to that case, then it suffices to multiply by k the reciprocal of ka.)

We first perform an error analysis of Newton's method (§4.2) assuming all computations are done with infinite precision, thus neglecting roundoff errors.

Lemma 3.4.1 *Let $1/2 \le a < 1$, $\rho = 1/a$, $x > 0$, and $x' = x + x(1 - ax)$. Then:*

$$0 \le \rho - x' \le \frac{x^2}{\theta^3}(\rho - x)^2,$$

for some $\theta \in [\min(x, \rho), \max(x, \rho)]$.

Proof. Newton's iteration is based on approximating the function by its tangent. Let $f(t) = a - 1/t$, with ρ the root of f. The second-order expansion of f at $t = \rho$ with explicit remainder is:

$$f(\rho) = f(x) + (\rho - x)f'(x) + \frac{(\rho - x)^2}{2}f''(\theta),$$

for some $\theta \in [\min(x, \rho), \max(x, \rho)]$. Since $f(\rho) = 0$, this simplifies to

$$\rho = x - \frac{f(x)}{f'(x)} - \frac{(\rho - x)^2}{2}\frac{f''(\theta)}{f'(x)}. \tag{3.5}$$

Substituting $f(t) = a - 1/t$, $f'(t) = 1/t^2$ and $f''(t) = -2/t^3$, it follows that:

$$\rho = x + x(1 - ax) + \frac{x^2}{\theta^3}(\rho - x)^2,$$

which proves the claim. □

Algorithm **ApproximateReciprocal** computes an approximate reciprocal. The input A is assumed to be normalised, i.e., $\beta^n/2 \le A < \beta^n$. The output integer X is an approximation to β^{2n}/A.

```
1  Algorithm ApproximateReciprocal.
2  Input: A = Σ_{i=0}^{n-1} a_i β^i, with 0 ≤ a_i < β and β/2 ≤ a_{n-1}.
3  Output: X = β^n + Σ_{i=0}^{n-1} x_i β^i with 0 ≤ x_i < β.
4  if n ≤ 2 then return ⌈β^{2n}/A⌉ − 1
5  else
6     ℓ ← ⌊(n−1)/2⌋, h ← n − ℓ
7     A_h ← Σ_{i=0}^{h-1} a_{ℓ+i} β^i
8     X_h ← ApproximateReciprocal(A_h)
9     T ← A X_h
10    while T ≥ β^{n+h} do
11       (X_h, T) ← (X_h − 1, T − A)
12    T ← β^{n+h} − T
13    T_m ← ⌊T β^{−ℓ}⌋
14    U ← T_m X_h
15    Return X_h β^h + ⌊U β^{ℓ−2h}⌋.
```

Lemma 3.4.2 *If β is a power of two satisfying $\beta \geq 8$, the output X of Algorithm* ***ApproximateReciprocal*** *satisfies:*

$$AX < \beta^{2n} < A(X+2).$$

Proof. For $n \leq 2$ the algorithm returns $X = \lfloor \frac{\beta^{2n}}{A} \rfloor$, except when $A = \beta^n/2$ where it returns $X = 2\beta^n - 1$. In all cases we have $AX < \beta^{2n} \leq A(X+1)$, thus the lemma holds.

Now consider $n \geq 3$. We have $\ell = \lfloor \frac{n-1}{2} \rfloor$ and $h = n - \ell$, thus $n = h + \ell$ and $h > \ell$. The algorithm first computes an approximate reciprocal of the upper h words of A, and then updates it to n words using Newton's iteration.

After the recursive call at line 8, we have by induction

$$A_h X_h < \beta^{2h} < A_h(X_h + 2). \tag{3.6}$$

After the product $T \leftarrow AX_h$ and the while-loop at steps 10-11, we still have $T = AX_h$, where T and X_h may have new values, and in addition $T < \beta^{n+h}$. We also have $\beta^{n+h} < T + 2A$; we prove the latter by distinguishing two cases. Either we entered the while-loop, then since the value of T decreases by A at each loop, the previous value $T + A$ was necessarily larger or equal to β^{n+h}. If we didn't enter the while-loop, the value of T is the original one $T = AX_h$. Multiplying Eq. (3.6) by β^ℓ gives: $\beta^{n+h} < A_h\beta^\ell(X_h + 2) \leq A(X_h + 2) = T + 2A$. We thus have:

$$T < \beta^{n+h} < T + 2A.$$

It follows $T > \beta^{n+h} - 2A > \beta^{n+h} - 2\beta^n$. As a consequence, the value of $\beta^{n+h} - T$ computed at step 12 cannot exceed $2\beta^n - 1$. The last lines compute the product $T_m X_h$, where T_m is the upper part of T, and put its ℓ most significant words in the low part X_ℓ of the result X.

Now let us perform the error analysis. Compared to Lemma 3.4.1, x stands for $X_h\beta^{-h}$, a stands for $A\beta^{-n}$, and x' stands for $X\beta^{-n}$. The while-loop ensures that we start from an approximation $x < 1/a$, i.e., $AX_h < \beta^{n+h}$. Then Lemma 3.4.1 guarantees that $x \leq x' \leq 1/a$ if x' is computed with infinite precision. Here we have $x \leq x'$, since $X = X_h\beta^h + X_\ell$, where $X_\ell \geq 0$. The only differences with infinite precision are:

- the low ℓ words from $1 - ax$ — here T at line 12 — are neglected, and only its upper part $(1 - ax)_h$ — here T_m — is considered;
- the low $2h - \ell$ words from $x(1 - ax)_h$ are neglected.

Those two approximations make the computed value of x' smaller or equal to the one which would be computed with infinite precision, thus we have for the computed value x':

$$x \le x' \le 1/a.$$

The mathematical error is bounded from Lemma 3.4.1 by $\frac{x^2}{\theta^3}(\rho - x)^2 < 4\beta^{-2h}$ since $\frac{x^2}{\theta^3} \le 1$ and $|\rho - x| < 2\beta^{-h}$. The truncation from $1 - ax$, which is multiplied by $x < 2$, produces an error less than $2\beta^{-2h}$. Finally the truncation of $x(1 - ax)_h$ produces an error less than β^{-n}. The final result is thus:

$$x' \le \rho < x' + 6\beta^{-2h} + \beta^{-n}.$$

Assuming $6\beta^{-2h} \le \beta^{-n}$, which holds as soon as $\beta \ge 6$ since $2h > n$, this simplifies to:

$$x' \le \rho < x' + 2\beta^{-n},$$

which gives with $x' = X\beta^{-n}$ and $\rho = \beta^n/A$:

$$X \le \frac{\beta^{2n}}{A} < X + 2.$$

Since β is assumed to be a power of two, equality can hold only when A is a power of two itself, i.e., $A = \beta^n/2$. In that case there is only one value of X_h that is possible for the recursive call, namely $X_h = 2\beta^h - 1$. In that case $T = \beta^{n+h} - \beta^n/2$ before the while-loop, which is not entered. Then $\beta^{n+h} - T = \beta^n/2$, which multiplied by X_h gives (again) $\beta^{n+h} - \beta^n/2$, whose h most significant words are $\beta - 1$. Thus $X_\ell = \beta^\ell - 1$, and $X = 2\beta^n - 1$: equality does not occur either in that case. □

Remark. The Lemma might be extended to the case $\beta^{n-1} \le A < \beta^n$ or to a radix β which is not a power of two. However we prefer to state this restricted Lemma with simple bounds.

Complexity Analysis. Let $I(n)$ be the cost to invert an n-word number using Algorithm **ApproximateReciprocal**. If we neglect the linear costs, we have $I(n) \approx I(n/2) + M(n, n/2) + M(n/2)$, where $M(n, n/2)$ is the cost of an $n \times (n/2)$ product — the product AX_h at step 9 — and $M(n/2)$ the cost of an $(n/2) \times (n/2)$ product — the product $T_m X_h$ at step 14. If the $n \times (n/2)$ product is performed via two $(n/2) \times (n/2)$ products, we have $I(n) \approx I(n/2) + 3M(n/2)$, which yields $M(n)$ in the quadratic range, $\frac{3}{2}M(n)$ in the Karatsuba range, $\approx 1.704M(n)$ in the Toom-Cook 3-way range, and

$3M(n)$ in the FFT range. In the FFT range, an $n \times (n/2)$ product might be directly computed by an FFT of length $3n/2$ words, which therefore amounts to $M(3n/4)$; in that case the complexity decreases to $2.5M(n)$.

The wrap-around trick. We know describe a slight modification of Algorithm **ApproximateReciprocal** which yields a complexity $2M(n)$. In the product AX_h at step 9, Eq. (3.6) tells that the result approaches β^{n+h}, more precisely:

$$\beta^{n+h} - 2\beta^n < AX_h < \beta^{n+h} + 2\beta^n. \tag{3.7}$$

Assume we use an FFT algorithm such as the Schönhage-Strassen algorithm that computes products modulo β^m+1, for some integer $m \in (n, n+h)$. Let $AX_h = U\beta^m + V$ with $0 \le V < \beta^m$. It follows from Eq. (3.7) that $U = \beta^{n+h-m}$ or $U = \beta^{n+h-m} - 1$. Let $T = AX_h \bmod (\beta^m + 1)$ be the value computed by the FFT. We have $T = V - U$ or $T = V - U + (\beta^m + 1)$. It follows that $AX_h = T + U(\beta^m + 1)$ or $AX_h = T + (U - 1)(\beta^m + 1)$. Taking into account the two possible values of U, we have $AX_h = T + (\beta^{n+h-m} - \varepsilon)(\beta^m + 1)$ where $\varepsilon \in \{0, 1, 2\}$. Since $\beta \ge 6$, $\beta^m > 4\beta^n$, thus only one value of ε yields a value of AX_h in the interval $[\beta^{n+h} - 2\beta^n, \beta^{n+h} + 2\beta^n]$.

We thus replace step 9 in Algorithm **ApproximateReciprocal** by the following code:

```
Compute T = AX_h mod (β^m + 1) using an FFT with m > n
T ← T + β^{n+h} + β^{n+h−m}        { case ε = 0 }
while T ≥ β^{n+h} + 2β^n do
   T ← T − (β^m + 1)
```

Assuming one can take m close to n, the cost of the product AX_h is only about that of an FFT of length n, that is $M(n/2)$.

3.4.2 Division

In this section we consider the case where the divisor changes between successive operations, so no precomputation involving the divisor can be performed. We first show that the number of consecutive zeros in the result is bounded by the divisor length, then we consider the division algorithm and its complexity. Lemma 3.4.3 analyses the case where the division operands are truncated, because they have a larger precision than desired in the result. Finally we discuss "short division" and the error analysis of Barrett's algorithm.

A floating-point division reduces to an integer division as follows. Assume dividend $a = \ell \cdot \beta^e$ and divisor $d = m \cdot \beta^f$, where ℓ, m are integers. Then $\frac{a}{d} = \frac{\ell}{m}\beta^{e-f}$. If k bits of the quotient are needed, we first determine a scaling factor g such that $\beta^{k-1} \leq |\frac{\ell\beta^g}{m}| < \beta^k$, and we divide $\ell\beta^g$ — truncated if needed — by m. The following theorem gives a bound on the number of consecutive zeros after the integer part of the quotient of $\lfloor \ell\beta^g \rfloor$ by m.

Theorem 3.4.1 *Assume we divide an m-digit positive integer by an n-digit positive integer in radix β, with $m \geq n$. Then the quotient is either exact, or its radix β expansion admits at most $n-1$ consecutive zeros or ones after the digit of weight β^0.*

Proof. We first consider consecutive zeros. If the expansion of the quotient q admits n or more consecutive zeros after the binary point, we can write $q = q_1 + \beta^{-n}q_0$, where q_1 is an integer and $0 \leq q_0 < 1$. If $q_0 = 0$, then the quotient is exact. Otherwise, if a is the dividend and d is the divisor, one should have $a = q_1 d + \beta^{-n}q_0 d$. However, a and $q_1 d$ are integers, and $0 < \beta^{-n}q_0 d < 1$, so $\beta^{-n}q_0 d$ cannot be an integer, so we have a contradiction.

For consecutive ones, the proof is similar: write $q = q_1 - \beta^{-n}q_0$, with $0 \leq q_0 \leq 1$. Since $d < \beta_n$, we still have $0 \leq \beta^{-n}q_0 d < 1$. □

The following algorithm performs the division of two n-digit floating-point numbers. The key idea is to approximate the inverse of the divisor to half precision only, at the expense of additional steps. (At step 7, the notation

```
1 Algorithm Divide.
2 Input: n−digit floating-point numbers c and d, with n even
3 Output: an approximation of c/d
4 Write d = d_1 β^{n/2} + d_0 with 0 ≤ d_1, d_0 < β^{n/2}
5 r ← ApproximateReciprocal(d_1, n/2)
6 q_0 ← cr truncated to n/2 digits
7 e ← MP(q_0, d)
8 q ← q_0 − re
```

$MP(q_0, d)$ denotes the middle product of q_0 and d, i.e., the $n/2$ middle digits of that product.) At step 5, r is an approximation to $1/d_1$, and thus to $1/d$, with precision $n/2$ digits. Therefore at step 6, q_0 approximates c/d to about $n/2$ digits, and the upper $n/2$ digits of $q_0 d$ at step 7 agree with those

of c. The value e computed at step 7 thus equals $q_0 d - c$ to precision $n/2$. It follows that $re \approx e/d$ agrees with $q_0 - c/d$ to precision $n/2$; hence the correction term added in the last step.

In the FFT range, the cost of Algorithm **Divide** is $\frac{5}{2}M(n)$: step 5 costs $2M(n/2) \approx M(n)$ with the wrap-around trick, and steps 6-8 each cost $M(n/2)$ — using a fast middle product algorithm for step 7. By way of comparison, if we computed a full precision inverse as in Barrett's algorithm (see below), the cost would be $\frac{7}{2}M(n)$.

In the Karatsuba range, Algorithm **Divide** costs $\frac{3}{2}M(n)$, and is useful provided the middle product of step 7 is performed with cost $M(n/2)$. In the quadratic range, Algorithm **Divide** costs $2M(n)$, and a classical division should be preferred.

When the requested precision for the output is smaller than that of the inputs of a division, one has to truncate the inputs, in order to avoid some unnecessarily expensive computation. Assume for example that one wants to divide two numbers of $10,000$ bits, with a 10-bit quotient. To apply the following lemma, just replace β by an appropriate value such that A_1 and B_1 have about $2n$ and n digits respectively, where n is the wanted number of digits for the quotient; for example one might have $\mu = \beta^k$ to truncate k words.

Lemma 3.4.3 *Let A and B be two positive integers, and $\mu \geq 2$ a positive integer. Let $Q = \lfloor A/B \rfloor$, $A_1 = \lfloor A/\mu \rfloor$, $B_1 = \lfloor B/\mu \rfloor$, $Q_1 = \lfloor A_1/B_1 \rfloor$. If $A/B \leq 2B_1$, then*

$$Q \leq Q_1 \leq Q + 2.$$

The condition $A/B \leq 2B_1$ is quite natural: it says that the truncated divisor B_1 should have essentially at least as many digits as the wanted quotient.

Proof. Let $A_1 = Q_1 B_1 + R_1$. We have $A = A_1\mu + A_0$, $B = B_1\mu + B_0$, thus

$$\frac{A}{B} = \frac{A_1\mu + A_0}{B_1\mu + B_0} \leq \frac{A_1\mu + A_0}{B_1\mu} = Q_1 + \frac{R_1\mu + A_0}{B_1\mu}.$$

Since $R_1 < B_1$ and $A_0 < \mu$, $R_1\mu + A_0 < B_1\mu$, thus $A/B < Q_1 + 1$. Taking the floor of each side proves, since Q_1 is an integer, that $Q \leq Q_1$.

Now consider the second inequality. For given truncated parts A_1 and B_1, and thus given Q_1, the worst case is when A is minimal, say $A = A_1\beta$,

and B is maximal, say $B = B_1\beta + (\beta - 1)$. In this case we have:

$$\left|\frac{A_1}{B_1} - \frac{A}{B}\right| = \left|\frac{A_1}{B_1} - \frac{A_1\beta}{B_1\beta + (\beta-1)}\right| = \left|\frac{A_1(\beta-1)}{B_1(B_1\beta+\beta-1)}\right|.$$

The numerator equals $A - A_1 \le A$, and the denominator equals $B_1 B$, thus the difference $A_1/B_1 - A/B$ is bounded by $A/(B_1 B) \le 2$, and so is the difference between Q and Q_1. □

The following algorithm is useful in the Karatsuba and Toom-Cook range. The key idea is that, when dividing a $2n$-digit number by an n-digit number, some work that is necessary for a full $2n$-digit division can be avoided (see Fig. 3.5).

```
1  Algorithm ShortDivision.
2  Input: 0 ≤ A < β^{2n}, β^n/2 ≤ B < β^n
3  Output: an approximation of A/B
4  if n ≤ n_0 then return ⌊A/B⌋
5  choose k ≥ n/2, ℓ ← n − k
6  (A_1, A_0) ← (A div β^{2ℓ}, A mod β^{2ℓ})
7  (B_1, B_0) ← (B div β^ℓ, B mod β^ℓ)
8  (Q_1, R_1) ←DivRem(A_1, B_1)
9  A' ← R_1 β^{2ℓ} + A_0 − Q_1 B_0 β^ℓ
10 Q_0 ←ShortDivision(A' div β^k, B div β^k)
11 Return Q_1 β^ℓ + Q_0.
```

Theorem 3.4.2 *The approximate quotient Q' returned by* `ShortDivision` *differs at most by $2\lg n$ from the exact quotient $Q = \lfloor A/B \rfloor$, more precisely:*

$$Q \le Q' \le Q + 2\lg n.$$

Proof. If $n \le n_0$, $Q = Q'$ so the statement holds. Assume $n > n_0$. We have $A = A_1\beta^{2\ell} + A_0$ and $B = B_1\beta^\ell + B_0$, thus since $A_1 = Q_1 B_1 + R_1$, $A = (Q_1 B_1 + R_1)\beta^{2\ell} + A_0 = Q_1 B \beta^\ell + A'$, with $A' < \beta^{n+\ell}$. Let $A' = A'_1\beta^k + A'_0$, and $B = B'_1\beta^k + B'_0$, with $0 \le A'_0, B'_0 < \beta^k$, and $A'_1 < \beta^{2\ell}$. From Lemma 3.4.3, the exact quotient of A' div β^k by B div β^k is greater or equal to that of A' by B, thus by induction $Q_0 \ge A'/B$. Since $A/B = Q_1\beta^\ell + A'/B$, this proves that $Q' \ge Q$.

Now by induction $Q_0 \leq \frac{A'_1}{B'_1} + 2\lg \ell$, and $\frac{A'_1}{B'_1} \leq A'/B + 2$ (from Lemma 3.4.3 again, whose hypothesis $A'/B \leq 2B'_1$ is satisfied, since $A' < B_1\beta^{2\ell}$, thus $A'/B \leq \beta^l \leq 2B'_1$), so $Q_0 \leq A'/B + 2\lg n$, and $Q' \leq A/B + 2\lg n$. □

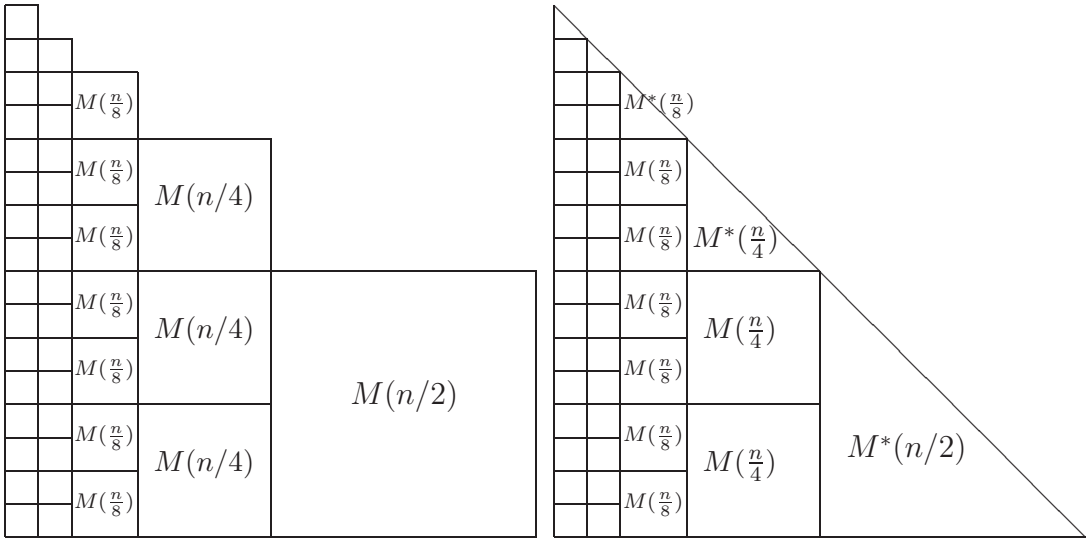


Figure 3.5: Divide and conquer short division: a graphical view. Left: with plain multiplication; right: with short multiplication. See also Fig. 1.1.

