Modern Computer Arithmetic

Richard P. Brent and Paul Zimmermann

Version 0.1.1

Contents

Notation

β	the "word" base (usually 2^{32} or 2^{64})
t	number of base β digits in floating-point fraction or a temporary variable, depending on the context
ε	the "machine precision" $\frac{1}{2}\beta^{1-t}$
n	an integer
$M(n)$	the time to multiply n-bit integers or polynomials of degree $n-1$, depending on the context
$\mathrm{sign}(n)$	$+1$ 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 ($0 \le a - qb < b$)
$\nu(n)$	the 2-valuation of n, i.e. the largest power of two that divides n, with $\nu(0) = \infty$
log, ln	the natural logarithm
$\log_2$, lg	the base-2 logarithm
${}^t[a, b]$ or $[a, b]^t$	the vector $\begin{pmatrix} a \\ b \end{pmatrix}$
$[a, b; c, d]$	the 2×2 matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$
$\mathbb{Z}/n\mathbb{Z}$	the ring of residues modulo n
C^n	the set of (real or complex) functions with n continuous derivatives in the region of interest

$\bar{z}$	the conjugate of the complex number z
$\|z\|$	the Euclidean norm of the complex number z
$\mathrm{ord}(A)$	for a power series $A(z) = a_0 + a_1 z + \cdots$, $\mathrm{ord}(A) = \min\{j : a_j \neq 0\}$ (note the special case $\mathrm{ord}(0) = +\infty$)
$\mathbb{C}$	the set of complex numbers
$\mathbb{N}$	the set of natural numbers (nonnegative integers)
$\mathbb{Q}$	the set of rational numbers
$\mathbb{R}$	the set of real numbers
$\mathbb{Z}$	the set of integers
$\Re(z)$	the real part of a complex number z
$\Im(z)$	the imaginary part of a complex number z

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.

Several physical representations are possible. We consider here only the most common one, namely a dense representation in a fixed integral base. Choose a *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 $0 \le a_i \le \beta - 1$, and a_{n-1} is sometimes assumed to be non-zero. Since the base β is usually fixed in a given program, it does not need to be represented. Thus 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. 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 *lazy* or on-line algorithms, and *relaxed* algorithms [60].

1.2 Addition and Subtraction

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

```
1 Algorithm IntegerAddition.
2 Input: $A = \sum_0^{n-1} a_i\beta^i$, $B = \sum_0^{n-1} b_i\beta^i$
3 Output: $C := \sum_0^{n-1} c_i\beta^i$ and $0 \leq d \leq 1$ such that $A + B = d\beta^n + C$
4 $d \leftarrow 0$
5 for $i$ from $0$ to $n-1$ do
6       $s \leftarrow a_i + b_i + d$
7       $c_i \leftarrow s \bmod \beta$
8       $d \leftarrow s \operatorname{div} \beta$
9 Return $C, d$.
```

Let M be the number of different values taken by the data type representing the coefficients a_i, b_i. (Clearly $\beta \leq M$ but equality does not necessarily hold, e.g. $\beta = 10^9$ and $M = 2^{32}$.) At step 6, the value of s can be as large as $2\beta - 1$, which is not representable if $\beta = M$. 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 M — equals $t := a_i + b_i - M < a_i$; thus comparing t

and a_i will determine if a carry occurred. A third solution is to keep a bit in reserve, taking $\beta \leq \lfloor M/2 \rfloor$.

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 $-\beta \leq s < \beta$ (recall that mod gives a nonnegative remainder). The other steps are unchanged.

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. Assume we want to multiply two polynomials $A(x)$ and $B(x)$ with non-negative integer 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, $X > n\rho^2$, 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 $C(X)$.

Conversely, suppose we want to multiply two integers $a = \sum_{0 \leq i < n} a_i \beta^i$ and $b = \sum_{0 \leq j < n} b_j \beta^j$. Multiply the polynomials $A(x) = \sum_{0 \leq i < n} a_i x^i$ and $B(x) = \sum_{0 \leq j < n} b_j x^j$, 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 of order $n\beta^2$. These examples demonstrate the analogy between operations on polynomials and integers, and also show the limits of the analogy.

1.3.1 Naive Multiplication

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

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

Remark. 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 $A \cdot b_j$ at step 6, which is accumulated into C. Since all fast algorithms rely on multiplication, the most important operation to optimize in multiple-precision software is the multiplication of an array of m words by one word, with accumulation of the result in another array of m or $m+1$ 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.

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 (see Ex. 1.9.2).

Theorem 1.3.2 *Algorithm **KaratsubaMultiply** correctly computes the product AB, 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 and B_0 have (at most) $\lceil n/2 \rceil$ words, and $|A_0 - A_1|$ and $|B_0 - B_1|$, and A_1 and B_1 have $\lfloor n/2 \rfloor$ 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 $2^{l-1}n_0 \leq n \leq 2^l n_0$ with $l \geq 1$, 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 $C = 3^{1-\log_2 n_0} \max(K(n_0), (n_0 - 1)^2)$. □

```
Algorithm KaratsubaMultiply.
Input: $A = \sum_0^{n-1} a_i\beta^i$, $B = \sum_0^{n-1} b_j\beta^j$
Output: $C = AB := \sum_0^{2n-1} c_k\beta^k$
if $n < n_0$ then return BasecaseMultiply$(A, B)$
$k \leftarrow \lceil n/2 \rceil$
$(A_0, B_0) := (A, B) \bmod \beta^k$, $(A_1, B_1) := (A, B) \operatorname{div} \beta^k$
$s_A \leftarrow \operatorname{sign}(A_0 - A_1)$, $s_B \leftarrow \operatorname{sign}(B_0 - B_1)$
$C_0 \leftarrow$ KaratsubaMultiply$(A_0, B_0)$
$C_1 \leftarrow$ KaratsubaMultiply$(A_1, B_1)$
$C_2 \leftarrow$ KaratsubaMultiply$(|A_0 - A_1|, |B_0 - B_1|)$
Return $C := C_0 + (C_0 + C_1 - s_A s_B C_2)\beta^k + C_1\beta^{2k}$.
```

This variant of Karatsuba's algorithm is known as the *subtractive* version. Different variants of Karatsuba's algorithm exist. 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 "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.

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 (however 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, that will be used both for those intermediate results and for the recursive calls. It can be shown that an auxiliary space of $m = 2n$ words is sufficient (see Ex. 1.9.3).

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. 1.9.5).

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. 1.9.4).

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_1x and B_0+B_1x evaluated in $x=\beta^k$; 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

The above 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 $b_0+\cdots+b_{r-1}x^{r-1}$, with $x \leftarrow \beta^k$, 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 books, for example [52], 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 $O(n^\beta)$ with $\beta=\frac{\log(2r-1)}{\log r}$. 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 people simply say "Toom-Cook algorithm" for $r=3$. The following algorithm uses evaluation points $0,1,-1,2,\infty$, and tries to optimize the evaluation and interpolation formulæ.

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

We refer the reader interested in higher order Toom-Cook implementations to [65], which considers the 4- and 5-way variants, and also squaring. Toom-Cook r-way has to invert a $(2r-1)\times(2r-1)$ Vandermonde matrix with

[1] Evaluating $C(x)$ at ∞ means computing the product A_1B_1 of the leading coefficients.

[2] An exact division can be performed from the least significant bits, which is usually more efficient: see §1.4.5.

```
Algorithm ToomCook3.
Input: two integers $0 \le A, B < \beta^n$.
Output: $AB := c_0 + c_1\beta^k + c_2\beta^{2k} + c_3\beta^{3k} + c_4\beta^{4k}$ with $k = \lceil n/3 \rceil$.
if $n < 3$ then return KaratsubaMultiply$(A, B)$
Write $A = a_0 + a_1x + a_2x^2$, $B = b_0 + b_1x + b_2x^2$ with $x = \beta^k$.
$v_0 \leftarrow$ ToomCook3$(a_0, b_0)$
$v_1 \leftarrow$ ToomCook3$(a_{02} + a_1, b_{02} + b_1)$ where $a_{02} \leftarrow a_0 + a_2, b_{02} \leftarrow b_0 + b_2$
$v_{-1} \leftarrow$ ToomCook3$(a_{02} - a_1, b_{02} - b_1)$
$v_2 \leftarrow$ ToomCook3$(a_0 + 2a_1 + 4a_2, b_0 + 2b_1 + 4b_2)$
$v_\infty \leftarrow$ ToomCook3$(a_2, b_2)$
$t_1 \leftarrow (3v_0 + 2v_{-1} + v_2)/6 - 2v_\infty$, $t_2 \leftarrow (v_1 + v_{-1})/2$
$c_0 \leftarrow v_0$, $c_1 \leftarrow v_1 - t_1$, $c_2 \leftarrow t_2 - v_0 - v_\infty$, $c_3 \leftarrow t_1 - t_2$, $c_4 \leftarrow v_\infty$
```

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 (see Ex. 1.9.8 for a generalization of this result).