Barrett's division algorithm

Here we consider division using Barrett's algorithm (§2.3.1) and provide a rigorous error bound. This algorithm is useful when the same divisor is used several times; otherwise Algorithm **Divide** is faster (see Exercise 3.7.12). Assume we want to divide a by b of n bits, with a quotient of n bits. Barrett's algorithm is as follows:

1. Compute the reciprocal r of b to n bits [rounding to nearest]
2. $q \leftarrow \circ_n(a \times r)$ [rounding to nearest]

The cost of the algorithm in the FFT range is $3M(n)$: $2M(n)$ to compute the reciprocal with the wrap-around trick, and $M(n)$ for the product $a \times r$.

Lemma 3.4.4 *At step 2 of Barrett's algorithm, we have* $|a - bq| \leq \frac{3}{2}|b|$.

Proof. By scaling a and b, we can assume that b and q are integers, $2^{n-1} \leq b, q < 2^n$, thus $a < 2^{2n}$. We have $r = \frac{1}{b} + \varepsilon$ with $|\varepsilon| \leq \frac{1}{2}\operatorname{ulp}(2^{-n}) = 2^{-2n}$. Also $q = ar + \varepsilon'$ with $|\varepsilon'| \leq \frac{1}{2}\operatorname{ulp}(q) = \frac{1}{2}$ since q has n bits. Thus $q = a(\frac{1}{b} + \varepsilon) + \varepsilon' = \frac{a}{b} + a\varepsilon + \varepsilon'$, and $|bq - a| = |b||a\varepsilon + \varepsilon'| \leq \frac{3}{2}|b|$. □

As a consequence, q differs by at most one unit in last place from the n-bit quotient of a and b, rounded to nearest.

Lemma 3.4.4 can be applied as follows: to perform several divisions with a precision of n bits with the same divisor, precompute a reciprocal with $n + g$ bits, and use the above algorithm with a working precision of $n + g$ bits. If the last g bits of q are neither $000\ldots00x$ nor $111\ldots11x$ (where x stands for 0 or 1), then rounding q down to n bits will yield $\circ_n(a/b)$ for a directed rounding mode.

3.5 Square Root

Algorithm **FPSqrt** computes a floating-point square root, using as subroutine Algorithm **SqrtRem** to determine an integer square root (with remainder). It assumes an integer significand m, and a directed rounding mode (see Exercise 3.7.13 for rounding to nearest).

```
1 Algorithm FPSqrt.
2 Input: x = m·2^e, a target precision n, a rounding mode ∘
3 Output: y = ∘_n(√x)
4 If e is odd, (m', f) ← (2m, e−1), else (m', f) ← (m, e)
5 If m' has less than 2n−1 bits, then (m', f) ← (m'2^{2ℓ}, f−2ℓ)
6 Write m' := m_1 2^{2k} + m_0, m_1 having 2n or 2n−1 bits, 0 ≤ m_0 < 2^{2k}
7 (s, r) ← SqrtRem(m_1)
8 If round to zero or down or r = m_0 = 0, return s·2^{k+f/2}
9 else return (s+1)·2^{k+f/2}.
```

Theorem 3.5.1 *Algorithm **FPSqrt** returns the square root of x, correctly-rounded.*

Proof. Since m_1 has $2n$ or $2n-1$ bits, s has exactly n bits, and we have $x \geq s^2 2^{2k+f}$, thus $\sqrt{x} \geq s2^{k+f/2}$. On the other hand, `SqrtRem` ensures that

$r \le 2s$, thus $x2^{-f} = (s^2+r)2^{2k}+m_0 < s^2+r+1 \le (s+1)^2$. Since $y := s\cdot 2^{k+f/2}$ and $y^+ = (s+1)\cdot 2^{k+f/2}$ are two consecutive n-bit floating-point numbers, this concludes the proof. □

NOTE: in the case $s = 2^n - 1$, $s + 1 = 2^n$ is still representable with n bits, and y^+ is in the upper binade.

An different method is to use a subroutine computing an approximation to a reciprocal square root (§3.5.1), as follows:

```
1 Algorithm FPSqrt2.
2 Input: an n−bit floating−point number x
3 Output: a n−bit approximation y of √x
4 r ←ApproximateRecSquareRoot(x)
5 t ← ∘_{n/2}(hr)
6 u ← x − t^2
7 Return t + (r/2)u
```

Step 5 costs $M(n/2)$. Since the $n/2$ most significant bits of t^2 are known to match those of x in step 6, we can perform a transform mod $x^{n/2} - 1$ in the FFT range, hence step 6 costs $M(n/4)$. Finally step 7 costs $M(n/2)$. In the FFT range, with a cost of $\frac{7}{2}M(n/2)$ for **ApproximateRecSquareRoot**(x) (§3.5.1), the total cost is $3M(n)$. (See §3.8 for faster algorithms.)

3.5.1 Reciprocal Square Root

In this section we describe an algorithm to compute the reciprocal square root $a^{-1/2}$ of a floating-point number a, with a rigorous error bound.

Lemma 3.5.1 *Let $a, x > 0$, $\rho = a^{-1/2}$, and $x' = x + \frac{x}{2}(1 - ax^2)$. Then*

$$0 \le \rho - x' \le \frac{3}{2}\frac{x^3}{\theta^4}(\rho - x)^2,$$

for some $\theta \in [\min(x, \rho), \max(x, \rho)]$.

Proof. The proof is very similar to that of Lemma 3.4.1. Here we use $f(t) = a - 1/t^2$, with ρ the root of f. Eq. (3.5) translates to:

$$\rho = x + \frac{x}{2}(1 - ax^2) + \frac{3}{2}\frac{x^3}{\theta^4}(\rho - x)^2,$$

which proves the Lemma. □

```
1  Algorithm ApproximateRecSquareRoot.
2  Input: integer A with β^n ≤ A < 4β^n
3  Output: integer X, β^n/2 ≤ X < β^n
4  if n ≤ 2 then return min(β^n − 1, ⌊β^n/√(Aβ^{−n})⌋)
5  else
6     ℓ ← ⌊(n−1)/2⌋, h ← n − ℓ
7     A_h ← ⌊Aβ^{−ℓ}⌋
8     X_h ←ApproximateRecSquareRoot(A_h)
9     T ← A(X_h^2)
10    T_h ← ⌊Tβ^{−n}⌋
11    T_ℓ ← β^{2h} − T_h
12    U ← T_ℓ X_h
13    Return min(β^n − 1, X_h β^ℓ + ⌊Uβ^{ℓ−2h}/2⌋).
```

NOTE: even if $A_h X_h^2 \leq \beta^{3h}$ at line 8, we might have $AX_h^2 > \beta^{n+2h}$ at line 9, which might cause T_ℓ to be negative.

Lemma 3.5.2 *As long as $\beta \geq 38$, if X is the value returned by Algorithm **ApproximateRecSquareRoot**, $a = A\beta^{-n}$, and $x = X\beta^{-n}$, then $1/2 \leq x < 1$ and*

$$|x - a^{-1/2}| \leq 2\beta^{-n}.$$

Proof. We have $1 \leq a < 4$. Since X is bounded by $\beta^n - 1$ at lines 4 and 13, we have $x, x_h < 1$, with $x_h = X_h\beta^{-h}$. We prove the statement by induction. It is true for $n \leq 2$. Now assume the value X_h at step 8 satisfies:

$$|x_h - a_h^{-1/2}| \leq \beta^{-h},$$

where $a_h = A_h\beta^{-h}$. We have three sources of error, that we will bound in this order:

1. the rounding errors in lines 10 and 13;

2. the mathematical error given by Lemma 3.5.1, which would occur even if all computations were exact;

3. the error coming from the fact we use A_h instead of A in the recursive call at step 8.

At step 9 we have exactly:

$$t := T\beta^{-n-2h} = ax_h^2,$$

which gives $|t_h - ax_h^2| < \beta^{-2h}$ with $t_h := T_h\beta^{-2h}$, and in turn $|t_\ell - (1-ax_h^2)| < \beta^{-2h}$ with $t_\ell := T_\ell\beta^{-2h}$. At step 12, it follows $|u - x_h(1-ax_h^2)| < \beta^{-2h}$, where $u = U\beta^{-3h}$. Thus finally $|x - [x_h + \frac{x_h}{2}(1-ax_h^2)]| < \frac{1}{2}\beta^{-2h} + \frac{1}{2}\beta^{-n}$, taking into account the rounding error in the last step.

Now we apply Lemma 3.5.1 to $x \to x_h$, $x' \to x$, to bound the mathematical error, assuming no rounding error occurs:

$$0 \le a^{-1/2} - x \le \frac{3}{2}\frac{x_h^3}{\theta^4}(a^{-1/2} - x_h)^2,$$

which gives[1] $|a^{-1/2} - x| \le 3.04(a^{-1/2} - x_h)^2$. Now $|a^{-1/2} - a_h^{-1/2}| \le \frac{1}{2}|a - a_h|\nu^{-3/2}$ for $\nu \in [\min(a_h, a), \max(a_h, a)]$, thus $|a^{-1/2} - a_h^{-1/2}| \le \beta^{-h}/2$. Together with the induction hypothesis $|x_h - a_h^{-1/2}| \le 2\beta^{-h}$, it follows that $|a^{-1/2} - x_h| \le \frac{5}{2}\beta^{-h}$. Thus $|a^{-1/2} - x| \le 19\beta^{-2h}$.

The total error is thus bounded by:

$$|a^{-1/2} - x| \le 3/2\beta^{-n} + 19\beta^{-2h}.$$

Since $2h \ge n+1$, we see that $19\beta^{-2h} \le 1/2\beta^{-n}$ for $\beta \ge 38$, and the proof follows. □

Let $R(n)$ be the cost of **ApproximateRecSquareRoot** for an n-bit input. We have $h, \ell \approx n/2$, thus the recursive call costs $R(n/2)$, step 9 costs $M(n/2)$ to compute X_h^2, and $M(n)$ for the product $A(X_h^2)$ — or $M(3n/4)$ in the FFT range using the wrap-around trick described in §3.4.1, since we know the upper $n/2$ bits of the product gives 1 — and again $M(n/2)$ for step 12. We get $R(n) = R(n/2) + 2M(n)$ — $R(n/2) + \frac{7}{4}M(n)$ in the FFT range —, which yields $R(n) = 4M(n)$ — $\frac{7}{2}M(n)$ in the FFT range.

The above algorithm is not the optimal one in the FFT range, especially when using an FFT algorithm with cheap point-wise products (like the complex FFT, see §3.3.1). Indeed, Algorithm **ApproximateRecSquareRoot** uses the following form of Newton's iteration:

$$x' = x + \frac{x}{2}(1 - ax^2).$$

[1] Since $\theta \in [x_h, a^{-1/2}]$ and $|x_h - a^{-1/2}| \le \frac{5}{2}\beta^{-h}$, we have $\theta \ge x_h - \frac{5}{2}\beta^{-h}$, thus $x_h/\theta \le 1 + 5\beta^{-h}/(2\theta) \le 1 + 5\beta^{-h}$ (remember $\theta \in [x_h, a^{-1/2}]$), and it follows that $\theta \ge 1/2$. For $\beta \ge 38$, since $h \ge 2$, we have $1 + 5\beta^{-h} \le 1.0035$, thus $\frac{3}{2}x_h^3/\theta^4 \le (3/(2\theta))1.0035^3 \le 3.04$.

It might be better to write:

$$x' = x + \frac{1}{2}(x - ax^3).$$

Indeed, the product x^3 might be computed with a *single* FFT transform of length $3n/2$, replacing the point-wise products $\hat{x}_i^2$ by $\hat{x}_i^3$, with a total cost of about $\frac{3}{4}M(n)$. Moreover, the same idea can be used for the full product ax^3 of $5n/2$ bits, but whose $n/2$ upper bits match those of x, thus with the wrap-around trick a transform of length $2n$ is enough, with a cost of $M(n)$ for the last iteration, and a total cost of $2M(n)$ for the reciprocal square root. With that result, Algorithm **FPSqrt2** costs $2.25M(n)$ only.

3.6 Conversion

Since most software tools work in radix 2 or 2^k, and humans usually enter or read floating-point numbers in radix 10 or 10^k, conversions are needed from one radix to the other one. Most applications perform only very few conversions, in comparison to other arithmetic operations, thus the efficiency of the conversions is rarely critical[2]. The main issue here is therefore more correctness than efficiency. Correctness of floating-point conversions is not an easy task, as can be seen from the past bugs in Microsoft Excel[3].

The algorithms described in this section use as subroutine the integer-conversion algorithms from Chapter 1. As a consequence, their efficiency depends on the efficiency of the integer-conversion algorithms.

3.6.1 Floating-Point Output

In this section we follow the convention of using small letters for parameters related to the internal radix b, and capitals for parameters related to the external radix B. Consider the problem of printing a floating-point number, represented internally in radix b (say $b = 2$) in an external radix B (say $B = 10$). We distinguish here two kinds of floating-point output:

[2] An important exception is the computation of billions of digits of constants like $\pi, \log 2$, where a quadratic conversion routine would be far too slow.

[3] In Excel 2007, the product 850×77.1 prints as $100,000$ instead of $65,535$; this is really an output bug, since if one multiplies "$100,000$" by 2, one gets $131,070$. An input bug occurred in Excel 3.0 to 7.0, where the input 1.40737488355328 gave 0.64.

- fixed-format output, where the output precision is given by the user, and we want the output value to be correctly rounded according to the given rounding mode. This is the usual method when values are to be used by humans, for example to fill a table of results. In that case the input and output precision may be very different: for example one may want to print 1000 digits of 2/3, which uses only one digit internally in radix 3. Conversely, one may want to print only a few digits of a number accurate to 1000 bits.

- free-format output, where we want the output value, when read with correct rounding (usually to nearest), to give back the initial number. Here the minimal number of printed digits may depend on the input number. This kind of output is useful when storing data in a file, while guaranteeing that reading the data back will produce *exactly* the same internal numbers, or for exchanging data between different programs.

In other words, if we denote by x the number we want to print, and X the printed value, the fixed-format output requires $|x - X| < \mathrm{ulp}(X)$, and the free-format output requires $|x - X| < \mathrm{ulp}(x)$ for directed rounding. Replace $< \mathrm{ulp}(\cdot)$ by $\leq \frac{1}{2}\,\mathrm{ulp}(\cdot)$ for rounding to nearest.

```
1  Algorithm PrintFixed.
2  Input: x = f · b^{e−p} with f, e, p integers, b^{p−1} ≤ |f| < b^p,
3         external radix B and precision P, rounding mode ∘
4  Output: X = F · B^{E−P} with F, E integers, B^{P−1} ≤ |F| < B^P,
5          such that X = ∘(x) in radix B and precision P
6  λ ← o(log b/log B)
7  E ← 1 + ⌊(e − 1)λ⌋
8  q ← ⌈P/λ⌉
9  y ← ∘(xB^{P−E}) with precision q
10 If one cannot round y to an integer, increase q and goto 9
11 F ← Integer(y, ∘).
12 If |F| ≥ B^P then E ← E + 1 and goto 9.
13 Return F, E.
```

Some comments on Algorithm **PrintFixed**:

- it assumes that we have precomputed values of $\lambda_B = o(\frac{\log b}{\log B})$ for any possible external radix B (the internal radix b is assumed to be fixed for

a given implementation). Assuming the input exponent e is bounded, it is possible — see Exercise 3.7.15 — to choose these values precisely enough that

$$E = 1 + \left\lfloor (e-1)\frac{\log b}{\log B} \right\rfloor, \tag{3.8}$$

thus the value of λ at step 6 is simply read from a table.

- the difficult part is step 9, where one has to perform the exponentiation B^{P-E} — remember all computations are done in the internal radix b — and multiply the result by x. Since we expect an integer of q digits in step 11, there is no need to use a precision of more than q digits in these computations, but a rigorous bound on the rounding errors is required, so as to be able to correctly round y.

- in step 10, "one can round y to an integer" means that the interval containing all possible values of xB^{P-E} — including the rounding errors while approaching xB^{P-E}, and the error while rounding to precision q — contains no rounding boundary (if $\circ$ is a directed rounding, it should contain no integer; if $\circ$ is rounding to nearest, it should contain no half-integer).

Theorem 3.6.1 *Algorithm* ***PrintFixed*** *is correct.*

Proof. First assume that the algorithm finishes. Eq. (3.8) implies $B^{E-1} \leq b^{e-1}$, thus $|x|B^{P-E} \geq B^{P-1}$, which implies that $|F| \geq B^{P-1}$ at step 11. Thus $B^{P-1} \leq |F| < B^P$ at the end of the algorithm. Now, printing x gives $F \cdot B^a$ iff printing xB^k gives $F \cdot B^{a+k}$ for any integer k. Thus it suffices to check that printing xB^{P-E} gives F, which is clear by construction.

The algorithm terminates because at step 9, xB^{P-E}, if not an integer, cannot be arbitrarily close to an integer. If $P - E \geq 0$, let k be the number of digits of B^{P-E} in radix b, then xB^{P-E} can be represented exactly with $p + k$ digits. If $P - E < 0$, let $g = B^{E-P}$, of k digits in radix b. Assume $f/g = n + \varepsilon$ with n integer; then $f - gn = g\varepsilon$. If ε is not zero, $g\varepsilon$ is a non-zero integer, thus $|\varepsilon| \geq 1/g \geq 2^{-k}$.

The case $|F| \geq B^P$ at step 12 can occur for two reasons: either $|x|B^{P-E} \geq B^P$, thus its rounding also satisfies this inequality; or $|x|B^{P-E} < B^P$, but its rounding equals B^P (this can only occur for rounding away from zero or to nearest). In the former case we have $|x|B^{P-E} \geq B^{P-1}$ at the next pass

in step 9, while in the latter case the rounded value F equals B^{P-1} and the algorithm terminates. □

Now consider free-format output. For a directed rounding mode we want $|x - X| < \text{ulp}(x)$ knowing $|x - X| < \text{ulp}(X)$. Similarly for rounding to nearest, if we replace ulp by $\frac{1}{2}\,\text{ulp}$.

It is easy to see that a sufficient condition is that $\text{ulp}(X) \leq \text{ulp}(x)$, or equivalently $B^{E-P} \leq b^{e-p}$ in Algorithm **PrintFixed** (with P not fixed at input, which explain the "free-format" name). To summarise, we have

$$b^{e-1} \leq |x| < b^e, \quad B^{E-1} \leq |X| < B^E.$$

Since $|x| < b^e$, and X is the rounding of x, we must have $B^{E-1} \leq b^e$. It follows that $B^{E-P} \leq b^e B^{1-P}$, and the above sufficient condition becomes:

$$P \geq 1 + p\,\frac{\log b}{\log B}.$$

For example, with $b = 2$ and $B = 10$, $p = 53$ gives $P \geq 17$, and $p = 24$ gives $P \geq 9$. As a consequence, if a double-precision IEEE 754 binary floating-point number is printed with at least 17 significant decimal digits, it can be read back without any discrepancy, assuming input and output are performed with correct rounding to nearest (or directed rounding, with appropriately chosen directions).

3.6.2 Floating-Point Input

The problem of floating-point input is the following. Given a floating-point number X with a significand of P digits in some radix B (say $B = 10$), a precision p and a given rounding mode, we want to correctly round X to a floating-point number x with p digits in the internal radix b (say $b = 2$).

At first glance, this problem looks very similar to the floating-point output problem, and one might think it suffices to apply Algorithm **PrintFixed**, simply exchanging (b, p, e, f) and (B, P, E, F). Unfortunately, this is not the case. The difficulty is that, in Algorithm **PrintFixed**, all arithmetic operations are performed in the internal radix b, and we do not have such operations in radix B (see however Ex. 1.8.29).

3.7 Exercises

Exercise 3.7.1 Determine exactly for which IEEE 754 double precision numbers does the trick described in §3.1.5 work, to get the next number away from zero.

Exercise 3.7.2 (Kidder, Boldo) Assume a binary representation. The "rounding to odd" mode [22, 84, 126] is defined as follows: in case the exact value is not representable, it rounds to the unique adjacent number with an odd significand. ("Von Neumann rounding" [22] omits the test for the exact value being representable or not, and rounds to odd in all nonzero cases.) Note that overflow never occurs during rounding to odd. Prove that if $y = \text{round}(x, p + k, \text{odd})$ and $z = \text{round}(y, p, \text{nearest\_even})$, and $k > 1$, then $z = \text{round}(x, p, \text{nearest\_even})$, i.e., the double-rounding problem does not occur.

Exercise 3.7.3 Show that, if $\sqrt{a}$ is computed using Newton's iteration for $a^{-1/2}$:

$$x' = x + \frac{3}{2}(1 - ax^2)$$

(see §3.5.1) and the identity $\sqrt{a} = a \times a^{-1/2}$ with rounding mode "round towards zero", then it might never be possible to determine the correctly rounded value of $\sqrt{a}$, regardless of the number of additional "guard" digits used in the computation.

Exercise 3.7.4 How does truncating the operands of a multiplication to $n + g$ digits (as suggested in §3.3) affect the accuracy of the result? Considering the cases $g = 1$ and $g > 1$ separately, what could happen if the same strategy were used for subtraction?

Exercise 3.7.5 Is the bound of Theorem 3.3.1 optimal?

Exercise 3.7.6 Adapt Mulders' short product algorithm [102] to floating-point numbers. In case the first rounding fails, can you compute additional digits without starting again from scratch?

Exercise 3.7.7 If a balanced ternary system is used, that is radix 3 with possible digits $\{0, \pm 1\}$, then "round to nearest" is equivalent to truncation.

Exercise 3.7.8 (Percival) One computes the product of two complex floating point numbers $z_0 = a_0 + ib_0$ and $z_1 = a_1 + ib_1$ in the following way: $x_a = \circ(a_0 a_1)$, $x_b = \circ(b_0 b_1)$, $y_a = \circ(a_0 b_1)$, $y_b = \circ(a_1 b_0)$, $z = \circ(x_a - x_b) + \circ(y_a + y_b) \cdot i$. All computations being done in precision n, with rounding to nearest, compute an error bound of the form $|z - z_0 z_1| \leq c 2^{-n} |z_0 z_1|$. What is the best possible c?

Exercise 3.7.9 Show that, if $\mu = O(\varepsilon)$ and $n\varepsilon \ll 1$, the bound in Theorem 3.3.2 simplifies to

$$||z' - z||_\infty = O(|x| \cdot |y| \cdot n\varepsilon).$$

If the rounding errors cancel we expect the error in each component of z' to be $O(|x| \cdot |y| \cdot n^{1/2}\varepsilon)$. The error $||z' - z||_\infty$ could be larger since it is a maximum of $N = 2^n$ component errors. Using your favourite implementation of the FFT, compare the worst-case error bound given by Theorem 3.3.2 with the error $||z' - z||_\infty$ that occurs in practice.

Exercise 3.7.10 (Enge) Design an algorithm that correctly rounds the product of two complex floating-point numbers with 3 multiplications only. [Hint: assume all operands and the result have n-bit significand.]

Exercise 3.7.11 Write a computer program to check the entries of Table 3.3 are correct and optimal.

Exercise 3.7.12 To perform k divisions with the same divisor, which of Algorithm **Divide** and of Barrett's algorithm is the fastest one?

Exercise 3.7.13 Adapt Algorithm **FPSqrt** to the rounding to nearest mode.

Exercise 3.7.14 Prove that for any n-bit floating-point numbers $(x, y) \neq (0, 0)$, and if all computations are correctly rounded, with the same rounding mode, the result of $x/\sqrt{x^2 + y^2}$ lies in $[-1, 1]$, except in a special case. What is this special case?

Exercise 3.7.15 Show that the computation of E in Algorithm **PrintFixed**, step 7, is correct — i.e., $E = 1 + \left\lfloor (e-1)\frac{\log b}{\log B} \right\rfloor$ — as long as there is no integer n such that $\left| \frac{n}{e-1}\frac{\log B}{\log b} - 1 \right| < \varepsilon$, where ε is the relative precision when computing λ: $\lambda = \frac{\log B}{\log b}(1 + \theta)$ with $|\theta| \leq \varepsilon$. For a fixed range of exponents $-e_{\max} \leq e \leq e_{\max}$, deduce a working precision ε. Application: for $b = 2$, and $e_{\max} = 2^{31}$, compute the required precision for $3 \leq B \leq 36$.

Exercise 3.7.16 (Lefèvre) The IEEE 754 standard requires binary to decimal conversions to be correctly rounded in the range $m \cdot 10^n$ for $|m| \leq 10^{17} - 1$ and $|n| \leq 27$ in double precision. Find the hardest-to-print double precision number in that range (with rounding to nearest for example). Write a C program that outputs double precision numbers in that range, and compare it to the `sprintf` C-language function of your system. Same question for a conversion from the IEEE 754R `binary64` format (significand of 53 bits, $2^{-1074} \leq |x| < 2^{1024}$) to the `decimal64` format (significand of 16 decimal digits).

Exercise 3.7.17 Same question as the above, for the decimal to binary conversion, and the `atof` C-language function.

3.8 Notes and References

In her PhD [94, Chapter V], Valérie Ménissier-Morain discusses alternatives to the classical non-redundant representation considered here: continued fractions and redundant representations. She also considers in Chapter III the theory of computable reals, their representation by B-adic numbers, and the computation of algebraic or transcendental functions.

Nowadays most computers use radix two, but other choices (for example radix 16) were popular in the past, before the widespread adoption of the IEEE 754 standard. A discussion of the best choice of radix is given in [22].

The main reference for floating-point arithmetic is the IEEE 754 standard [3], which defines four binary formats: single precision, single extended (deprecated), double precision, and double extended. The IEEE 854 standard [45] defines radix-independent arithmetic, and mainly decimal arithmetic. Both standards are replaced by the revision of IEEE 754 (approved by the IEEE Standards Committee on June 12, 2008).

The rule regarding the precision of a result given possibly differing precisions of operands was considered in [29, 71].

Floating-point expansions were introduced by Priest [108]. They are mainly useful for a small numbers of summands, mainly two or three, and when the main operations are additions or subtractions. For a larger number of summands the combinatorial logic becomes complex, even for addition. Also, except for simple cases, it seems difficult to obtain correct rounding with expansions.

Some good references on error analysis of floating-point algorithms are the books by Higham [70] and Muller [103]. Older references include Wilkinson's classics [131, 132].

Collins, Krandick and Lefèvre proposed algorithms for multiple-precision floating-point addition [47, 88].

The problem of leading zero anticipation and detection in hardware is classical; see [110] for a comparison of different methods.

The idea of having a "short product" together with correct rounding was studied by Krandick and Johnson [83] in 1993, where they attribute the term "short product" to Knuth. They considered both the schoolbook

and the Karatsuba domains. In 2000 Mulders [102] invented an improved "short product" algorithm based on Karatsuba multiplication, and also a "short division" algorithm. The problem of consecutive zeros or ones — also called runs of zeros or ones — has been studied by several authors in the context of computer arithmetic: Iordache and Matula [73] studied division (Theorem 3.4.1), square root, and reciprocal square root. Muller and Lang [86] generalised their results to algebraic functions.

A description of the Fast Fourier Transform (FFT) using complexfloating-point numbers can be found in Knuth [81]. See also the asymptotically faster algorithm by Fürer [59]. Many variations of the FFT are discussed in the books by Crandall [49, 50]. For further references, see the Notes and References section of Chapter 2.

Theorem 3.3.2 is from Percival [106]: previous rigorous error analyses of complex FFT gave very pessimistic bounds. Note that the proof given in [106] is incorrect, but we have a correct proof (see [37] and Ex. 3.7.8).

The concept of "middle product" for power series is discussed in [65]. Bostan, Lecerf and Schost have shown it can be seen as a special case of "Tellegen's principle", and have generalised it to operations other than multiplication [21]. The link between usual multiplication and the middle product using trilinear forms was mentioned by Victor Pan in [104] for the multiplication of two complex numbers: "*The duality technique enables us to extend any successful bilinear algorithms to two new ones for the new problems, sometimes quite different from the original problem* $\cdots$" A detailed and comprehensive description of the Payne and Hanek argument reduction method can be found in [103].

The $2M(n)$ reciprocal algorithm — with the wrap-around trick — of § 3.4.1, is due to Schönhage, Grotefeld and Vetter. [115], It can be improved, as noticed by Bernstein [9]. If we keep the FFT-transform of x, we can save $\frac{1}{3}M(n)$ (assuming the term-to-term products have negligible cost), which gives $\frac{5}{3}M(n)$. Bernstein also proposes a "messy" $\frac{3}{2}M(n)$ algorithm [9]. Schönhage's $\frac{3}{2}M(n)$ algorithm is better [114]. The idea is to write Newton's iteration as $x' = 2x - ax^2$. If x is accurate to $n/2$ bits, then ax^2 has (in theory) $2n$ bits, but we know the upper $n/2$ bits cancel with x, and we are not interested in the low n bits. Thus we can perform one modular FFT of size $3n/2$, which amounts to cost $M(3n/4)$. See also [48] for the roundoff error analysis when using a floating-point multiplier.

Bernstein in [9] obtains faster square root algorithms in the FFT domain, by caching some Fourier transforms, more precisely he gets $\frac{11}{6}M(n)$ for the

square root, and $\frac{5}{2}M(n)$ for the simultaneous computation of $x^{1/2}$ and $x^{-1/2}$.

Classical floating-point conversion algorithms are due to Steele and White [121], Gay [60], and Clinger [44]; most of these authors assume fixed precision. Mike Cowlishaw maintains an extensive bibliography of conversion to and from decimal arithmetic (see §5.3). What we call "free-format" output is called "idempotent conversion" by Kahan [77]; see also Knuth [81, exercise 4.4-18].

Algebraic Complexity Theory by Bürgisser, Clausen and Shokrollahi [40] is an excellent book on topics including lower bounds, fast multiplication of numbers and polynomials, and Strassen-like algorithms for matrix multiplication.

There is a large literature on interval arithmetic, which is outside the scope of this chapter. A good entry point is the Interval Computations web page (see the Appendix). See also the recent book by Kulisch [85].

This chapter does not consider complex arithmetic, except where relevant for its use in the FFT. An algorithm for the complex (floating-point) square root, which allows correct rounding, is given in [55].

Chapter 4

Newton's Method and Function Evaluation

In this Chapter we consider various applications of Newton's method, which can be used to compute reciprocals, square roots, and more generally algebraic and functional inverse functions. We then consider unrestricted algorithms for computing elementary and special functions. The algorithms of this Chapter are presented at a higher level than in Chapter 3. Rounding issues are not discussed, apart from a few exceptions where they are significant. A full and detailed analysis of one special function might be the subject of an entire chapter!

4.1 Introduction

This Chapter is concerned with algorithms for computing elementary and special functions, although the methods apply more generally. First we consider Newton's method, which is useful for computing inverse functions. For example, if we have an algorithm for computing $y = \log x$, then Newton's method can be used to compute $x = \exp y$ (see §4.2.5). However, Newton's method has many other applications. In fact we already mentioned Newton's method in Chapters 1–3, but here we consider it in more detail.

After considering Newton's method, we go on to consider various methods for computing elementary and special functions. These methods include power series (§4.4), asymptotic expansions (§4.5), continued fractions

(§4.6), recurrence relations (§4.7), the arithmetic-geometric mean (§4.8), binary splitting (§4.9), and contour integration (§4.10). The methods that we consider are *unrestricted* in the sense that there is no restriction on the attainable precision — in particular, it is not limited to the precision of IEEE standard 32-bit or 64-bit floating-point arithmetic. Of course, this depends on the availability of a suitable software package for performing floating-point arithmetic on operands of arbitrary precision, as discussed in Chapter 3.

Unless stated explicitly, we do not consider rounding issues in this Chapter; it is assumed that methods described in Chapter 3 are used. Also, to simplify the exposition, we assume a binary radix, although most of the content might be extended to any radix. We recall that n denotes the precision — hence in bits — of the wanted approximation; in most cases the absolute computed value will be in the neighbourhood of 1, thus we want an approximation to within 2^{-n}.

4.2 Newton's Method

Newton's method is a major tool in arbitrary precision arithmetic. We have already seen it or its p-adic counterpart, namely Hensel's lifting, in previous Chapters (see for example Algorithm **ExactDivision** in §1.4.5, or the algorithm to compute a modular inverse in §2.4, or the algorithms to compute a floating-point reciprocal or reciprocal square root in §3.4.1 and §3.5.1). Newton's method is also useful in small precision: most modern processors only implement multiplication in hardware, and division and square root are microcoded, using Newton's method. This Section discusses Newton's method is more detail, in the context of floating-point computations, for the computation of inverse roots (§4.2.1), reciprocals (§4.2.2), reciprocal square roots (§4.2.3), formal power series (§4.2.4), and functional inverse functions (§4.2.5). We also discuss higher order Newton-like methods (§4.2.6).

Newton's Method via Linearisation

Recall that a function f of a real variable is said to have a *zero* ζ if $f(\zeta) = 0$. Similarly for functions several real (or complex) variables. If f is differentiable in a neighbourhood of ζ, and $f'(\zeta) \neq 0$, then ζ is said to be a *simple* zero. In the case of several variables, ζ is a simple zero if the Jacobian matrix evaluated at ζ is nonsingular.

Newton's method for approximating a simple zero ζ of f is based on the idea of making successive linear approximations to $f(x)$ a the neighbourhood of ζ. Suppose that x_0 is an initial approximation, and that $f(x)$ has two continuous derivatives in the region of interest. From Taylor's theorem[1]

$$f(\zeta) = f(x_0) + (\zeta - x_0)f'(x_0) + \frac{(\zeta - x_0)^2}{2}\, f''(\xi) \tag{4.1}$$

for some point ξ in an interval including $\{\zeta, x_0\}$. Since $f(\zeta) = 0$, we see that

$$x_1 = x_0 - f(x_0)/f'(x_0)$$

is an approximation to ζ, and

$$x_1 - \zeta = O\left(|x_0 - \zeta|^2\right).$$

Provided x_0 is sufficiently close to ζ, we will have

$$|x_1 - \zeta| \leq |x_0 - \zeta|/2 < 1.$$

This motivates the definition of *Newton's method* as the iteration

$$x_{j+1} = x_j - \frac{f(x_j)}{f'(x_j)}\,, \quad j = 0, 1, \ldots \tag{4.2}$$

Provided $|x_0 - \zeta|$ is sufficiently small, we expect x_n to converge to ζ and the *order of convergence* will be at least 2, that is

$$|e_{n+1}| \leq K|e_n|^2$$

for some constant K independent of n, where $e_n = x_n - \zeta$ is the error after n iterations.

A more careful analysis [125] — see also Lemma 3.4.1 — shows that

$$e_{n+1} = \frac{f''(\zeta)}{2f'(\zeta)}\, {e_n}^2 + O\left({e_n}^3\right),$$

provided $f \in C^3$ near ζ. Thus, the order of convergence is exactly 2 if $f''(\zeta) \neq 0$ and e_0 is sufficiently small but nonzero. (Such an iteration is also said to be *quadratically convergent*.)

[1] We use Taylor's theorem at x_0, since this yields a formula in terms of derivatives at x_0 which is known, instead of at ζ, which is unknown.

4.2.1 Newton's Method for Inverse Roots

Consider applying Newton's method to the function

$$f(x) = y - x^{-m},$$

where m is a positive integer constant, and (for the moment) y is a nonzero constant. Since $f'(x) = mx^{-(m+1)}$, Newton's iteration simplifies to

$$x_{j+1} = x_j + x_j(1 - x_j^m y)/m. \tag{4.3}$$

This iteration converges to $\zeta = y^{-1/m}$ provided the initial approximation x_0 is sufficiently close to ζ. It is perhaps surprising that (4.3) does not involve divisions, except for a division by the integer constant m. Thus, we can easily compute reciprocals (the case $m = 1$) and reciprocal square roots (the case $m = 2$) by Newton's method. These cases are sufficiently important that we discuss them separately in the following subsections.

4.2.2 Newton's Method for Reciprocals

Taking $m = 1$ in (4.3), we obtain the iteration

$$x_{j+1} = x_j + x_j(1 - x_j y) \tag{4.4}$$

which we expect to converge to $1/y$ provided x_0 is a sufficiently good approximation. To see what "sufficiently good" means, define

$$u_j = 1 - x_j y.$$

Note that $u_j \to 0$ if and only if $x_j \to 1/y$. Multiplying each side of (4.4) by y, we get

$$1 - u_{j+1} = (1 - u_j)(1 + u_j),$$

which simplifies to

$$u_{j+1} = u_j^2. \tag{4.5}$$

Thus

$$u_j = (u_0)^{2^j}. \tag{4.6}$$

We see that the iteration converges if and only if $|u_0| < 1$, which (for real x_0 and y) is equivalent to the condition $x_0 y \in (0, 2)$. Second-order convergence

is reflected in the double exponential with exponent 2 on the right-hand-side of (4.6).

The iteration (4.4) is sometimes implemented in hardware to compute reciprocals of floating-point numbers, see for example [79]. The sign and exponent of the floating-point number are easily handled, so we can assume that $y \in [0.5, 1.0)$ (recall we assume a binary radix in this Chapter). The initial approximation x_0 is found by table lookup, where the table is indexed by the first few bits of y. Since the order of convergence is two, the number of correct bits approximately doubles at each iteration. Thus, we can predict in advance how many iterations are required. Of course, this assumes that the table is initialised correctly[2].

Computational Issues

At first glance, it seems better to replace Eq. 4.4 by

$$x_{j+1} = x_j(2 - x_j y),$$

which looks simpler. However, although those two forms are mathematically equivalent, they are not computationally equivalent. Indeed, in Eq. 4.4, if x_j approximates $1/y$ to within $n/2$ bits, then $1 - x_j y = O(2^{-n/2})$, and the product of x_j by $1 - x_j y$ might be computed with a precision of $n/2$ bits only. In the above — apparently simpler — form, $2 - x_j y = 1 + O(2^{-n/2})$, thus the product of x_j by $2 - x_j y$ has to be performed with a full precision of n bits.

As a general rule, it is usually better to keep apart the terms of different order in Newton's iteration, and to not try to factor common expressions (see however the discussion about the $2M(n)$ reciprocal algorithm in §3.8).

4.2.3 Newton's Method for (Reciprocal) Square Roots

Taking $m = 2$ in (4.3), we obtain the iteration

$$x_{j+1} = x_j + x_j(1 - x_j^2 y)/2, \tag{4.7}$$

[2]In the case of the infamous *Pentium* `fdiv` *bug* [64], a lookup table used for division was initialised incorrectly, and the division was occasionally inaccurate. Although the algorithm used in the Pentium did not involve Newton's method, the moral is the same – tables must be initialised correctly.

which we expect to converge to $y^{-1/2}$ provided x_0 is a sufficiently good approximation.

If we want to compute $y^{1/2}$, we can do this in one multiplication after first computing $y^{-1/2}$, since

$$y^{1/2} = y \times y^{-1/2}.$$

This method does not involve any divisions (except by 2). In contrast, if we apply Newton's method to the function $f(x) = x^2 - y$, we obtain Heron's[3] iteration (see Algorithm **SqrtInt** in §1.5.1)

$$x_{j+1} = \frac{1}{2}\left(x_j + \frac{y}{x_j}\right) \tag{4.8}$$

for the square root of y. This requires a division by x_j at iteration j, so it is essentially different from the iteration (4.7). Although both iterations have second-order convergence, we expect (4.7) to be more efficient (however this might depend on the relative cost of division with respect to multiplication).

4.2.4 Newton's Method for Formal Power Series

(This section is not required for function evaluation, however it gives a complementary point of view on Newton's method.)

Newton's method can be applied to find roots of functions defined by formal power series as well as of functions of a real or complex variable. For simplicity we consider formal power series of the form

$$A(z) = a_0 + a_1 z + a_2 z^2 + \cdots$$

where $a_i \in \mathbb{R}$ (or any field of characteristic zero) and $\mathrm{ord}(A) = 0$, i.e., $a_0 \neq 0$.