1.3.4 Fast Fourier Transform

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. The Fast Fourier Transform (FFT) is a way to perform evaluation and interpolation in an efficient way for some special values of r. This explains why multiplication algorithms of best asymptotic complexity are based on the Fast Fourier Transform (FFT).

There are different flavours of FFT multiplication, depending on the ring where the operations are performed. The asymptotically best algorithm, due to Schönhage-Strassen [53], 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.

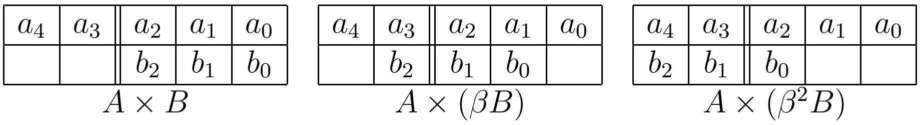
Another method commonly used is to work with floating-point complex numbers [36, Section 4.3.3.C]; one drawback is that, due to the inexact nature of floating-point computations, a careful error analysis is required to guarantee the correctness of the implementation. We refer to Chapter 3 for a description of this method.

1.3.5 Unbalanced Multiplication

How to 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, with $m \geq n$.

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. 1.9.9.

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, the Karatsuba cut being represented by a double column:



a_4	a_3	a_2	a_1	a_0
		b_2	b_1	b_0

$A \times B$

a_4	a_3	a_2	a_1	a_0
	b_2	b_1	b_0	

$A \times (\beta B)$

a_4	a_3	a_2	a_1	a_0
b_2	b_1	b_0		

$A \times (\beta^2 B)$

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 $m/2 \leq n \leq m$, any such "padding strategy" will require $K(\lceil m/2 \rceil, \lceil m/2 \rceil)$ for the product of the differences 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 $O(m^\alpha)$ cost. The "odd-even strategy" (Ex. 1.9.10) avoids this wrap around. For example, we get $K(3, 2) = 5$ with the odd-even strategy; compare $K(3, 2) = 6$ for the classical strategy.

As for the classical strategy, there are several ways of padding with the odd-even strategy. Consider again $m = 5$, $n = 3$, 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 both operands equal. 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. 1.9.11).

For naive multiplication, Algorithm **BasecaseMultiply** (§1.3.1) can be modified to obtain a theoretical speedup of two, since only 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. However, the speedup obtained is less than two, and the threshold obtained is larger than the corresponding multiplication threshold (see Ex. 1.9.11).

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, and not only its size in bits or digits. They also differ in the model of complexity used. For example, Bernstein's algorithm [7], which is used by several compilers to compute addresses in data structure records, considers as basic operation $x, y \to 2^ix \pm y$, with a cost assumed to be independent of the integer i.

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

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

We refer the reader to [39] 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 §2.2.1).

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 dividing two residues modulo n) is needed. Finally 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

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 (except when $\beta = 2$) than simply requiring that b_{n-1} be nonzero.

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.

```
 1  Algorithm BasecaseDivRem.
 2  Input: $A = \sum_0^{n+m-1} a_i\beta^i$, $B = \sum_0^{n-1} b_j\beta^j$, $B$ normalized
 3  Output: quotient $Q$ and remainder $R$ of $A$ divided by $B$.
 4  if $A \ge \beta^m B$ then $q_m \leftarrow 1$, $A \leftarrow A - \beta^m B$ else $q_m \leftarrow 0$
 5  for $j$ from $m-1$ downto $0$ do
 6      $q_j^* \leftarrow \lfloor (a_{n+j}\beta + a_{n+j-1})/b_{n-1} \rfloor$
 7      $q_j \leftarrow \min(q_j^*, \beta - 1)$
 8      $A \leftarrow A - q_j\beta^j B$
 9      while $A < 0$ do
10          $q_j \leftarrow q_j - 1$
11          $A \leftarrow A + \beta^j B$
12  Return $Q = \sum_0^m q_j\beta^j$, $R = A$.
```