For example, if we replace y in (4.4) by $1 - z$, and take initial approximation $x_0 = 1$, we obtain a quadratically-convergent iteration for the formal power series

$$(1 - z)^{-1} = \sum_{n=0}^{\infty} z^n.$$

In the case of formal power series, "quadratically convergent" means that $\mathrm{ord}(e_j) \to +\infty$ like 2^j. In our example, with the notations of §4.2.2, $u_0 =$

[3] Heron of Alexandria, *circa* 10–75 AD.

$1 - x_0 y = z$, so $u_j = z^{2^j}$ and

$$x_j = \frac{1 - u_j}{1 - z} = \frac{1}{1 - z} + O\left(z^{2^j}\right).$$

Some operations on formal power series have no analogue for integers. For example, given a formal power series $A(z) = \sum_{j \geq 0} a_j z^j$, we can define the formal *derivative*

$$A'(z) = \sum_{j>0} j a_j z^{j-1} = a_1 + 2a_2 z + 3a_3 z^2 + \cdots,$$

and the *integral*

$$\sum_{j \geq 0} \frac{a_j}{j+1}\, x^{j+1},$$

but there is no useful analogue for multiple-precision integers

$$\sum_{j=0}^{n} a_j \beta^j.$$

4.2.5 Newton's Method for Functional Inverses

Given a function $g(x)$, its functional inverse $h(x)$ satisfies $g(h(x)) = x$, which is also denoted $h(x) := g^{(-1)}(x)$. For example $g(x) = \log x$ and $h(x) = \exp x$ are functional inverses, or $g(x) = \tan x$ and $h(x) = \arctan x$. Using the function $f(x) = y - g(x)$ in Eq. (4.2), one gets a root ζ of f, i.e., a value such that $g(\zeta) = y$, or $\zeta = g^{(-1)}(y)$:

$$x_{j+1} = x_j + \frac{y - g(x_j)}{g'(x_j)}.$$

Since this iteration only involves g and g', it provides an efficient way to evaluate $h(y)$, assuming that $g(x_j)$ and $g'(x_j)$ can be efficiently computed. Moreover if the complexity of evaluating g' is less or equal to that of g, we get a means to evaluate the functional inverse h of g within the same complexity.

As an example, if one has an efficient implementation of the logarithm, a similarly efficient implementation of the exponential is deduced as follows. Consider the root e^y of the function $f(x) = y - \log x$, which yields the iteration:

$$x_{j+1} = x_j + x_j(y - \log x_j),$$

and in turn the following algorithm (for sake of simplicity, we consider here only one Newton iteration):

```
1 Algorithm LiftExp.
2 Input: x_j, (n/2)-bit approximation to exp(y).
3 Output: x_{j+1}, n-bit approximation to exp(y).
4 t ← log x_j [computed to n-bit accuracy]
5 u ← y − t [computed to (n/2)-bit accuracy]
6 v ← x_j u [computed to (n/2)-bit accuracy]
7 x_{j+1} ← x_j + v
```

4.2.6 Higher Order Newton-like Methods

The classical Newton's method is based on a linear approximation of $f(x)$ around $f(x_0)$. If one consider a higher order approximation, one get a higher order method. Consider for example an order 2 approximation. Eq. 4.1 becomes:

$$f(\zeta) = f(x_0) + (\zeta - x_0)f'(x_0) + \frac{(\zeta - x_0)^2}{2}f''(x_0) + \frac{(\zeta - x_0)^3}{6}f'''(\xi).$$

Since $f(\zeta) = 0$, and neglecting the 3rd-order term, one gets:

$$\zeta = x_0 - \frac{f(x_0)}{f'(x_0)} - \frac{(\zeta - x_0)^2}{2}\frac{f''(x_0)}{f'(x_0)} + O((\zeta - x_0)^3).$$

One difficulty here is that ζ is not known in the second-order term. Let $\zeta = x_0 - \frac{f(x_0)}{f'(x_0)} + \nu$, where ν is the second-order error term. Replacing in the above equation yields the iteration:

$$x_{j+1} = x_j - \frac{f(x_j)}{f'(x_j)} - \frac{f(x_j)^2 f''(x_j)}{2f'(x_j)^3}.$$

For the computation of the reciprocal with $f(x) = y - 1/x$ (§4.2.2), this yields

$$x_{j+1} = x_j + x_j(1 - x_j y) + x_j(1 - x_j y)^2.$$

For the computation of $\exp x$ using functional inversion (§4.2.5), one gets:

$$x_{j+1} = x_0 + x_0(y - \log x_0) + \frac{1}{2}x_0(y - \log x_0)^2.$$

4.3 Argument Reduction

Argument reduction is a classical method to improve the efficiency of the evaluation of mathematical functions. The key idea is to reduce the initial problem to a domain where the function is easier to evaluate. More precisely, given $f(x)$ to evaluate, one proceeds in three steps:

- *argument reduction*: x is transformed into a *reduced argument* x';
- *evaluation*: $f(x')$ is evaluated;
- *reconstruction*: $f(x)$ is computed from $f(x')$ using a functional identity.

In some cases the argument reduction or the reconstruction is trivial, for example $x' = x/2$ in radix 2, or $f(x) \approx f(x')$ (some examples will illustrate that below). It might also be that the evaluation step uses a different function $g(x')$ instead of $f(x')$; for example $\sin(x + \pi/2) = \cos(x)$, thus with $f = \sin$ and $x' = x - \pi/2$, we have $g = \cos$.

Note that argument reduction is only possible when a functional identity relies $f(x)$ and $f(x')$ — or $g(x')$. The elementary functions have *addition formulae* such as

$$\exp(x + y) = \exp(x)\exp(y),$$
$$\log(xy) = \log(x) + \log(y),$$
$$\sin(x + y) = \sin(x)\cos(y) + \cos(x)\sin(y),$$
$$\tan(x + y) = \frac{\tan(x) + \tan(y)}{1 - \tan(x)\tan(y)}.$$

We can use these formulæ to reduce the argument so that power series converge more rapidly. Usually we take $x = y$ to get *doubling formulae* such as

$$\exp(2x) = \exp(x)^2, \tag{4.9}$$

though occasionally *tripling formulae* such as

$$\sin(3x) = 3\sin(x) - 4\sin^3(x)$$

might be useful (indeed, $\sin(2x) = 2\sin x \cos x$ involves two auxiliary functions, see however §4.9.1). Unfortunately, such functional identities do not exist for every function; for example, no argument reduction is known for the error function.

One usually distinguishes two kinds of argument reduction:

- *multiplicative argument reduction* where $x' = x/c^k$ for some real constant c and some integer k. This occurs for the computation of $\exp x$ when using the doubling formula $\exp(2x) = (\exp x)^2$;

- *additive argument reduction* where $x' = x - kc$, for some real constant c and some integer k. This occurs in particular when $f(x)$ is periodic, for example for the sine and cosine functions with $c = 2\pi$.

Note that for a given function, the two kinds of argument reduction might be available. For example, for $\sin x$, one might either use the tripling formula $\sin(3x) = 3\sin x - 4\sin^3 x$, or alternatively $\sin(x + 2k\pi) = \sin x$.

4.3.1 Repeated Use of Doubling Formulæ

If we apply the doubling formula for $\exp$ k times, we get

$$\exp(x) = \exp(x/2^k)^{2^k}.$$

Thus, if $|x| = O(1)$, we can reduce the problem of evaluating $\exp(x)$ to that of evaluating $\exp(x/2^k)$, where the argument is now $O(2^{-k})$. The extra cost is the k squarings that we need to get the final result from $\exp(x/2^k)$.

There is a trade-off here and k should be chosen to minimise the total time. If the obvious method for power series evaluation is used, then the optimal k is of order $\sqrt{n}$ and the overall time is $O(n^{1/2}M(n))$. (We'll see soon that there are faster ways to evaluate power series, so this is not the best possible result.)

We assumed here that $|x| = \Theta(1)$. A more careful analysis shows that the optimal k depends on the order of magnitude of x (see Ex. 4.13.4).

4.3.2 Loss of Precision

For some power series, especially those with alternate signs, a loss of precision might occur due to a cancellation between successive terms. An extreme example is $\exp x$ for $x < 0$. Assume for example we want 10 significant digits of $\exp(-10)$. The first ten terms $x^k/k!$ are:

$$1., -10., 50., -166.6666667, 416.6666667, -833.3333333, 1388.888889,$$
$$-1984.126984, 2480.158730, -2755.731922.$$

If we add the first 51 terms with a working precision of 10 digits, we get an approximation of $exp(-10)$ that is accurate to about 3 digits only!

In that case we might use

$$\exp(x) = 1/\exp(-x)$$

instead to avoid cancellation in the power series summation. In other cases an alternate power series without sign changes might exist.

4.3.3 Guard Digits

Care has to be taken to use the right number of guard digits, and/or the right working precision. Consider once again our running example of $\exp x$, with reduced argument $x/2^k$. Since $x/2^k$ is $O(2^{-k})$, when we sum the power series $1 + x/2^k + \cdots$ from left to right, we "lose" about k bits of precision. Indeed, if $x/2^k + \cdots$ is accurate to n bits, then $1 + x/2^k + \cdots$ is accurate to $n + k$ bits, but if we use the same working precision n, one will obtain n correct bits only. After squaring k times in the reconstruction step, about k bits will be lost due to rounding errors, thus the final accuracy will be $n - k$ only. If one would sum the power series in reverse order instead, and use a working precision of $n + k$ when adding 1 and $x/2^k + \cdots$, one would obtain an accuracy of $n + k$ bits before the k squarings, and of n bits finally.

Another way to avoid this loss of precision is to evaluate $\mathrm{expm1}(x/2^k)$, where the function expm1 is defined by

$$\mathrm{expm1}(x) = \exp(x) - 1$$

and has a doubling formula that avoids loss of significance when $|x|$ is small. See Exercises 4.13.6–4.13.8.

4.3.4 Doubling versus Tripling Formula

It is more efficient to do argument reduction via the doubling formula (4.9) for exp than the tripling formula for sinh:

$$\sinh(3x) = \sinh(x)(3 + 4\sinh^2(x)),$$

because it takes one multiplication and one squaring (which may be cheaper) to apply the tripling formula, but only two squarings to apply the doubling formula twice (and $3 < 2^2$).

4.4 Power Series

Once argument reduction has been applied, whenever possible (§4.3), one is usually faced with the evaluation of a power series. The elementary and special functions have power series expansions such as:

$$\begin{aligned}
\exp x &= \sum_{j\ge 0} \frac{x^j}{j!}, \\
\log(1+x) &= \sum_{j\ge 0} \frac{(-1)^j x^{j+1}}{j+1}, \\
\arctan x &= \sum_{j\ge 0} \frac{(-1)^j x^{2j+1}}{2j+1}, \\
\sinh x &= \sum_{j\ge 0} \frac{x^{2j+1}}{(2j+1)!},
\end{aligned}$$

This section discusses several techniques to recommend or to avoid. We use the following notations: x is the evaluation point, n is the wanted precision, and d is the order of the power series, or the degree of the corresponding polynomial.

If $f(x)$ is analytic in a neighbourhood of some point c, an obvious method to consider for the evaluation of $f(x)$ is summation of the Taylor series

$$f(x) = \sum_{j=0}^{d-1} (x-c)^j \; f^{(j)}(c)/j! + R_d(x,c).$$

As a simple but instructive example we consider the evaluation of $\exp(x)$ for $|x| \le 1$, using

$$\exp(x) = \sum_{j=0}^{d-1} x^j/j! + R_d(x), \tag{4.10}$$

where $|R_d(x)| \le |x|^d \exp(|x|)/d! \le e/d!$.

Using Stirling's approximation for $d!$, we see that $d \ge K(n) \sim n/\lg n$ is sufficient to ensure that $|R_d(x)| = O(2^{-n})$. Thus the time required to evaluate (4.10) with Horner's rule is $O(nM(n)/\log n)$.

In practice it is convenient to sum the series in the forward direction ($j = 0, 1, \ldots, d-1$). The terms $T_j = x^j/j!$ and partial sums

$$S_j = \sum_{i=0}^{j} T_i$$

may be generated by the recurrence $T_j = x \times T_{j-1}/j$, $S_j = S_{j-1} + T_j$, and the summation terminated when $|T_d| < 2^{-n}/e$. Thus, it is not necessary to estimate d in advance, as it would be if the series were summed by Horner's rule in the backward direction ($j = d-1, d-2, \ldots, 0$) (see however Ex. 4.13.3).

We now consider the effect of rounding errors, under the assumption that floating-point operations are correctly rounded, i.e., satisfy

$$\circ(x \text{ op } y) = (x \text{ op } y)(1+\delta),$$

where $|\delta| \le \varepsilon$ and "op" = "+", "−", "×" or "/". Here $\varepsilon = \frac{1}{2}\beta^{1-n}$ is the "machine precision" or "working precision". Let $\widehat{T}_j$ be the computed value of T_j, etc. Thus

$$|\widehat{T}_j - T_j| \,/\, |T_j| \;\le\; 2j\varepsilon + O(\varepsilon^2)$$

and

$$\begin{aligned} |\widehat{S}_d - S_d| &\le de\varepsilon + \sum_{j=1}^{d} 2j\varepsilon|T_j| + O(\varepsilon^2) \\ &\le (d+2)e\varepsilon + O(\varepsilon^2) = O(n\varepsilon). \end{aligned}$$

Thus, to get $|\widehat{S}_d - S_d| = O(2^{-n})$ it is sufficient that $\varepsilon = O(2^{-n}/n)$, i.e., we need to work with about $\log_\beta n$ guard digits. This is not a significant overhead if (as we assume) the number of digits may vary dynamically. We can sum with j increasing (the *forward* direction) or decreasing (the *backward* direction). A slightly better error bound is obtainable for summation in the backward direction, but this method has the disadvantage that the number of terms d has to be decided in advance. If we sum in the forward direction then d can be determined dynamically by checking if $|T_j|$ is small enough (assuming, of course, that $|T_j|$ is decreasing rapidly, which should be true if appropriate argument reduction is used).

In practice it is inefficient to keep the working precision ε fixed. We can profitably reduce it when computing T_j from T_{j-1} if $|T_{j-1}| \ll 1$, without

significantly increasing the error bound. We can also vary the working precision when accumulating the sum, provided that it is computed in the forward direction.

It is instructive to consider the effect of relaxing our restriction that $|x| \leq 1$. First suppose that x is large and positive. Since $|T_j| > |T_{j-1}|$ when $j < |x|$, it is clear that the number of terms required in the sum (4.10) is at least of order $|x|$. Thus, the method is slow for large $|x|$ (see §4.3 for faster methods in this case, if applicable).

If $|x|$ is large and x is negative, the situation is even worse. From Stirling's approximation we have

$$\max_{j \geq 0} |T_j| \simeq \frac{\exp|x|}{\sqrt{2\pi|x|}},$$

but the result is $\exp(-|x|)$, so about $2|x|/\log\beta$ guard digits are required to compensate for what Lehmer called "catastrophic cancellation" [57]. Since $\exp(x) = 1/\exp(-x)$, this problem may easily be avoided, but the corresponding problem is not always so easily avoided for other analytic functions.

Here is a less trivial example where power series expansions are useful. To compute the error function

$$\operatorname{erf}(x) = 2\pi^{-1/2} \int_0^x e^{-u^2}\, du,$$

we may use the series

$$\operatorname{erf}(x) = 2\pi^{-1/2} \sum_{j=0}^{\infty} \frac{(-1)^j\, x^{2j+1}}{j!(2j+1)} \tag{4.11}$$

or

$$\operatorname{erf}(x) = 2\pi^{-1/2} \exp(-x^2) \sum_{j=0}^{\infty} \frac{2^j\, x^{2j+1}}{1 \cdot 3 \cdot 5 \cdots (2j+1)}. \tag{4.12}$$

The series (4.12) is preferable to (4.11) for moderate $|x|$ because it involves no cancellation. For large $|x|$ neither series is satisfactory, because $\Omega(x^2)$ terms are required, and it is preferable to use the asymptotic expansion or continued fraction for $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$: see §§4.5–4.6.

In the following subsections we consider different methods to evaluate power series. We generally ignore the effect of rounding errors, but the results

obtained above are typical. For an example of an extremely detailed error analysis of an "unrestricted" algorithm, see [43]. Here *unrestricted* means that there is no *a priori* bound on $|x|$, the precision or the exponent range.

Power Series to Avoid

In some cases the coefficients in the series are nontrivial to evaluate. For example,

$$\tan x = \sum_{j\geq 1} \frac{T_j}{(2j-1)!}\, x^{2j-1},$$

where the constants T_j are called *tangent numbers* and can be expressed in terms of Bernoulli numbers. In such cases it is best to avoid direct power series evaluation.

For example, to evaluate $\tan x$ we can use Newton's method on the inverse function (arctan, see §4.2.5), or we can use $\tan x = \sin x / \cos x$.

Thus, we'll assume for the moment that we have a power series $\sum_{j\geq 0} a_j x^j$ where a_{j+1}/a_j is a rational function $R(j)$ of j, and hence it is easy to evaluate $a_0, a_1, a_2, \ldots$ sequentially. For example, in the case of $\exp x$,

$$\frac{a_{j+1}}{a_j} = \frac{j!}{(j+1)!} = \frac{1}{j+1}.$$

In general, our assumptions cover hypergeometric functions.

The Radius of Convergence

If the elementary function is an entire function (e.g., exp, sin) then the power series converges in the whole complex plane. In this case the degree of the denominator of $R(j) = a_{j+1}/a_j$ is greater than that of the numerator.

In other cases (such as log, arctan) the function is not entire and the power series only converges in a disk in the complex plane because the function has a singularity on the boundary of this disk. In fact $\log(x)$ has a singularity at the origin, which is why we consider the power series for $\log(1+x)$. This power series has radius of convergence 1.

Similarly, the power series for $\arctan(x)$ has radius of convergence 1 because $\arctan(x)$ has a singularity on the unit circle (even though it is uniformly bounded for all real x).

4.4.1 Direct Power Series Evaluation

Using periodicity (in the cases of $\sin, \cos$) or argument reduction techniques (§4.3), we can assume that we want to evaluate a power series $\sum_{j\geq 0} a_j x^j$ where $|x| \leq 1/2$ and the radius of convergence of the series is at least 1.

As before, assume that a_{j+1}/a_j is a rational function of j, and hence easy to evaluate.

To sum the series with error $O(2^{-n})$ it is sufficient to take $n+O(1)$ terms, so the time required is

$$O(nM(n)).$$

If the function is entire, then the series converges faster and the time is reduced to

$$O\left(\frac{nM(n)}{\log n}\right).$$

However, we can do much better by carrying the argument reduction further!

4.4.2 Power Series With Argument Reduction

By applying argument reduction $k + O(1)$ times, we can ensure that the argument x satisfies (say for $\exp x$)

$$|x| < 2^{-k}.$$

Then, to obtain n-bit accuracy we only need to sum $O(n/k)$ terms in the power series. Assuming that a step of argument reduction is $O(M(n))$, which is true for the elementary functions, the total cost is

$$O\left((k + n/k)M(n)\right).$$

Indeed, the argument reduction and/or reconstruction requires $O(k)$ steps of $O(M(n))$, and the evaluation of the power series of order n/k costs $n/kM(n)$; so choosing $k \sim n^{1/2}$ gives cost

$$O\left(n^{1/2}M(n)\right).$$

Examples

For example, this applies to the evaluation of $\exp(x)$ using

$$\exp(x) = \exp(x/2)^2,$$

to $\mathrm{log1p}(x) = \log(1+x)$ using

$$\mathrm{log1p}(x) = 2\,\mathrm{log1p}\left(\frac{x}{1+\sqrt{1+x}}\right),$$

and to $\arctan(x)$ using

$$\arctan x = 2\arctan\left(\frac{x}{1+\sqrt{1+x^2}}\right).$$

Note that in the last two cases each step of the argument reduction requires a square root, but this can be done with cost $O(M(n))$ by Newton's method (§3.5). Thus in all three cases the overall cost is

$$O\left(n^{1/2}M(n)\right),$$

although the implicit constant might be smaller for exp than for log1p or arctan.