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

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 $q_j b_{n-1} \ge a_{n+j}\beta + a_{n+j-1} - b_{n-1} + 1$, thus

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

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 $q_j b_{n-1} \le a_{n+j}\beta + a_{n+j-1}$:

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

Therefore $A - q_j\beta^j B + 2\beta^j B \ge (2b_{n-1} - q_j)\beta^{n+j-1} > 0$, which proves that the while-loop at steps 9-11 is performed at most twice [36, 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 $q_j \ne q_j^*$, *i.e.* $q_j^* \ge \beta$, 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 $(\sum_j^{m-1} q_i \beta^i)B + A$ 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), thus the total cost is $O(nm)$.

□

Here is an example of algorithm **BasecaseDivRem** for the inputs $A = 766970544842443844$ and $B = 862664913$, with $\beta = 1000$:

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

which gives as quotient $Q = 889071217$ and as remainder $R = 778334723$.

REMARK 1: 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.

REMARK 2: 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 $A - (\beta - 1)\beta^j B = A - \beta^{j+1} B + \beta^j B$.

REMARK 3: if instead of having B normalized, i.e. $b_n \geq \beta/2$, we have $b_n \geq \beta/k$, one can have up to k iterations of the while-loop (and step 4 has to be modified accordingly).

REMARK 4: a drawback of algorithm **BasecaseDivRem** is that the $A < 0$ test at line 9 is true with non-negligible probability, therefore it will cause branch prediction algorithms available on modern processors to fail, resulting in wasted cycles. A workaround is to compute a more accurate partial quotient, and therefore decrease the proportion of corrections almost to zero (see Ex. 1.9.14).

1.4.2 Divisor Preconditioning

It sometimes happens that the quotient selection — step 6 of algorithm **BasecaseDivision** — 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; then one way to compute q_j^* is 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 [56] makes the quotient selection trivial, after preconditioning the divisor. The main idea is that if b_{n-1} equals the base β, then the quotient selection is easy, since it suffices to take $q_j^* = a_{n+j}$. (In addition, the condition of step 7 is then always fulfilled.)

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

Remarks: At step 8, the most significant word $a_{n+j}\beta^{n+j}$ automatically cancels with $q_j\beta^{j-1}\beta^{n+1}$; one can thus subtract only the product of q_j by the lower part $\sum_0^{n-1} b'_j\beta^j$ of B'. 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 previous section, Svoboda's algorithm would give $k = 1160$, $B' = 1000691299080$,

j	A	q_j	$A - q_jB'\beta^j$	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 gives $R' = 817B + 778334723$, thus we finally get $Q = 1160 \cdot 766440 + 817 = 889071217$, and $R = 778334723$.

Svoboda's algorithm is especially interesting when only the remainder is needed, since one then avoids the post-normalization $Q = q_0 + kQ'$.

1.4.3 Divide and Conquer Division

The base-case division 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:

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

Then since q_j^* has now two words, one can use fast multiplication algorithms (§1.3) to 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 — depends mainly 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_j^*; in such a way that we only have to subtract the product of q_j from the low part of the divisor.

```
Algorithm RecursiveDivRem.
Input: $A = \sum_0^{n+m-1} a_i\beta^i$, $B = \sum_0^{n-1} b_j\beta^j$, $B$ normalized, $n \ge m$
Output: quotient $Q$ and remainder $R$ of $A$ divided by $B$.
if $m < 2$ then return BasecaseDivRem($A, B$)
$k \leftarrow \lfloor \frac{m}{2} \rfloor$, $B_1 \leftarrow B$ div $\beta^k$, $B_0 \leftarrow B \bmod \beta^k$
$(Q_1, R_1) \leftarrow$ RecursiveDivRem($A$ div $\beta^{2k}, B_1$)
$A' \leftarrow R_1\beta^{2k} + A \bmod \beta^{2k} - Q_1\beta^k B_0$
while $A' < 0$ do $Q_1 \leftarrow Q_1 - 1$, $A' \leftarrow A' + \beta^k B$
$(Q_0, R_0) \leftarrow$ RecursiveDivRem($A'$ div $\beta^k, B_1$)
$A'' \leftarrow R_0\beta^k + A' \bmod \beta^k - Q_0 B_0$
while $A'' < 0$ do $Q_0 \leftarrow Q_0 - 1$, $A'' \leftarrow A'' + B$
Return $Q := Q_1\beta^k + Q_0$, $R := A''$.
```