Using Symmetries

A not-so-well-known idea is to evaluate $\log(1+x)$ using the power series

$$\log\left(\frac{1+y}{1-y}\right) = 2\sum_{j\geq 0}\frac{y^{2j+1}}{2j+1}$$

with y defined by $(1+y)/(1-y) = 1+x$, i.e., $y = x/(2+x)$. This saves half the terms and also reduces the argument, since $y < x/2$ if $x > 0$. Unfortunately this nice idea can be applied only once. For another example, see Ex. 4.13.9.

4.4.3 The Rectangular Series Splitting

Once we determine how many terms in the power series are required for the desired accuracy, the problem reduces to evaluating a truncated power series, i.e., a polynomial.

Let $P(x) = \sum_{0 \le j < d} a_j x^j$ be the polynomial one wants to evaluate. In the general case x is a floating-point number of n bits, and we aim at an accuracy of n bits for $P(x)$. However the coefficients a_j, or their ratios $R(j) = a_{j+1}/a_j$, are usually small integer or rational numbers of $O(\log n)$ bits. A *scalar multiplication* involves one coefficient a_j and the variable x — or more generally an n-bit floating-point number —, whereas a nonscalar multiplication involves two powers of x — or more generally two n-bit floating-point numbers. Scalar multiplications are cheaper because a_j are small rationals of size $O(\log n)$, whereas x and its powers have $O(n)$ bits. It is possible to evaluate $P(x)$ with $O(\sqrt{n})$ nonscalar multiplications (plus $O(n)$ scalar multiplications and $O(n)$ additions, using $O(\sqrt{n})$ storage). The same idea applies, more generally, to evaluation of hypergeometric functions.

The Classical Splitting

Suppose $d = jk$, define $y = x^k$, and write

$$P(x) = \sum_{\ell=0}^{j-1} y^\ell P_\ell(x) \quad \text{where} \quad P_\ell(x) = \sum_{m=0}^{k-1} a_{k\ell+m}\, x^m.$$

One first computes the powers $x^2, x^3, \ldots, x^{k-1}, x^k = y$, then the polynomials $P_\ell(x)$ are evaluated by simply multiplying $a_{k\ell+m}$ and the precomputed x^m — it is very important *not to use* Horner's rule with respect to x here, since it would imply expensive nonscalar multiplications — and finally $P(x)$ is computed from the $P_\ell(x)$ using Horner's rule with respect to y. To see the idea geometrically, write $P(x)$ as

$$\begin{array}{rcccccccccc}
y^0\,[a_0 & + & a_1x & + & a_2x^2 & + & \cdots & + & a_{k-1}x^{k-1}] & + \\
y^1\,[a_k & + & a_{k+1}x & + & a_{k+2}x^2 & + & \cdots & + & a_{2k-1}x^{k-1}] & + \\
y^2\,[a_{2k} & + & a_{2k+1}x & + & a_{2k+2}x^2 & + & \cdots & + & a_{3k-1}x^{k-1}] & + \\
 & \vdots & & \vdots & & \vdots & & & & \\
y^{j-1}\,[a_{(j-1)k} & + & a_{(j-1)k+1}x & + & a_{(j-1)k+2}x^2 & + & \cdots & + & a_{jk-1}x^{k-1}] &
\end{array}$$

where $y = x^k$. The terms in square brackets are the polynomials $P_0(x)$, $P_1(x)$, ..., $P_{j-1}(x)$.

As an example, consider $d = 12$, with $j = 3$ and $k = 4$. This would give $P_0(x) = a_0 + a_1x + a_2x^2 + a_3x^3$, $P_1(x) = a_4 + a_5x + a_6x^2 + a_7x^3$, $P_2(x) = a_8 + a_9x + a_{10}x^2 + a_{11}x^3$, then $P(x) = P_0(x) + yP_1(x) + y^2P_2(x)$, where $y = x^4$.

Here we need to compute x^2, x^3, x^4, which requires three nonscalar products, and we need two nonscalar products to evaluate $P(x)$, thus a total of five nonscalar products, instead of $d-2=10$ with a naive application of Horner's rule on $P(x)$.

The Modular Splitting

An alternate splitting is the following, which corresponds to transpose the matrix of coefficients above, swap j and k, and interchange the powers of x and y. It might also be viewed as a generalized odd-even scheme (§1.3.5). Still suppose $d = jk$, and write with $y = x^j$:

$$P(x) = \sum_{\ell=0}^{j-1} x^\ell P_\ell(x^j) \quad \text{where} \quad P_\ell(y) = \sum_{m=0}^{k-1} a_{jm+\ell}\, y^m.$$

First compute $y = x^j, y^2, y^3, \ldots, y^{k-1}$. Now the polynomials $P_\ell(y)$ can be evaluated using only scalar multiplications of the form $a_{jm+\ell}y^m$.

To see the idea geometrically, write $P(x)$ as

$$\begin{array}{rccccccc}
x^0 & [a_0 & + & a_j y & + & a_{2j} y^2 & + & \cdots] & + \\
x^1 & [a_1 & + & a_{j+1} y & + & a_{2j+1} y^2 & + & \cdots] & + \\
x^2 & [a_2 & + & a_{j+2} y & + & a_{2j+2} y^2 & + & \cdots] & + \\
 & \vdots & & \vdots & & \vdots & & & \\
x^{j-1} & [a_{j-1} & + & a_{2j-1} y & + & a_{3j-1} y^2 & + & \cdots] &
\end{array}$$

where $y = x^j$. We traverse the first row of the array, then the second row, then the third, ..., finally the j-th row, accumulating sums $S_0, S_1, \ldots, S_{j-1}$ (one for each row). At the end of this process $S_\ell = P_\ell(y)$ and we only have to evaluate

$$P(x) = \sum_{\ell=0}^{j-1} x^\ell S_\ell.$$

The complexity of that scheme is almost the same (see Ex. 4.13.10). With $d = 12$ — $j = 3$ and $k = 4$ — this would give $P_0(y) = a_0 + a_3 y + a_6 y^2 + a_9 y^3$, $P_1(y) = a_1 + a_4 y + a_7 y^2 + a_{10} y^3$, $P_2(y) = a_2 + a_5 y + a_8 y^2 + a_{11} y^3$. We first compute $y = x^3$, y^2 and y^3, then we evaluate $P_0(y)$ in three *scalar* multiplications $a_3 y$, $a_6 y^2$, and $a_9 y^3$ and three additions, similarly for P_1 and P_2, and finally we evaluate $P(x)$ using

$$P(x) = P_0(y) + xP_1(y) + x^2 P_2(y),$$

(here we might use Horner's rule). In this example, we have a total of six nonscalar multiplications: four to compute y and its powers, and two to evaluate $P(x)$.

Complexity of the Rectangular Series Splitting

To evaluate a polynomial $P(x)$ of degree $d-1 = jk-1$, the rectangular series splitting takes $O(j+k)$ nonscalar multiplications — each costing $O(M(n))$ — and $O(jk)$ scalar multiplications. The scalar multiplications involve multiplication and/or division of a multiple-precision number by small integers. Assume that these multiplications and/or divisions take time $c(d)n$ (see Ex. 4.13.11 for a justification of this fact). The constant $c(d)$ accounts for the fact that the involved scalars — the coefficients a_j or the ratios a_{j+1}/a_j — have a size depending on the degree d of $P(x)$. In practice we can safely regard $c(d)$ as constant.

Choosing $j \sim k \sim d^{1/2}$ we get overall time

$$O(d^{1/2}M(n) + dn \cdot c(d)). \tag{4.13}$$

If the degree d of $P(x)$ is of the same order as the precision n of x, this is not an improvement on the bound $O(n^{1/2}M(n))$ that we obtained already by argument reduction and power series evaluation (§4.4.2). However, we can do argument reduction before applying the rectangular series splitting. Applying $\sim n^{1/3}$ steps of argument halving, we can take $d \sim n^{2/3}$ and get overall time

$$O(n^{1/3}M(n) + n^{5/3}c(n)).$$

The Slowly Growing Function $c(n)$

The scalar multiplications involve multiplication and/or division of an n-bit multiple-precision number by "small" integers. Here "small" means $O(d)$, i.e., integers with $O(\log d)$ digits. Suppose that these multiplications and/or divisions take time $c(n)n$ (since $d = O(n)$, we can replace $c(d)$ by $c(n)$). There are three cases:

1. The small integers fit in one word. Then $c(n) = O(1)$ is a constant. This is the case that occurs in practice.

2. If the small integers do not fit in one word, they certainly fit in $O(\log n)$ words, so a straightforward implementation gives $c(n) = O(\log n)$.

3. If we split the n-digit numbers into $O(n/\log n)$ blocks each of $O(\log n)$ bits, and apply $O(n \log^\alpha n)$ multiplication (or division using Newton's method) within each block, we get $c(n) = O(\log^\alpha \log n)$.

REMARK 1. Cases 2 and 3 should never occur in practice, because if n is so large that Case 1 does not apply then we should be using asymptotically faster algorithms (e.g., those based on the AGM) instead of power series evaluation.

We saw that the rectangular series splitting takes time

$$T(n) = O(n^{1/3}M(n) + n^{5/3}c(n)).$$

However this analysis does not take into account the cost of argument reduction, which is $O(\frac{n}{d}M(n))$ to get a polynomial of degree d. The total complexity is thus:

$$T(n) = O(\frac{n}{d}M(n) + d^{1/2}M(n) + dn\,c(n)).$$

Which term dominates? There are two cases:

1. $M(n) \gg n^{4/3}c(n)$. Here the minimum is obtained when the first two terms are equal, i.e., for $d \sim n^{2/3}$, which yields $T(n) = O(n^{1/3}M(n))$. This case applies if we use classical or Karatsuba multiplication, since $\lg 3 > 4/3$, or even Toom-Cook 3-, 4-, 5-, 6-way.

2. $M(n) \ll n^{4/3}c(n)$. Here the minimum is obtained when the first and the last terms are equal. The optimal value of d is then $\sqrt{M(n)/c(n)}$, and we get an improved bound $\Theta(n\sqrt{M(n)c(n)}) \gg n^{3/2}$. We can not approach the $O(n^{1+\varepsilon})$ that is achievable with AGM-based methods, so we probably should not be using the rectangular series splitting (or any method based on power series evaluation) for such large n.

4.5 Asymptotic Expansions

As seen in §4.4, the series (4.11) and (4.12) are not satisfactory for large $|x|$, since they require $\Omega(x^2)$ terms. For example, to evaluate erf(1000) with an accuracy of 6 digits, Eq. (4.11) requires at least $2,718,279$ terms! Instead,

we may use an asymptotic expansion. In the case of the error function, the complementary error function $\operatorname{erfc} x = 1 - \operatorname{erf} x$ satisfies

$$\operatorname{erfc} x \approx \frac{e^{-x^2}}{\sqrt{\pi}x}\left(1 + \sum_{j=1}^{k}(-1)^j \frac{(2j)!}{j!(4x^2)^j}\right), \tag{4.14}$$

with the error bounded in absolute value by the next term and of the same sign. In the $x = 1000$ case, the term $j = 1$ of the sum equals $-0.5 \cdot 10^{-6}$, thus $e^{-x^2}/(\sqrt{\pi}x)$ is an approximation of $\operatorname{erfc} x$ with an accuracy of 6 digits.

For such a function where both a power series for — at $x = 0$ — and an asymptotic expansion — at $x = \infty$ — are available, we might want to use the former or the later, depending on the value of the argument and on the wanted precision. We study here in detail the case of the error function.

Here the sum in (4.14) is divergent — its j-th term behaves roughly like $\frac{j^j}{e^j x^{2j}}$ — thus we need that its smallest term is $O(2^{-n})$ to be able to deduce an n-bit approximation of $\operatorname{erfc} x$. Its minimum is obtained for $j \approx x^2$, and is of order e^{-x^2}, thus we need $x > \sqrt{n \log 2}$. For example for $n = 10^6$ bits this yields $x > 833$. However, since $\operatorname{erfc} x$ is small for large x, says $\operatorname{erfc} x \approx 2^{-k}$, we need only $m = n - k$ correct bits of $\operatorname{erfc} x$ to get n correct bits of $\operatorname{erf} x = 1 - \operatorname{erfc} x$.

```
1 Algorithm Erf.
2 Input: floating-point number x, integer n
3 Output: an approximation of erf(x) to n bits
4 m ← ⌈n − (x^2 + log x + (1/2) log π)/(log 2)⌉
5 If m < n/2 then
6     t ← erfc(x) with (4.14) and precision m
7     Return 1 − t
8 Else compute erf(x) with the power series (4.11)
```

In Algorithm **Erf**, the number of terms needed if Eq. (4.11) is used is the unique positive root j_0 of $j(\log j - 2\log x - 1) = n \log 2$, whereas if Eq. (4.14) is used, it is the smallest j_∞ of the two positive roots of $j(2\log x + 1 - j) = m \log 2$. Fig. 4.1 shows that j_∞ is always smaller than j_0, thus whenever the asymptotic expansion can be used, it should be. (The condition $m < n/2$ in the algorithm comes from $m \approx n - x^2/(\log 2)$ and the inequality $x > \sqrt{m \log 2}$ which ensures m bits from the asymptotic expansion.)

For example for $x = 800$ and a precision of one million bits, Eq. (4.11) requires about $j_0 = 2,339,601$ terms. Eq. (4.14) tells us that $\operatorname{erfc} x \approx 2^{-923335}$, thus we need only $m = 76665$ bits of precision for $\operatorname{erfc} x$; in that case Eq. (4.14) requires only about $j_\infty = 10,375$ terms.

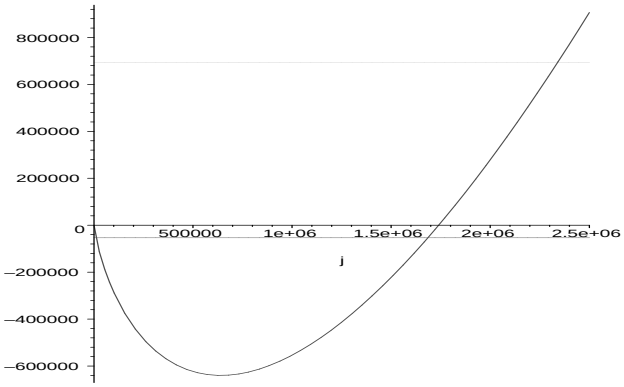


Figure 4.1: Graph of $j(\log j - 2\log x - 1)$ for $0 \le j \le 2,500,000$, $x = 800$, together with the lines $n \log 2$ and $-m \log 2$ for $n = 10^6$ and $m = 76665$.

Another example near the boundary: For $x = 589$, still with $n = 10^6$, we have $m = 499,489$, which gives $j_0 = 1,497,924$, and $j_\infty = 325,092$.

4.6 Continued Fractions

[This section to be completed]

Examples: Ei, erf, Bessel functions. Cf Section 6 of [28], and new book *Handbook of Continued Fractions for Special Functions* [52].

$$\operatorname{Ei} x = \exp x \cfrac{1}{x + \cfrac{1}{1 + \cfrac{1}{x + \cfrac{2}{1 + \cfrac{2}{x + \cfrac{3}{1 + \cdots}}}}}}.$$

$$\operatorname{erf} x = \frac{2xe^{-z^2}}{\sqrt{\pi}} \cfrac{1}{1 - \cfrac{2x^2}{3 + \cfrac{4x^2}{5 - \cfrac{6x^2}{7 + \cfrac{8x^2}{9 - \cfrac{10x^2}{11 + \cdots}}}}}}.$$

4.7 Recurrence Relations

[This section to be completed]

Linear and/or nonlinear recurrence relations (an example of nonlinear is considered in §4.8). Example: Bessel functions (cf Section 7 of [28]):

$$J_{\nu-1}(x) + J_{\nu+1}(x) = \frac{2\nu}{x} J_\nu(x),$$

or for the Bessel-Clifford function[4] $C_n(x) = \sum_{j\geq 0} \frac{x^k}{k!(k+n)!}$ which is related to the Bessel I function by $C_n(x) = x^{-n/2} I_n(2\sqrt{x})$:

$$xC_{n+2}(x) + (n+1)C_{n+1}(x) = C_n(x).$$

This also leads to a continued fraction for $Q_n(x) = C_{n+1}(x)/C_n(x)$.

4.8 Arithmetic-Geometric Mean

The fastest known methods for very large precision n are based on the arithmetic-geometric mean (AGM) iteration of Gauss. The AGM is another nonlinear recurrence, important enough to treat separately. Its complexity is $O(M(n)\log n)$; the implicit constant here can be quite large, so other methods are better for small n.

Given (a_0, b_0), the AGM iteration is defined by

$$(a_{j+1}, b_{j+1}) = \left(\frac{a_j + b_j}{2}, \sqrt{a_j b_j}\right).$$

For simplicity we'll only consider real, positive starting values (a_0, b_0) here (see §3.8).

The AGM iteration converges *quadratically* to a limit which we'll denote by $\mathrm{AGM}(a_0, b_0)$.

Why the AGM is Useful

The AGM is useful because:

[4] `http://en.wikipedia.org/wiki/Bessel-Clifford_function`

1. It converges quadratically. Eventually the number of correct digits doubles at each iteration, so only $O(\log n)$ iterations are required.

2. Each iteration takes time $O(M(n))$ because the square root can be computed in time $O(M(n))$ by Newton's method (see §3.5 and §4.2.3).

3. If we take suitable starting values (a_0, b_0), the result $\mathrm{AGM}(a_0, b_0)$ can be used to compute logarithms (directly) and other elementary functions (less directly), as well as constants such as π and $\log 2$.

4.8.1 Elliptic Integrals

The theory of the AGM iteration is intimately linked to the theory of elliptic integrals.

The *complete elliptic integral of the first kind* is

$$K(k) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - k^2 \sin^2 \theta}} = \int_0^1 \frac{dt}{\sqrt{(1-t^2)(1-k^2t^2)}},$$

and the *complete elliptic integral of the second kind* is

$$E(k) = \int_0^{\pi/2} \sqrt{1 - k^2 \sin^2 \theta}\, d\theta = \int_0^1 \sqrt{\frac{1-k^2t^2}{1-t^2}}\, dt,$$

where $k \in [0, 1]$ is called the *modulus* and $k' = \sqrt{1-k^2}$ is the *complementary modulus*. It is traditional (though confusing) to write $K'(k)$ for $K(k')$ and $E'(k)$ for $E(k')$.

The Connection With Elliptic Integrals

Gauss discovered that

$$\frac{1}{\mathrm{AGM}(1, k)} = \frac{2}{\pi} K'(k). \tag{4.15}$$

This identity can be used to compute the elliptic integral K rapidly via the AGM iteration. We can also use it to compute logarithms.

From the definition

$$K(k) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - k^2 \sin^2 \theta}},$$

we see that $K(k)$ has a series expansion that converges for $|k| < 1$ (in fact $K(k) = \frac{\pi}{2}F(\frac{1}{2}, \frac{1}{2}; 1; k^2)$ is a hypergeometric function). For small k we have

$$K(k) = \frac{\pi}{2}\left(1 + \frac{k^2}{4} + O(k^4)\right). \tag{4.16}$$

It can also be shown [20, (1.3.10)] that

$$K'(k) = \frac{2}{\pi}\ln\left(\frac{4}{k}\right)K(k) - \frac{k^2}{4} + O(k^4). \tag{4.17}$$

4.8.2 First AGM Algorithm for the Logarithm

From these formulæ we easily get

$$\frac{\pi/2}{\mathrm{AGM}(1,k)} = \log\frac{4}{k}(1 + O(k^2)).$$

Thus, if $x = 4/k$ is large, we have

$$\log(x) = \frac{\pi/2}{\mathrm{AGM}(1,4/x)}\left(1 + O\left(\frac{1}{x^2}\right)\right).$$

If $x \geq 2^{n/2}$, we can compute $\log(x)$ to precision n using the AGM iteration. It takes about $2\lg(n)$ iterations to converge if $x \in [2^{n/2}, 2^n]$.

Note that we need the constant π, which could be computed by using our formula twice with slightly different arguments x_1 and x_2, then taking differences. More efficient is to use the Brent-Salamin algorithm, which is based on the AGM and the Legendre relation

$$EK' + E'K - KK' = \frac{\pi}{2}.$$

Argument Expansion

If x is not large enough, we can compute

$$\log(2^k x) = k\,\log 2 + \log x$$

by the AGM method (assuming the constant $\log 2$ is known). Alternatively, if $x > 1$, we can square x enough times and compute

$$\log\left(x^{2^k}\right) = 2^k \log(x).$$

This method with $x = 2$ gives a way of computing $\log 2$.