Theorem 1.4.2 *Algorithm **RecursiveDivRem** is correct, and uses $D(m,n)$ operations, where $D(2m,n) = 2D(m,n-m) + 2M(m) + O(n)$. In particular $D(n) := D(n,n)$ satisfies $D(2n) = 2D(n) + 2M(n) + O(n)$, 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 $A = (Q_1 B_1 + R_1)\beta^{2k} + A \bmod \beta_{2k}$, thus after step 7: $A' = A - Q_1 \beta^k B$, which still holds after step 8. After step 9, we have $A' = (Q_0 B_1 + R_0)\beta^k + A' \bmod \beta^k$, thus after step 10: $A'' = A' - Q_0 B$, which still holds after step 11. At step 12 we thus have $A = QB + R$.

A div β^{2k} has $m+n-2k$ words, while B_1 has $n-k$ words, thus $0 \le Q_1 < 2\beta^{m-k}$ and $0 \le R_1 < B_1 < \beta^{n-k}$. 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. At step 9 we have $0 \le A' < \beta^k B$, thus A' div β^k has at most n words. It follows $0 \le Q_0 < 2\beta^k$ and $0 \le R_0 < B_1 < \beta^{n-k}$. Hence at step 10, $-2\beta^{2k} < A'' < B$, and after at most four iterations at step 11, we have $0 \le A'' < B$. □

A graphical view of Algorithm **RecursiveDivRem** in the case $m = 2n$ is given on Fig. 1.1, which represents the multiplication $Q \cdot B$: one firstly computes the lower left corner in $D(n/2)$, secondly the lower right corner in $M(n/2)$, thirdly the upper left corner in $D(n/2)$, and finally the upper right corner in $M(n/2)$. **RecursiveDivRem**.

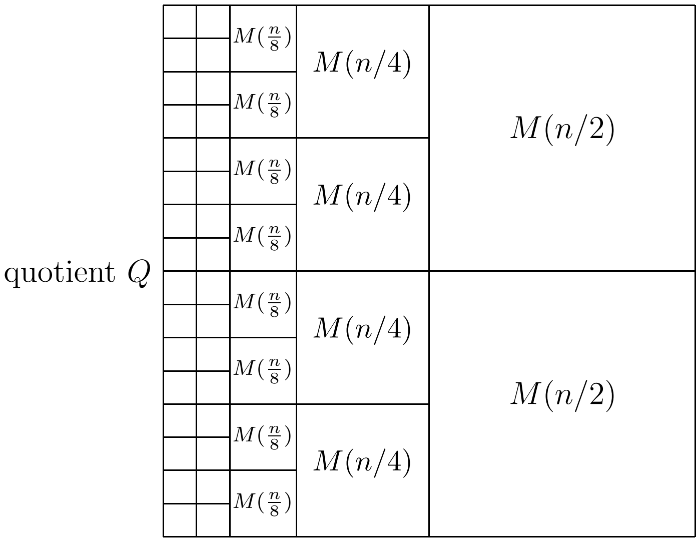


Figure 1.1: Divide and conquer division: a graphical view (most significant parts at the lower left corner).

REMARK 1: we may replace the condition $m < 2$ at step 4 by $m < T$ for any integer $T \geq 2$. In practice, T may be in the range 50 to 200 words.

REMARK 2: we can not require here $A < \beta^m B$, since this condition may not be satisfied in the recursive calls. Consider for example $A = 5517$, $B = 56$ with $\beta = 10$: the first recursive call will divide 55 by 5, which requires a two-digit quotient. 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. 1.9.13). See also Ex. 1.9.15.

REMARK 3: Theorem 1.4.2 gives $D(n) \sim 2M(n)$ for Karatsuba multiplication, and $D(n) \sim 2.63M(n)$ for Toom-Cook 3-way. In the FFT range, see Ex. 1.9.16.

REMARK 4: the same idea as in Ex. 1.9.14 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**.

Large dividend

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), the best strategy (see Ex. 1.9.17) is to get n words of the quotient at a time (this simply reduces to the base-case algorithm, replacing β by β^n).

1.4.4 Newton's Division

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 (Algorithm **InvRem**, page 25). We refer here to Ch. 4. The p-adic version of Newton's method, also called Hensel lifting, is used in §1.4.5 for the exact division.