The Error Term

The $O(k^2)$ error term in the formula

$$\frac{\pi/2}{\mathrm{AGM}(1,k)} = \log\left(\frac{4}{k}\right)(1+O(k^2))$$

is a nuisance. [20, p.11, ex. 4(c)] gives a rigorous bound

$$\left|\frac{\pi/2}{\mathrm{AGM}(1,k)} - \log\left(\frac{4}{k}\right)\right| \le 4k^2(8-\log k)$$

for all $k \in (0,1]$, and the bound can be sharpened to $0.37k^2(2.4-\log(k))$ if $k \in (0,0.5]$.

The error $O(k^2|\log k|)$ makes it difficult to accelerate convergence by using a larger value of k (i.e., a smaller value of $x = 4/k$). There is an *exact* formula which is much more elegant and avoids this problem. Before giving this formula we need to define some *theta functions* and show how they can be used to parameterise the AGM iteration.

4.8.3 Theta Functions

The theta functions that we need are $\theta_2(q)$, $\theta_3(q)$ and $\theta_4(q)$, defined for $|q| < 1$ by

$$\begin{aligned}
\theta_2(q) &= \sum_{n=-\infty}^{+\infty} q^{(n+1/2)^2} = 2q^{1/4}\sum_{n=0}^{+\infty} q^{n(n+1)}, \\
\theta_3(q) &= \sum_{n=-\infty}^{+\infty} q^{n^2} = 1 + 2\sum_{n=1}^{+\infty} q^{n^2}, \qquad (4.18)\\
\theta_4(q) &= \theta_3(-q) = 1 + 2\sum_{n=1}^{+\infty}(-1)^n q^{n^2}. \qquad (4.19)
\end{aligned}$$

Note that the defining power series are sparse so it is easy to compute $\theta_2(q)$ and $\theta_3(q)$ for small q. Unfortunately, the fast method from §4.4.3 does not help to speed up the computation.

The asymptotically fastest methods to compute theta functions use the AGM. However, we won't follow this trail because it would lead us in circles! (We want to use theta functions to give starting values for the AGM iteration.)

Theta Function Identities

There are many identities involving theta functions (see [20, Chapter 2]). Two that are of interest to us are:

$$\frac{\theta_3^2(q) + \theta_4^2(q)}{2} = \theta_3^2(q^2)$$

and

$$\theta_3(q)\theta_4(q) = \theta_4^2(q^2)$$

which may be written as

$$\sqrt{\theta_3^2(q)\theta_4^2(q)} = \theta_4^2(q^2)$$

to show the connection with the AGM:

$$\begin{aligned} & \mathrm{AGM}(\theta_3^2(q)\,, \theta_4^2(q)) = \mathrm{AGM}(\theta_3^2(q^2)\,, \theta_4^2(q^2)) = \cdots \\ = \;& \mathrm{AGM}(\theta_3^2(q^{2^k})\,, \theta_4^2(q^{2^k})) = \cdots = 1 \end{aligned}$$

for any $|q| < 1$. (The limit is 1 because q^{2^k} converges to 0, thus both θ_3 and θ_4 converge to 1.) Apart from scaling, the AGM iteration is parameterised by $(\theta_3^2(q^{2^k}), \theta_4^2(q^{2^k}))$ for $k = 0, 1, 2, \ldots$

The Scaling Factor

Since $\mathrm{AGM}(\theta_3^2(q)\,, \theta_4^2(q)) = 1$, and $\mathrm{AGM}(\lambda a, \lambda b) = \lambda\,\mathrm{AGM}(a, b)$, scaling gives

$$\mathrm{AGM}(1, k') = \frac{1}{\theta_3^2(q)}$$

if

$$k' = \frac{\theta_4^2(q)}{\theta_3^2(q)}.$$

Equivalently, since $\theta_2^4 + \theta_4^4 = \theta_3^4$ (Jacobi),

$$k = \frac{\theta_2^2(q)}{\theta_3^2(q)}.$$

However, we know that

$$\frac{1}{\mathrm{AGM}(1, k')} = \frac{2}{\pi}K(k),$$

so

$$K(k) = \frac{\pi}{2}\theta_3^2(q). \tag{4.20}$$

Thus, the theta functions are closely related to elliptic integrals. In the theory q is usually called the *nome* associated with the modulus k.

From q to k and k to q

We saw that

$$k = \frac{\theta_2^2(q)}{\theta_3^2(q)},$$

which gives k in terms of q. There is also a nice inverse formula which gives q in terms of k:

$$q = \exp(-\pi K'(k)/K(k)),$$

or equivalently

$$\log\left(\frac{1}{q}\right) = \frac{\pi K'(k)}{K(k)}. \tag{4.21}$$

For a proof see [20, §2.3].

Sasaki and Kanada's Formula

Substituting (4.15) and (4.20) with $k = \theta_2^2(q)/\theta_3^2(q)$ in (4.21) gives Sasaki and Kanada's elegant formula:

$$\log\left(\frac{1}{q}\right) = \frac{\pi}{\mathrm{AGM}(\theta_2^2(q)\,, \theta_3^2(q))}. \tag{4.22}$$

4.8.4 Second AGM Algorithm for the Logarithm

Suppose $x \gg 1$. Let $q = 1/x$, compute $\theta_2(q^4)$ and $\theta_3(q^4)$ from their defining series (4.18) and (4.19), then compute $\mathrm{AGM}(\theta_2^2(q^4), \theta_3^2(q^4))$. Sasaki and Kanada's formula (with q replaced by q^4 to avoid the $q^{1/4}$ term in the definition of $\theta_2(q)$) gives

$$\log(x) = \frac{\pi/4}{\mathrm{AGM}(\theta_2^2(q^4)\,, \theta_3^2(q^4))}.$$

There is a trade-off between increasing x (by squaring or multiplication by a power of 2, cf "Argument Expansion" above) and taking longer to compute

$\theta_2(q^4)$ and $\theta_3(q^4)$ from their series. In practice it seems good to increase x so that $q = 1/x$ is small enough that $O(q^{36})$ terms are negligible. Then we can use

$$\theta_2(q^4) = 2\left(q + q^9 + q^{25} + O(q^{49})\right),$$
$$\theta_3(q^4) = 1 + 2\left(q^4 + q^{16} + O(q^{36})\right).$$

We need $x \geq 2^{n/36}$ which is much better than the requirement $x \geq 2^{n/2}$ for the first AGM algorithm. We save about four AGM iterations at the cost of a few multiplications.

Implementation Notes

Since

$$\mathrm{AGM}(\theta_2^2, \theta_3^2) = \frac{\mathrm{AGM}(\theta_2^2 + \theta_3^2, 2\theta_2\theta_3)}{2},$$

we can avoid the first square root in the AGM iteration. Also, it only takes two nonscalar multiplications to compute $2\theta_2\theta_3$ and $\theta_2^2 + \theta_3^2$ from θ_2 and θ_3: $u = (\theta_2 + \theta_3)^2$, $v = \theta_2\theta_3$, then $2v$ and $u - 2v$ are the wanted values.

Constants

Using Bernstein algorithm (see §3.8), an n-bit square root takes time $\frac{11}{6}M(n)$, thus one AGM iteration takes time $\frac{17}{6}M(n)$.

The AGM algorithms require $2\lg(n) + O(1)$ AGM iterations. The total time to compute $\log(x)$ by the AGM is $\sim \frac{17}{3}\lg(n)M(n)$.

Drawbacks of the AGM

1. The AGM iteration is *not* self-correcting, so we have to work with full precision (plus any necessary guard digits) throughout. In contrast, when using Newton's method or evaluating power series, many of the computations can be performed with reduced precision.

2. The AGM with real arguments gives $\log(x)$ directly. To obtain $\exp(x)$ we need to apply Newton's method (§4.2.5). To evaluate trigonometric functions such as $\sin(x)$, $\cos(x)$, $\arctan(x)$ we need to work with complex arguments, which increases the constant hidden in the "O" time bound. Alternatively, we can use Landen transformations for incomplete elliptic integrals, but this gives even larger constants.

3. Because it converges so fast, it is difficult to speed up the AGM. At best we can save $O(1)$ iterations.

4.8.5 The Complex AGM

In some cases the asymptotically fastest algorithms require the use of complex arithmetic to produce a real result. It would be nice to avoid this because complex arithmetic is significantly slower than real arithmetic.

Examples where we seem to need complex arithmetic to get the asymptotically fastest algorithms are:

1. $\arctan(x)$, $\arcsin(x)$, $\arccos(x)$ via the AGM using e.g.,

$$\arctan(x) = \Im(\log(1 + ix)).$$

2. $\tan(x)$, $\sin(x)$, $\cos(x)$ using Newton's method and the above, or

$$\cos(x) + i\sin(x) = \exp(ix),$$

   where the complex exponential is computed by Newton's method from the complex logarithm, similarly as in §4.2.5 for the real case.

The theory that we outlined for the AGM iteration and AGM algorithms for $\log(z)$ can be extended without problems to complex $z \notin (-\infty, 0]$, provided we always choose the square root with positive real part.

A complex multiplication takes three real multiplications (using Karatsuba's trick), and a complex squaring takes two real multiplications. One can ever do better in the FFT domain, if one assumes that one multiplication of cost $M(n)$ is equivalent to three Fourier transforms. In that model a squaring costs $\frac{2}{3}M(n)$. A complex multiplication $(a+ib)(c+id) = (ac-bd)+i(ad+bc)$ requires four forward and two backward transforms, thus costs $2M(n)$. A complex squaring $(a+ib)^2 = (a+b)(a-b)+i(2ab)$ requires two forward and two backward transforms, thus costs $\frac{4}{3}M(n)$. Taking this into account, we get the following asymptotic upper bounds (0.666 should read $0.666\ldots$, and

so on):

Operation	real	complex
squaring	$0.666M(n)$	$1.333M(n)$
multiplication	$M(n)$	$2M(n)$
division	$2.0833M(n)$	$6.5M(n)$
square root	$1.8333M(n)$	$6.333M(n)$
AGM iteration	$2.8333M(n)$	$8.333M(n)$
log via AGM	$5.666\lg(n)M(n)$	$16.666\lg(n)M(n)$

(4.23)

See §4.14 for details about the algorithms giving those constants.

4.9 Binary Splitting

Since the asymptotically fastest algorithms for arctan, sin, cos etc have a large constant hidden in their time bound $O(M(n)\log n)$ (see paragraph "Drawbacks of the AGM" in §4.8.4), it is interesting to look for other algorithms that may be competitive for a large range of precisions even if not asymptotically optimal. One such algorithm (or class of algorithms) is based on *binary splitting* or the closely related *FEE method* (see §4.14). The time complexity of these algorithms is usually

$$O((\log n)^\alpha M(n))$$

for some constant $\alpha \geq 1$ depending on how fast the relevant power series converges, and also on the multiplication algorithm (classical, Karatsuba or quasi-linear).

The Idea

Suppose we want to compute $\arctan(x)$ for rational $x = p/q$, where p and q are small integers and $|x| \leq 1/2$. The Taylor series gives

$$\arctan\left(\frac{p}{q}\right) \approx \sum_{0\leq j\leq n/2} \frac{(-1)^j p^{2j+1}}{(2j+1)q^{2j+1}}.$$

The finite sum, if computed exactly, gives a rational approximation P/Q to $\arctan(p/q)$, and

$$\log|Q| = O(n\log n).$$

(Note: the series for exp converges faster, so in this case $\log|Q| = O(n)$.)

The finite sum can be computed by "divide and conquer": sum the first half to get P_1/Q_1 say, and the second half to get P_2/Q_2, then

$$\frac{P}{Q} = \frac{P_1}{Q_1} + \frac{P_2}{Q_2} = \frac{P_1Q_2 + P_2Q_1}{Q_1Q_2}.$$

The rationals P_1/Q_1 and P_2/Q_2 are computed by a recursive application of the same method, hence the term "binary splitting". If used with quadratic multiplication, this way of computing P/Q does not help; however, fast multiplication speeds up the balanced products P_1Q_2, P_2Q_1, and Q_1Q_2.

Complexity

The overall time complexity is

$$O\left(\sum_j M(\frac{n}{2^k})\log\frac{n}{2^k})\right) = O((\log n)^\alpha M(n)),$$

where $\alpha = 2$ in the FFT range; multiplication; in general $\alpha \le 2$.

We can save a little by working to precision n rather than $n\log n$ at the top levels; for classical or Karatsuba multiplication this reduces α to 1, but we still have $\alpha = 2$ for quasi-linear multiplication.

In practice the multiplication algorithm would not be fixed but would depend on the size of the integers being multiplied. The complexity depends on the algorithm that is used at the top levels.

Repeated Application of the Idea

If $x \in (0, 0.5)$ and we want to compute $\arctan(x)$, we can approximate x by a rational p/q and compute $\arctan(p/q)$ as a first approximation to $\arctan(x)$, say $p/q \le x < (p+1)/q$. Now

$$\tan(\arctan(x) - \arctan(p/q)) = \frac{x - p/q}{1 + px/q},$$

so

$$\arctan(x) = \arctan(p/q) + \arctan(\delta)$$

where

$$\delta = \frac{x - p/q}{1 + px/q} = \frac{qx - p}{q + px}.$$

We can apply the same idea to approximate $\arctan(\delta)$, until eventually we get a sufficiently accurate approximation to $\arctan(x)$. Note that $|\delta| < |x - p/q|$, so it is easy to ensure that the process converges.

Complexity of Repeated Application

If we use a sequence of about $\lg n$ rationals $p_1/q_1, p_2/q_2, \ldots$, where

$$q_i = 2^{2^i},$$

then the computation of each $\arctan(p_i/q_i)$ takes time $O((\log n)^\alpha M(n))$ and the overall time to compute $\arctan(x)$ is

$$O((\log n)^{\alpha+1} M(n)).$$

Indeed, we have $0 \le p_i < 2^{2^{i-1}}$, thus p_i has at most 2^{i-1} bits, and p_i/q_i as a rational has value $O(2^{2^{-i}})$ and size $O(2^i)$. The exponent $\alpha + 1$ is 2 or 3. Although this is not asymptotically as fast as AGM-based algorithms, the implicit constants for binary splitting are small and the idea is useful for quite large n (at least 10^6 decimal places).

Generalisations

The idea of binary splitting can be generalised. For example, the Chudnovsky brothers gave a "bit-burst" algorithm which applies to fast evaluation of solutions of linear differential equations. This is described in §4.9.2

4.9.1 A Binary Splitting Algorithm for $\sin, \cos$

In [25, Theorem 6.2], Brent claims an $O(M(n) \log^2 n)$ algorithm for $\exp x$ and $\sin x$, however the proof only details the case of the exponential, and ends by "the proof of (6.28) is similar". Most probably the author had in mind deducing $\sin x$ from a complex computation of $\exp(ix) = \cos x + i \sin x$. Algorithm **SinCos** is a variation of Brent's algorithm for $\exp x$ that computes simultaneously $\sin x$ and $\cos x$, and in such a way avoids computations with complex numbers. The simultaneous computation of $\sin x$ and $\cos x$ might

1 Algorithm **SinCos**.
2 **Input**: floating-point $|x| < 1/2$, integer n
3 **Output**: an approximation of $\sin x$ and $\cos x$ with error $O(2^{-n})$
4 Write $x = \sum_{i=0}^{k} p_i \cdot 2^{-2^{i+1}}$ where $0 \le p_i < 2^{2^i}$ and $k = \lceil \lg n \rceil - 1$
5 Let $x_j = \sum_{i=j}^{k} p_i \cdot 2^{-2^{i+1}}$ and $y_i = p_i \cdot 2^{-2^{i+1}}$
6 $(S_{k+1}, C_{k+1}) \leftarrow (0, 1)$
7 **For** $j = k, k-1, \ldots, 1, 0$
8 Compute $\sin y_i$ and $\cos y_i$ using binary splitting
9 $S_j \leftarrow \sin y_i C_{j+1} + \cos y_i S_{j+1}$, $C_j \leftarrow \cos y_i C_{j+1} + \sin y_i S_{j+1}$
10 Return (S_0, C_0)

be useful, for example to compute $\tan x$. At step 5 of Algorithm **SinCos**, we have $x_j = y_i + x_{j+1}$, thus $\sin x_j = \sin y_i \cos x_{j+1} + \cos y_i \sin x_{j+1}$, and similarly for $\cos x_j$, which are the formulæ used at step 9. Step 8 uses the binary splitting algorithm described above for $\arctan(p/q)$: y_i is a small rational, or is small itself, so that all needed powers do not exceed n bits in size. This algorithm has the same complexity $O(M(n) \log^2 n)$ as Brent's algorithm for $\exp x$.

4.9.2 The Bit-Burst Algorithm

The binary-splitting algorithms described above for $\arctan x$, $\exp x$, $\sin x$ rely on a functional equation: $\tan(x+y) = (x+y)/(1-xy)$, $\exp(x+y) = \exp(x)\exp(y)$, $\sin(x+y) = \sin x \cos y + \sin y \cos x$. We describe here a more general algorithm, called "bit-burst", which does not require such a functional equation. This algorithm applies to the so-called "D-finite" functions.

A function $f(x)$ is said to be *D-finite* — or *holonomic* — iff it satisfies a linear differential equation with polynomial coefficients in x. Equivalently, the Taylor coefficients u_k of f satisfy a linear recurrence with polynomial coefficients in k. For example, the $\exp, \log, \sin, \cos$ functions are D-finite, but $\tan$ is not. An important subclass of D-finite functions are the hypergeometric functions, whose Taylor coefficients satisfy an homogeneous recurrence of order 1: $u_{k+1}/u_k = R(k)$ where $R(k)$ is a rational function of k (see §4.4). However, D-finite functions are much more general than hypergeometric functions; in particular the ratio of two consecutive terms in a hypergeometric series has size $O(\log k)$ — as a rational —, but might be

much larger for D-finite functions (see Ex. 4.13.14).

Theorem 4.9.1 *If f is D-finite and has no singularities on a finite, closed interval $[A, B]$, where $A < 0 < B$ and $f(0) = 0$, then $f(x)$ can be computed to an accuracy of n bits for any n-bit floating-point number $x \in [A, B]$ in time $O(M(n) \log^3 n)$.*

NOTE: the condition $f(0) = 0$ is just a technical condition to simplify the proof of the theorem; $f(0)$ can be any value that can be computed to n bits in time $O(M(n) \log^3 n)$.

Proof. Without loss of generality, we assume $0 \leq x < 1 < B$; the binary expansion of x can then be written $x = 0.b_1 b_2 \ldots b_n$. Define $r_1 = 0.b_1$, $r_2 = 0.0b_2 b_3$, $r_3 = 0.000b_4 b_5 b_6 b_7$ (the same decomposition was already used in §4.9.1): r_1 consists of the first bit of the binary expansion of x, r_2 consists of the next two bits, r_3 the next four bits, and so on. We thus have $x = r_1 + r_2 + \ldots + r_k$ where $2^{k-1} \leq n < 2^k$.

Define $x_i = r_1 + \cdots + r_i$ with $x_0 = 0$. The idea of the algorithm is to translate the Taylor series of f from x_i to x_{i+1}, which since f is D-finite reduces to translating the recurrence. The condition that f has no singularity in $[0, x] \subset [A, B]$ ensures that the translated recurrence is well-defined. We define $f_0(t) = f(t)$, $f_1(t) = f_0(r_1+t)$, $f_2(t) = f_1(r_2+t)$, $\ldots$, $f_i(t) = f_{i-1}(r_i+t)$ for $i \leq k$. We have $f_i(t) = f(x_i+t)$, and $f_k(t) = f(x+t)$ since $x_k = x$. Thus we are looking for $f_k(0) = f(x)$.

Let $f_i^*(t) = f_i(t) - f_i(0)$ be the non-constant part of the Taylor expansion of f_i. We have $f_i^*(r_{i+1}) = f_i(r_{i+1}) - f_i(0) = f_{i+1}(0) - f_i(0)$ since $f_{i+1}(t) = f_i(r_{i+1} + t)$. Thus $f_0^*(r_1) + \cdots + f_{k-1}^*(r_k) = (f_1(0) - f_0(0)) + \cdots + (f_k(0) - f_{k-1}(0)) = f_k(0) - f_0(0) = f(x) - f(0)$. Since $f(0) = 0$, this gives:

$$f(x) = \sum_{i=0}^{k-1} f_i^*(r_{i+1}).$$

To conclude the proof, we will show that each term $f_i^*(r_{i+1})$ can be evaluated to n bits in $O(M(n) \log^2 n)$.

The rational r_{i+1} has numerator of at most 2^i bits, and

$$0 \leq r_{i+1} < 2^{1-2^i}.$$

Thus, to evaluate $f_i^*(r_{i+1})$ to n bits, $n/2^i + O(\log n)$ terms of the Taylor expansion of $f_i^*(t)$ are enough.

We now use the fact that f is D-finite. Assume f satisfies the following linear differential equation with polynomial coefficients[5]:

$$c_m(t)f^{(m)}(t) + \cdots + c_1(t)f'(t) + c_0(t)f(t) = 0.$$

Substituting $x_i + t$ for t, we obtain a differential equation for f_i:

$$c_m(x_i + t)f_i^{(m)}(t) + \cdots + c_1(x_i + t)f_i'(t) + c_0(x_i + t)f_i(t) = 0.$$

From this latter equation we deduce (see §4.14) a linear recurrence for the Taylor coefficients of $f_i(t)$, of the same order than that for $f(t)$. The coefficients in the recurrence for $f_i(t)$ have $O(2^i)$ bits, since $x_i = r_1 + \cdots + r_i$ has $O(2^i)$ bits. It follows that the ℓ-th Taylor coefficient from $f_i(t)$ has size $O(\ell(2^i + \log \ell))$ — the $\ell \log \ell$ term comes from the polynomials in ℓ in the recurrence]. Since ℓ goes to $n/2^i + O(\log n)$ at most, this is $O(n \log n)$.

However we do not want to evaluate the ℓ-th Taylor coefficient u_ℓ of $f_i(t)$, but the series

$$s_\ell = \sum_{j=1}^{\ell} u_j r_{i+1}^j.$$

Noticing that $u_\ell = (s_\ell - s_{\ell-1})/r_{i+1}^\ell$, and substituting that value in the recurrence for (u_ℓ), say of order d, we obtain a recurrence of order $d + 1$ for (s_ℓ). Putting this latter recurrence in matrix form $S_\ell = M_\ell S_{\ell-1}$, where S_ℓ is the vector $(s_\ell, s_{\ell-1}, s_{\ell-d+1})$, we obtain

$$S_\ell = M_\ell M_{\ell-1} \cdots M_d S_{d-1}, \tag{4.24}$$

where the matrix product $M_\ell M_{\ell-1} \cdots M_d$ can be evaluated in $O(M(n) \log^2 n)$ using binary splitting. □

We illustrate the above theorem with the arc-tangent function, which satisfies the differential equation:

$$f'(t)(1 + t^2) = 1.$$

This equation evaluates at $x_i + t$ into $f_i'(t)(1 + (x_i + t)^2) = 1$ for $f_i(t) = f(x_i + t)$, which gives the recurrence

$$(1 + x_i^2)\ell u_\ell + 2x_i(\ell - 1)u_{\ell-1} + (\ell - 2)u_{\ell-2} = 0$$

[5]If f satisfies a non-homogeneous differential equation, say $E(t, f(t), f'(t), \ldots) = b(t)$, where $b(t)$ is polynomial in t, a differentiation of $(\deg b) + 1$ times yields an homogeneous equation.

for the Taylor coefficients u_ℓ of f_i. This recurrence translates to

$$(1+x_i^2)\ell v_\ell + 2x_i r_{i+1}(\ell-1)v_{\ell-1} + r_{i+1}^2(\ell-2)v_{\ell-2} = 0$$

for $v_\ell = u_\ell r_{i+1}^\ell$, and to

$$(1+x_i^2)\ell(s_\ell - s_{\ell-1}) + 2x_i r_{i+1}(\ell-1)(s_{\ell-1} - s_{\ell-2}) + r_{i+1}^2(\ell-2)(s_{\ell-2} - s_{\ell-3}) = 0$$

for $s_\ell = \sum_{j=1}^{\ell} v_j$. This recurrence of order 3 is then put in matrix form, and Eq. (4.24) enables one to efficiently compute $s_\ell \approx f_i(r_i+1) - f_i(0)$ using the multiplication of 3×3 matrices and fast integer multiplication.

4.10 Contour Integration

[This section to be completed]

Example (Bernoulli numbers):

$$\frac{z}{e^z - 1} + \frac{z}{2}.$$

4.11 Other Special Functions

[This section to be completed]

$\Gamma(x)$, $\Psi(x)$, $\zeta(x)$, $\zeta(\frac{1}{2}+iy)$, etc. Bombieri conjecture re evaluation of $\zeta(s)$ on critical line. Borwein algorithm for $\zeta(s)$ with small $\Im(s)$. (reference?)

4.12 Constants

[This section to be completed]

Ex: exp, π, γ [36], etc. Cf `http://cr.yp.to/1987/bernstein.html` for π and e. Cf also [62].

4.13 Exercises

Exercise 4.13.1 If $A(x) = \sum_{j\geq 0} a_j x^j$ is a formal power series over $\mathbb{R}$ with $a_0 = 1$, show that $\log(A(x))$ can be computed with error $O(x^n)$ in time $O(M(n))$, where $M(n)$ is the time required to multiply two polynomials of degree $n-1$. (Assume a

reasonable smoothness condition on the growth of $M(n)$ as a function of n.) [Hint: $(d/dx)\log(A(x)) = A'(x)/A(x)$.] Does a similar result hold for n-bit numbers if x is replaced by $1/2$?

Exercise 4.13.2 (Schost) Assume one wants to compute $1/s(x) \bmod x^n$, for $s(x)$ a power series. Design an algorithm using an odd-even scheme (§1.3.5), and estimate its complexity in the FFT range.

Exercise 4.13.3 Design a Horner-like algorithm evaluating a series $\sum_{j\geq 0}^{k} a_j x^j$ in the forward direction.

Exercise 4.13.4 Assume one wants n bits of $\exp x$ for x of order 2^j, with the repeated use of the doubling formula (§4.3.1), and the naive method to evaluate power series. What is the best reduced argument $x/2^k$ in terms of n and j? [Consider both cases $j > 0$ and $j < 0$.]

Exercise 4.13.5 Assuming one can compute n bits of $\log x$ in time $T(n) \gg M(n) \gg n$, where $M(n)$ satisfies a reasonable smoothness condition, show how to compute $\exp x$ in time $\sim T(n)$.

Exercise 4.13.6 Care has to be taken to use enough guard digits when computing $\exp(x)$ by argument reduction followed by the power series (4.10). If x is of order unity and k steps of argument reduction are used to compute $\exp(x)$ via

$$\exp(x) = \left(\exp(x/2^k)\right)^{2^k},$$

show that about k bits of precision will be lost (so it is necessary to use about k guard bits).

Exercise 4.13.7 Show that the problem analysed in Ex. 4.13.6 can be avoided if we work with the function

$$\mathrm{expm1}(x) = \exp(x) - 1 = \sum_{1}^{\infty} \frac{x^j}{j!}$$

which satisfies the doubling formula

$$\mathrm{expm1}(2x) = \mathrm{expm1}(x)(2 + \mathrm{expm1}(x)).$$

Exercise 4.13.8 Prove the reduction formula

$$\mathrm{log1p}(x) = 2\,\mathrm{log1p}\left(\frac{x}{1+\sqrt{1+x}}\right)$$

where the function $\mathrm{log1p}(x)$ is defined by

$$\mathrm{log1p}(x) = \log(1+x).$$

Explain why it might be desirable to work with log1p instead of log in order to avoid loss of precision (here in the argument reduction rather than in the reconstruction like in Ex. 4.13.6). Note however that argument reduction for log1p is more expensive than that for expm1, because of the square root.

Exercise 4.13.9 (White) Show that $\exp(x)$ can be computed via $\sinh(x)$ using the formula

$$\exp(x) = \sinh(x) + \sqrt{1+\sinh^2(x)}.$$

Since

$$\sinh(x) = \frac{e^x - e^{-x}}{2} = \sum_{k\geq 0} \frac{x^{2k+1}}{(2k+1)!}$$

this saves computing about half the terms in the power series for $\exp(x)$ (at the expense of one square root). How would you modify this method for negative arguments x?

Exercise 4.13.10 Count precisely the number of nonscalar products necessary for the two variants of the rectangular series splitting (§4.4.3).

Exercise 4.13.11 A drawback of the rectangular series splitting as presented in §4.4.3 is that the coefficients — $a_{k\ell+m}$ in the classical splitting, or $a_{jm+\ell}$ in the modular splitting — involved in the scalar multiplications might become large. Indeed, they are typically a product of factorials, and thus have size $O(d \log d)$. Assuming a_{i+1}/a_i are small rationals, propose an alternate way of evaluating $P(x)$.

Exercise 4.13.12 Deduce from Eq. (4.16) and (4.17) an expansion of $\log(4/k)$ with error term $O(k^4 \log(4/k))$. Use any means to figure out an effective bound on the $O()$ term. Deduce an algorithm requiring $x \geq 2^{n/4}$ only to get n bits of $\log x$.

Exercise 4.13.13 Improve the constants at the end of §4.8.5.

Exercise 4.13.14 (Salvy) Is the function $\exp(x) + x/(1-x^2)$ D-finite?

4.14 Notes and References

One of the main references for special functions is the "Handbook of Mathematical Functions" by Abramowitz & Stegun [1]. Another more recent book is that of Nico Temme [124]. A large part of the content of this Chapter comes from [28], and was already implemented in the MP package [27].

Some details about the use of Newton's method in modern processors can be found in [72]. The idea of first computing $y^{-1/2}$, then multiplying by y to get $y^{1/2}$ (§4.2.3) was pushed further by Karp and Markstein [79], who perform this at the penultimate iteration, and modify the last iteration of Newton's method for $y^{-1/2}$ to directly get $y^{1/2}$ (see §1.4.5 for an example of the Karp-Markstein trick for the division). For more on Newton's method for power series, we refer to [23, 34, 38, 81].

The rectangular series splitting to evaluate a power series with $O(\sqrt{n})$ nonscalar multiplications (§4.4.3) was first published by Paterson and Stockmeyer in 1973 [105]. It was then rediscovered in the context of multiple-precision evaluation of elementary functions by Smith in 1991 [118, §8.7], who gave the name "concurrent series" (note that Smith proposed the modular splitting of the series, whereas the classical splitting seems slightly better). Smith already noticed that the simultaneous use of this fast technique and argument reduction yields $O(n^{1/3}M(n))$ algorithms. Earlier, Estrin had found in 1960 a similar technique with $n/2$ nonscalar multiplications, but $O(\log n)$ parallel complexity [56].

Formula (4.14) is from [1], formulæ 7.1.23 and 7.1.24.

Some references about the Arithmetic-Geometric Mean (AGM) are Brent [23, 26, 31], the Borweins' book [20], Arndt and Haenel [5], and Bernstein [8], who gives a survey of the different AGM algorithms from the literature to compute the logarithm. For an extension of the AGM to complex starting values, see Borwein & Borwein [20, pp. 15–16]. The use of the exact formula (4.22) to compute $\log x$ was first suggested by Sasaki and Kanada (see [20, (7.2.5)], but beware the typo). See [26] about Landen transformations, and [23] about more efficient methods; note that the constants given in those papers might be improved using faster square root algorithms (Chapter 3). The use of the complex AGM is discussed in [54].

The constants from (4.8.5) are obtained as follows. We assume we are in the FFT domain, and one Fourier transform costs $\frac{1}{3}M(n)$. The $2.0833M(n)$ cost for real division is from [67]. The complex division uses the "faster" algorithm from [23, Section 11] which computes $\frac{t+iu}{v+iw}$ as $\frac{(t+iu)(v-iw)}{v^2+w^2}$, with one

complex multiplication for $(t+iu)(v-iw)$, two squarings for v^2 and w^2, one reciprocal, and two real multiplications; noting that v^2+w^2 can be computed in $M(n)$ by sharing the backward transform, and using Schönhage's $1.5M(n)$ algorithm for the reciprocal, we get a total cost of $6.5M(n)$. The $1.8333M(n)$ cost for the real square root is from [9]. The complex square root uses Friedland's algorithm [58]: $\sqrt{x+iy} = w + iy/(2w)$ where $w = \sqrt{(|x|+\sqrt{x^2+y^2})/2}$; as for the complex division, x^2+y^2 costs $M(n)$, then we compute its square root in $1.8333M(n)$, and we use Bernstein's $2.5M(n)$ algorithm [9] to compute simultaneously $w^{1/2}$ and $w^{-1/2}$, which we multiply by y in $M(n)$, which gives a total of $6.333M(n)$. The cost of one AGM iteration is simply the sum of the multiplication cost and of the square root cost, while the logarithm via the AGM costs $2\lg(n)$ AGM iterations.

There is quite a controversy in the literature about "binary splitting" and the "FEE method" of E. A. Karatsuba [78]. We choose the name "binary splitting" because it is more descriptive, and let the reader replace it by "FEE method" if he/she prefers. Whatever its name, the idea is quite old, since Brent in [25, Theorem 6.2] gave in 1976 a binary splitting algorithm to compute $\exp x$ in $O(M(n)\log^2 n)$. The CLN library implements several functions with binary splitting [63], and is thus quite efficient for precisions of a million bits or more. The "bit-burst" algorithm was invented by David and Gregory Chudnovsky [42].

For more about D-finite functions, see for example the Maple GFUN package [109], which allows — among other things — to deduce the recurrence of the Taylor coefficients of $f(x)$ from its differential equation.

Chapter 5

Appendix: Implementations and Pointers

5.1 Software Tools

5.1.1 CLN

CLN (Class Library for Numbers, `http://www.ginac.de/CLN/`) is a library for efficient computations with all kinds of numbers in arbitrary precision. It was written by Bruno Haible, and is currently maintained by Richard Kreckel. It is written in C++ and distributed under the GNU General Public License (GPL). CLN provides some elementary and special functions, and fast arithmetic on large numbers, in particular it implements Schönhage-Strassen multiplication, and the binary splitting algorithm [63]. CLN can be configured to use GMP low-level MPN routines, which "is known to be quite a boost for CLN's performance".

5.1.2 GNU MP

The GNU MP library is the main reference for arbitrary-precision arithmetic. It has been developed by Torbjörn Granlund and other contributors since 1993 (the first public version, GMP 1.3.2, was released in May 1993, and indicates that work on GMP started in 1991). GNU MP (GMP for short) implements several of the algorithms described in this book. In particular, we recommend reading Chapter Algorithms of the GMP reference manual

[61]. GMP is written in the C language, is released under the GNU Lesser General Public License (LGPL), and is available from `gmplib.org`.

GMP's MPZ class implements arbitrary-precision integers (corresponding to Ch. 1), while the MPF class implements arbitrary-precision floating-point numbers (corresponding to Ch. 3). The performance of GMP comes mostly from its low-level MPN class, which is well designed and highly optimized in assembly code for many architectures. As of version 4.2.2, MPZ implements different multiplication algorithms (schoolbook, Karatsuba, Toom-Cook 3-way, and Schönhage-Strassen); however its division routine implements Algorithm **RecursiveDivRem** (§1.4.3) and has thus complexity $O(M(n)\log n)$ instead of $O(M(n))$, and so does its square root, which implements Algorithm **SqrtRem**, since it relies on division. GMP 4.2.2 does not implement elementary nor special functions (Ch. 4), and neither provides modular arithmetic with invariant divisor (Ch. 2).

5.1.3 MPFR

[This section to be completed]

5.1.4 ZEN

[This section to be completed]

5.2 Mailing Lists

5.2.1 The BNIS Mailing List

The BNIS mailing-list was created by Dan Bernstein for "Anything of interest to implementors of large-integer arithmetic packages". It has low traffic (a few messages per year only). See `http://cr.yp.to/lists.html` to subscribe. An archive of this list is available at `http://www.nabble.com/cr.yp.to---bnis-f846.html`.

5.2.2 The GMP Lists

There are four mailing-lists associated to GMP: `gmp-bugs` for bug reports; `gmp-announce` for important announcements about GMP, in particular new

releases; `gmp-discuss` for general discussions about GMP; `gmp-devel` for technical discussions between GMP developers. We recommend subscription to `gmp-announce` (very low traffic) and to `gmp-discuss` (medium to high traffic), and to `gmp-devel` if you are interested about the internals of GMP.

5.3 On-Line Documents

The Wolfram Functions Site (`http://functions.wolfram.com/`) contains a lot of information about mathematical functions (definition, specific values, general characteristics, representations as series, limits, integrals, continued fractions, differential equations, transformations, and so on).

The Encyclopedia of Special Functions (`http://algo.inria.fr/esf/`) is another nice web site, whose originality is that all formulæ are automatically generated from very few data that uniquely define the corresponding function in a general class [95].

A huge set of informations about interval arithmetic can be found on the Interval Computations page (`http://www.cs.utep.edu/interval-comp/`) (introduction, software, languages, books, courses, information about the interval arithmetic community, applications).

Mike Cowlishaw maintains an extensive bibliography of conversion to and from decimal arithmetic at `http://www2.hursley.ibm.com/decimal/decbibconversion.html`.

Bibliography

[1] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions*. Dover, 1973. [167]

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. [50, 78]

[3] ANSI/IEEE. IEEE standard for binary floating-point arithmetic. Technical Report ANSI-IEEE Standard 754-1985, ANSI/IEEE, New York, 1985. [124]

[4] Jörg Arndt. *Algorithms for programmers*. 2008. Book in preparation. 970 pages as of 17 April 2008. [79]

[5] Jörg Arndt and Christoph Haenel. π *Unleashed*. Springer-Verlag, 2001. Review by Carl D. Mueller on `http://www.maa.org/reviews/piunleashed.html`. [167]

[6] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology, Proceedings of Crypto'86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer-Verlag, 1987. [55]

[7] Dan J. Bernstein. Detecting perfect powers in essentially linear time. *Mathematics of Computation*, 67:1253–1283, 1998. [49]

[8] Dan J. Bernstein. Computing logarithm intervals with the arithmetic-geometric-mean iteration. `http://cr.yp.to/arith.html`, 2003. 8 pages. [167]

[9] Dan J. Bernstein. Removing redundancy in high-precision Newton iteration. `http://cr.yp.to/fastnewton.html`, 2004. 13 pages. [125, 168]

[10] Dan J. Bernstein. Fast multiplication and its applications. `http://cr.yp.to/arith.html`, 2008. 60 pages. [79]

[11] Daniel J. Bernstein. Multidigit modular multiplication with the explicit Chinese remainder theorem, 1995. `http://cr.yp.to/papers/mmecrt.pdf`, 7 pages. [78]

[12] Daniel J. Bernstein. Pippenger's exponentiation algorithm. `http://cr.yp.to/papers.html`, 2002. 21 pages. [79]

[13] Daniel J. Bernstein, Hendrik W. Lenstra, Jr., and Jonathan Pila. Detecting perfect powers by factoring into coprimes. *Mathematics of Computation*, 76:385–388, 2007. [49]

[14] Daniel J. Bernstein and Jonathan P. Sorenson. Modular exponentiation via the explicit Chinese remainder theorem. *Mathematics of Computation*, 76(257):443–454, January 2007. [79]

[15] R. Bernstein. Multiplication by integer constants. *Software, Practice and Experience*, 16(7):641–652, 1986. [21]

[16] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29:225–252, 2002. Special Issue on Automating and Mechanising Mathematics: In honour of N.G. de Bruijn. [49]

[17] Marco Bodrato and Alberto Zanoni. Integer and polynomial multiplication: Towards optimal Toom-Cook matrices. In C. W. Brown, editor, *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation, ISSAC'2007, July 29-August 1st, 2007*, pages 17–24, Waterloo, Ontario, Canada, 2007. ACM. [49]

[18] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998. [49]

[19] Allan Borodin and Ian Munro. *The Computational Complexity of Algebraic and Numeric Problems*. Elsevier Computer Science Library, 1975. [79]

[20] J. M. Borwein and P. B. Borwein. *Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity*. Wiley, 1998. [152, 153, 154, 155, 167]

[21] Alin Bostan, Grégoire Lecerf, and Éric Schost. Tellegen's principle into practice. In *Proceedings of the 2003 international symposium on Symbolic and algebraic computation*, pages 37–44, Philadelphia, PA, USA, 2003. [125]

[22] Richard P. Brent. On the precision attainable with various floating-point number systems. *IEEE Transactions on Computers*, C-22:601–607, 1973. `http://wwwmaths.anu.edu.au/~brent/pub/pub017.html`. [122, 124]

[23] Richard P. Brent. Multiple-precision zero-finding methods and the complexity of elementary function evaluation. In J. F. Traub, editor, *Analytic Computational Complexity*, pages 151–176, New York, 1975. Academic Press. `http://wwwmaths.anu.edu.au/~brent/pub/pub028.html`. [167]

[24] Richard P. Brent. Analysis of the binary Euclidean algorithm. In J. F. Traub, editor, *New Directions and Recent Results in Algorithms and Complexity*, pages 321–355. Academic Press, New York, 1976. `http://wwwmaths.anu.edu.au/~brent/pub/pub037.html`. Errata: see the online version. [49]

[25] Richard P. Brent. The complexity of multiple-precision arithmetic. In R. S. Anderssen and Richard P. Brent, editors, *The Complexity of Computational Problem Solving*, pages 126–165. University of Queensland Press, 1976. `http://wwwmaths.anu.edu.au/~brent/pub/pub032.html`. [160, 168]

[26] Richard P. Brent. Fast multiple-precision evaluation of elementary functions. *Journal of the ACM*, 23(2):242–251, 1976. `http://wwwmaths.anu.edu.au/~brent/pub/pub034.html`. [167]

[27] Richard P. Brent. Algorithm 524: MP, a Fortran multiple-precision arithmetic package. *toms*, 4:71–81, 1978. [167]

[28] Richard P. Brent. Unrestricted algorithms for elementary and special functions. In S. H. Lavington, editor, *Information Processing*, volume 80, pages 613–619, 1980. `http://wwwmaths.anu.edu.au/~brent/pub/pub052.html`. [149, 150, 167]

[29] Richard P. Brent. An idealist's view of semantics for integer and real types. *Australian Computer Science Communications*, 4:130–140, 1982. `http://wwwmaths.anu.edu.au/~brent/pub/pub069.html`. [124]

[30] Richard P. Brent. Twenty years' analysis of the binary Euclidean algorithm. In A. W. Roscoe J. Davies and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 41–53. Palgrave, New York, 2000. `http://wwwmaths.anu.edu.au/~brent/pub/pub183.html`. [49]

[31] Richard P. Brent. Fast algorithms for high-precision computation of elementary functions. `http://wwwmaths.anu.edu.au/~brent/talks.html`, 2006. Invited talk presented at the Real Numbers and Computation Conference (RNC7), Nancy, France, July 2006. [167]

[32] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. The Software gf2x. http://wwwmaths.anu.edu.au/~brent/gf2x.html. [79]

[33] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster multiplication in GF(2)[x]. In A. J. van der Poorten and A. Stein, editors, *Proceedings of the 8th International Symposium on Algorithmic Number Theory (ANTS VIII)*, volume 5011 of *Lecture Notes in Computer Science*, pages 153–166. Springer-Verlag, 2008. [79]

[34] Richard P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. *Journal of the ACM*, 25(2):581–595, 1978. http://wwwmaths.anu.edu.au/~brent/pub/pub045.html. [167]

[35] Richard P. Brent and H. T. Kung. Systolic VLSI arrays for linear-time GCD computation. In F. Anceau and E. J. Aas, editors, *VLSI 83*, pages 145–154. North Holland, Amsterdam, 1983. http://wwwmaths.anu.edu.au/~brent/pub/pub082.html. [50]

[36] Richard P. Brent and Edwin M. McMillan. Some new algorithms for high-precision computation of Euler's constant. *Mathematics of Computation*, 34(149):305–312, 1980. http://wwwmaths.anu.edu.au/~brent/pub/pub049.html. [164]

[37] Richard P. Brent, Colin Percival, and Paul Zimmermann. Error bounds on complex floating-point multiplication. *Mathematics of Computation*, 76:1469–1481, 2007. http://wwwmaths.anu.edu.au/~brent/pub/pub221.html. [125]

[38] Richard P. Brent and Joseph F. Traub. On the complexity of composition and generalized composition of power series. *SIAM Journal on Computing*, 9:54–66, 1980. http://wwwmaths.anu.edu.au/~brent/pub/pub050.html. [167]

[39] Richard P. Brent and Paul Zimmermann. Ten new primitive binary trinomials. *Mathematics of Computation*, to appear (accepted April 2008). http://wwwmaths.anu.edu.au/~brent/pub/pub233.html. [79]

[40] Peter Bürgisser, Michael Clausen, and M. Amin Shokrollahi (with the collaboration of Thomas Lickteig). *Algebraic Complexity Theory*. Grundlehren der mathematischen Wissenschaften 315. Springer, 1997. [126]

[41] Christoph Burnikel and Joachim Ziegler. Fast recursive division. Research Report MPI-I-98-1-022, MPI Saarbrücken, October 1998. `http://data.mpi-sb.mpg.de/internet/reports.nsf/NumberView/1998-1-022`. [49]

[42] David V. Chudnovsky and Gregory V. Chudnovsky. Computer algebra in the service of mathematical physics and number theory. In *Computers in Mathematics (Stanford, CA, 1986)*, volume 125 of *Lecture Notes in Pure and Applied Mathematics*, pages 109–232, New York, 1990. Dekker. [168]

[43] C. W. Clenshaw and F. W. J. Olver. An unrestricted algorithm for the exponential function. *SIAM J. Numerical Analysis*, 17:310–331, 1980. [141]

[44] W. D. Clinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 92–101, White Plains, NY, June 1990. [126]

[45] William J. Cody, J. T. Coonen, D. M. Gay, K. Hanson, D. Hough, William Kahan, R. Karpinski, J. Palmer, F. N. Ris, and D. Stevenson. A proposed radix- and word-length-independent standard for floating-point arithmetic. *IEEE Micro*, pages 86–100, August 1984. [124]

[46] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Graduate Texts in Mathematics 138. Springer-Verlag, 1993. 534 pages. [49, 79]

[47] George E. Collins and Werner Krandick. Multiprecision floating point addition. In Carlo Traverso, editor, *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation (ISSAC'2000)*, pages 71–77. ACM Press, 2000. [124]

[48] M. A. Cornea-Hasegan, R. A. Golliver, and P. Markstein. Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithms. In *Proceedings of Arith'14*, Adelaide, 1999. [125]

[49] Richard E. Crandall. *Projects in Scientific Computation*. TELOS, The Electronic Library of Science, Santa Clara, California, 1994. [125]

[50] Richard E. Crandall. *Topics in Advanced Scientific Computation*. TELOS, The Electronic Library of Science, Santa Clara, California, 1996. [103, 125]

[51] Richard E. Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. Springer-Verlag, second edition, 2005. [49]

[52] Annie Cuyt, Vigdis Brevik Petersen, Brigitte Verdonk, Haakon Waadeland, and William B. Jones (with contributions by Franky Backeljauw and Catherine Bonan-Hamada). *Handbook of Continued Fractions for Special Functions*. Springer, 2008. xvi+431 pages. [149]

[53] Brandon Dixon and Arjen K. Lenstra. Massively parallel elliptic curve factoring. In *Proceedings of Eurocrypt'92*, volume 658 of *Lecture Notes in Computer Science*, pages 183–193. Springer-Verlag, 1993. [49]

[54] Régis Dupont. Fast evaluation of modular functions using Newton iterations and the AGM. *Mathematics of Computation*, 2008 or 2009. To appear. [167]

[55] Miloš D. Ercegovac and Jean-Michel Muller. Complex square root with operand prescaling. *The Journal of VLSI Signal Processing*, 49(1):19–30, 2007. [126]

[56] Gerald Estrin. Organization of computer systems–the fixed plus variable structure computer. In *Proceedings Western Joint Computer Conference*, pages 33–40, May 1960. [167]

[57] George E. Forsythe. Pitfalls in computation, or why a math book isn't enough. *Amer. Math. Monthly*, 77:931–956, 1970. [140]

[58] Paul Friedland. Algorithm 312: Absolute value and square root of a complex number. *Communications of the ACM*, 10(10):665, 1967. [168]

[59] Martin Fürer. Faster integer multiplication. In David S. Johnson and Uriel Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC), San Diego, California, USA*, pages 57–66. ACM, 2007. [79, 125]

[60] David M. Gay. Correctly rounded binary-decimal and decimal-binary conversions. Numerical Analysis Manuscript 90-10, AT&T Bell Laboratories, November 1990. [126]

[61] *GNU MP: The GNU Multiple Precision Arithmetic Library*, 4.2.2 edition, 2007. `http://gmplib.org/`. [170]

[62] Xavier Gourdon and Pascal Sebah. Numbers, constants and computation. `http://numbers.computation.free.fr/Constants/constants.html`. [164]

[63] Bruno Haible and Thomas Papanikolaou. Fast multiprecision evaluation of series of rational numbers. In J. P. Buhler, editor, *Proceedings of the 3rd*

Algorithmic Number Theory Symposium (ANTS-III), volume 1423 of *Lecture Notes in Computer Science*, pages 338–350, 1998. [168, 169]

[64] Tom R. Halfhill. The truth behind the Pentium bug. *Byte*, March 1995. [131]

[65] Guillaume Hanrot, Michel Quercia, and Paul Zimmermann. The Middle Product Algorithm, I. Speeding up the division and square root of power series. *Applicable Algebra in Engineering, Communication and Computing*, 14(6):415–438, 2004. [125]

[66] Guillaume Hanrot and Paul Zimmermann. A long note on Mulders' short product. *Journal of Symbolic Computation*, 37:391–401, 2004. [49]

[67] Guillaume Hanrot and Paul Zimmermann. Newton iteration revisited. `http://www.loria.fr/~zimmerma/papers/fastnewton.ps.gz`, 2004. 2 pages. [167]

[68] David Harvey. Faster polynomial multiplication via multipoint Kronecker substitution. `http://arxiv.org/abs/0712.4046`, 2007. 14 pages. [45]

[69] William Hasenplaugh, Gunnar Gaubatz, and Vinodh Gopal. Fast modular reduction. In *Proceedings of the 18th Symposium on Computer Arithmetic*, pages 225–229, Montpellier, France, 2007. IEEE Computer Society Press. [78]

[70] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, second edition, 2002. [124]

[71] Thomas E. Hull. The use of controlled precision. In John K. Reid, editor, *The Relationship Between Numerical Computation and Programming Languages*, pages 71–84. North Holland, 1982. [124]

[72] Intel. Division, square root and remainder algorithms for the Intel(r) Itanium(tm) architecture, 2003. Application Note, `ftp://download.intel.com/software/opensource/divsqrt.pdf`, 120 pages. [167]

[73] Cristina Iordache and David W. Matula. On infinitely precise rounding for division, square root, reciprocal and square root reciprocal. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 233–240. IEEE Computer Society, 1999. [125]

[74] Tudor Jebelean. An algorithm for exact division. *Journal of Symbolic Computation*, 15:169–180, 1993. [49]

[75] Tudor Jebelean. A double-digit Lehmer-Euclid algorithm for finding the GCD of long integers. *Journal of Symbolic Computation*, 19:145–157, 1995. [50]

[76] Tudor Jebelean. Practical integer division with Karatsuba complexity. In W. W. Küchlin, editor, *Proceedings of International Symposium on Symbolic and Algebraic Computation (ISSAC'97)*, pages 339–341, Maui, Hawaii, 1997. [49]

[77] William Kahan. Idempotent binary → decimal → binary conversion. `http://www.cs.berkeley.edu/~wkahan/Math128/BinDecBin.pdf`, 2002. 1 page. [126]

[78] Ekatherina A. Karatsuba. Fast evaluation of hypergeometric functions by FEE. In N. Papamichael, St. Ruscheweyh, and E. B. Saff, editors, *Proceedings of Computational Methods and Function Theory (CMFT'97)*, pages 303–314. World Scientific Publishing, 1997. [168]

[79] Alan H. Karp and Peter Markstein. High-precision division and square root. *ACM Transactions on Mathematical Software*, 23(4):561–589, 1997. [49, 131, 167]

[80] Donald E. Knuth. The analysis of algorithms. In *Actes du Congrès International des Mathématiciens de 1970*, volume 3, pages 269–274, Paris, 1971. Gauthiers-Villars. [50]

[81] Donald E. Knuth. *The Art of Computer Programming*, volume 2 : Seminumerical Algorithms. Addison-Wesley, third edition, 1998. `http://www-cs-staff.stanford.edu/~knuth/taocp.html`. [18, 23, 37, 49, 50, 125, 126, 167]

[82] Werner Krandick and Tudor Jebelean. Bidirectional exact integer division. *Journal of Symbolic Computation*, 21(4–6):441–456, 1996. [49]

[83] Werner Krandick and Jeremy R. Johnson. Efficient multiprecision floating point multiplication with optimal directional rounding. In Earl Swartzlander, Mary Jane Irwin, and Graham Jullien, editors, *Proceedings of the 11th Symposium on Computer Arithmetic (ARITH'11)*, pages 228–233, 1993. [124]

[84] Hirondo Kuki and William J. Cody. A statistical study of the accuracy of floating-point number systems. *Communications of the ACM*, 16:223–230, 1973. [122]

[85] Ulrich W. Kulisch. *Computer Arithmetic and Validity. Theory, Implementation, and Applications*. Number 33 in Studies in Mathematics. de Gruyter, 2008. 410 pages. [126]

[86] Tomas Lang and Jean-Michel Muller. Bounds on runs of zeros and ones for algebraic functions. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 13–20. IEEE Computer Society, 2001. [125]

[87] V. Lefèvre. Multiplication by an integer constant. Research Report RR-4192, INRIA, May 2001. `ftp://ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-4192.ps.gz`. [21]

[88] V. Lefèvre. The generic multiple-precision floating-point addition with exact rounding (as in the MPFR library). In *Proceedings of the 6th Conference on Real Numbers and Computers*, pages 135–145, Dagstuhl, Germany, November 2004. [124]

[89] Charles F. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, 1992. [79]

[90] Roman Maeder. Storage allocation for the Karatsuba integer multiplication algorithm. DISCO, 1993. preprint. [46]

[91] Gérard Maze. Existence of a limiting distribution for the binary GCD algorithm. *Journal of Discrete Algorithms*, 5:176–186, 2007. `http://www.arxiv.org/abs/math.GM/0504426`. [50]

[92] Philip B. McLaughlin, Jr. New frameworks for Montgomery's modular multiplication method. *Mathematics of Computation*, 73(246):899–906, 2004. [78]

[93] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. `http://cacr.math.uwaterloo.ca/hac/`. [79]

[94] Valérie Ménissier-Morain. *Arithmétique exacte, conception, algorithmique et performances d'une implémentation informatique en précision arbitraire*. PhD thesis, University of Paris 7, 1994. `ftp.inria.fr/INRIA/Projects/cristal/Valerie.Menissier/these94.ps.gz`. [124]

[95] Ludovic Meunier and Bruno Salvy. ESF: an automatically generated encyclopedia of special functions. In *Proceedings of the 2003 international symposium on Symbolic and algebraic computation (ISSAC'03)*, pages 199–206, Philadelphia, PA, USA, 2003. ACM. [171]

[96] Preda Mihailescu. Fast convolutions meet Montgomery. *Mathematics of Computation*, 77:1199–1221, 2008. [78]

[97] R. Moenck and A. Borodin. Fast modular transforms via division. In *Proceedings of the 13th Annual IEEE Symposium on Switching and Automata Theory*, pages 90–96, October 1972. [49]

[98] Niels Möller. Notes on the complexity of CRT, February 2007. Preprint. 8 pages. [79]

[99] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. [79]

[100] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985. [78]

[101] Peter L. Montgomery. Personal communication, 2001. [47]

[102] T. Mulders. On short multiplications and divisions. *Applicable Algebra in Engineering, Communication and Computing*, 11(1):69–88, 2000. [122, 125]

[103] Jean-Michel Muller. *Elementary Functions. Algorithms and Implementation.* Birkhauser, 2006. 2nd edition. 265 pages. [124, 125]

[104] Victor Pan. *How to Multiply Matrices Faster*, volume 179 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984. [125]

[105] Michael S. Paterson and Larry J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973. [167]

[106] Colin Percival. Rapid multiplication modulo the sum and difference of highly composite numbers. *Mathematics of Computation*, 72(241):387–395, 2003. [79, 125]

[107] J. M. Pollard. The fast Fourier transform in a finite field. *Mathematics of Computation*, 25(114):365–374, April 1971. [79]

[108] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In Peter Kornerup and David Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–144, Grenoble, France, 1991. IEEE Computer Society Press. `http://www.cs.cmu.edu/~quake-papers/related/Priest.ps`. [124]

[109] B. Salvy and P. Zimmermann. Gfun: A Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software*, 20(2):163–177, June 1994. [168]

[110] Martin S. Schmookler and Kevin J. Nowka. Leading zero anticipation and detection – a comparison of methods. In Neil Burgess and Luigi Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH'15)*, pages 7–12. IEEE Computer Society, 2001. [124]

[111] Arnold Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971. [50]

[112] Arnold Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica*, 7:395–398, 1977. [78, 79]

[113] Arnold Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. In *Computer Algebra, EUROCAM'82*, number 144 in Lecture Notes in Computer Science, pages 3–15, 1982. [49]

[114] Arnold Schönhage. Variations on computing reciprocals of power series. *Information Processing Letters*, 74:41–46, 2000. [125]

[115] Arnold Schönhage, A. F. W. Grotefeld, and E. Vetter. *Fast Algorithms, A Multitape Turing Machine Implementation.* BI-Wissenschaftsverlag, 1994. [125]

[116] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971. [18]

[117] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 252–259, 1993. [49]

[118] David M. Smith. Algorithm 693: A Fortran package for floating-point multiple-precision arithmetic. *ACM Transactions on Mathematical Software*, 17(2):273–283, 1991. [167]

[119] Jonathan P. Sorenson. Two fast GCD algorithms. *Journal of Algorithms*, 16:110–144, 1994. [50]

[120] Allan Steel. Reduce everything to multiplication. Computing by the Numbers: Algorithms, Precision, and Complexity, Workshop for Richard Brent's 60th birthday, Berlin, July 2006. `http://www.mathematik.hu-berlin.de/~gaggle/EVENTS/2006/BRENT60/`. [49, 78]

[121] G. L. Steele and J. L. White. How to print floating-point numbers accurately. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 112–126, White Plains, NY, June 1990. [126]

[122] Damien Stehlé and Paul Zimmermann. A binary recursive gcd algorithm. In *Proceedings of the Algorithmic Number Theory Symposium (ANTS VI)*, 2004. [50]

[123] A. Svoboda. An algorithm for division. *Information Processing Machines*, 9:25–34, 1963. [24]

[124] Nico M. Temme. *Special Functions: An Introduction to the Classical Functions of Mathematical Physics*. Wiley, John and Sons, Inc., 1996. [167]

[125] J. F. Traub. *Iterative Methods for the Solution of Equations*. Prentice-Hall, Englewood Cliffs, New Jersey, 1964. [129]

[126] M. Urabe. Roundoff error distribution in fixed-point multiplication and a remark about the rounding rule. *SIAM Journal on Numerical Analysis*, 5:202–210, 1968. [122]

[127] Brigitte Vallée. Dynamics of the binary Euclidean algorithm: Functional analysis and operators. *Algorithmica*, 22:660–685, 1998. [50]

[128] Joris van der Hoeven. Relax, but don't be too lazy. *Journal of Symbolic Computation*, 34(6):479–542, 2002. `http://www.math.u-psud.fr/~vdhoeven`. [49, 77]

[129] Jean Vuillemin. private communication, January 2004. [46]

[130] Kenneth Weber. The accelerated integer GCD algorithm. *ACM Transactions on Mathematical Software*, 21(1):111–122, 1995. [38, 50]

[131] James H. Wilkinson. *Rounding Errors in Algebraic Processes*. HMSO, London, 1963. [124]

[132] James H. Wilkinson. *The Algebraic Eigevalue Problem*. Clarendon Press, Oxford, 1965. [124]

[133] C. K. Yap. *Fundamental Problems in Algorithmic Algebra*. Oxford University Press, 2000. [50]

[134] Paul Zimmermann. Karatsuba square root. Research Report 3805, INRIA, November 1999. [49]

[135] Dan Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, 1994. [17, 47]

Index