Theorem 1.4.3 *Algorithm **InvRem** is correct.*

```
Algorithm **UnbalancedDivision**.
**Input**: $A = \sum_0^{n+m-1} a_i\beta^i$, $B = \sum_0^{n-1} b_j\beta^j$.
**Output**: quotient $Q$ and remainder $R$ of $A$ divided by $B$.
Assumptions: $m > n$, $B$ normalized.
$Q \leftarrow 0$
**while** $m > n$ **do**
    $(q, r) \leftarrow \mathrm{RecursiveDivRem}(A \text{ div } \beta^{m-n}, B)$
    $Q \leftarrow Q\beta^n + q$
    $A \leftarrow r\beta^{m-n} + A \bmod \beta^{m-n}$
    $m \leftarrow m - n$
$(q, r) \leftarrow \mathrm{RecursiveDivRem}(A, B)$
Return $Q := Q\beta^m + q$, $R := r$.
```

```
Algorithm **InvRem**.
**Input**: a positive integer $B$, $\frac{1}{2}\beta^n \le B < \beta^n$.
**Output**: $X, R$ such that $\beta^{2n} = BX + R$, $0 \le R < B$.
**if** $n = 1$ **then** return **NaiveInvRem**$(B)$.
$h \leftarrow \lceil n/2 \rceil, l \leftarrow \lfloor n/2 \rfloor$, write $B = B_h\beta^l + B_l$
$(X_h, R_h) \leftarrow$ **InvRem**$(B_h)$
$Y \leftarrow R_h\beta^l - B_l X_h$
**while** $Y < 0$ **do** { $X_h \leftarrow X_h - 1$,   $Y \leftarrow Y + B$ }
$X_l \leftarrow \lfloor \frac{X_h Y}{\beta^{2h}} \rfloor$
$R \leftarrow Y\beta^l - BX_l$
**while** $R < 0$ **do** { $X_l \leftarrow X_l - 1$,   $R \leftarrow R + B$ }
**while** $R \ge B$ **do** { $X_l \leftarrow X_l + 1$,   $R \leftarrow R - B$ }
Return $X_h\beta^l + X_l, R$.
```

Proof At step 6, by induction $\beta^{2h} = B_h X_h + R_h$ with $0 \leq R_h < B_h$. Thus we have

$$\beta^{2n} = (B_h X_h + R_h)\beta^{2l} = (BX_h + Y)\beta^l = BX + R.$$

The conditions $0 \leq R < B$ are ensured thanks to the while loops at the end of the algorithm. □

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 division algorithms (most significant bits first), 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 start from least significant bits first. Indeed, 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: $A = \sum_0^{n-1} a_i\beta^i$, $B = \sum_0^{n-1} b_j\beta^j$
3  Output: quotient $Q = A/B \bmod \beta^n$
4  $C \leftarrow 1/b_0 \bmod \beta$
5  for $i$ from $\lceil \log_2 n \rceil - 1$ downto 1 do
6      $k \leftarrow \lceil n/2^i \rceil$
7      $C \leftarrow C + C(1 - BC) \bmod \beta^k$
8  $Q \leftarrow AC \bmod \beta^k$
9  $Q \leftarrow Q + C(A - BQ) \bmod \beta^n$
```

Remark: This algorithm 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) 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 first strategy, and the least $n/2$ bits using the second 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. 1.9.19).

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 and remainder is needed. For example, the division of two floating-point numbers reduces to the quotient of their fractions (see Ch. 3). Conversely, the multiplication of two numbers modulo n reduces to the remainder of their product after division by n (see Ch. 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 bits of the current remainder (line 8 of Algorithm `BasecaseDivRem`).

Surprisingly, 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 $A \bmod B_1$ 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.

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). The following algorithm performs a modular division:

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

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

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

Theorem 1.4.4 *The output of Algorithm* ***ConstantDivide*** *satisfies* $A = cQ + b\beta^n$.

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 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 $A_i + (b' + b'')\beta^{i+1} = A_{i-1} + \beta^i(a_i + b' + b''\beta) = cQ_{i-1} - b\beta^i + \beta^i(x + b - b' + b' + b''\beta) = cQ_{i-1} + \beta^i(x + b''\beta) = cQ_i$. □

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

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. the least significant bit of B to be set for β a power of two.

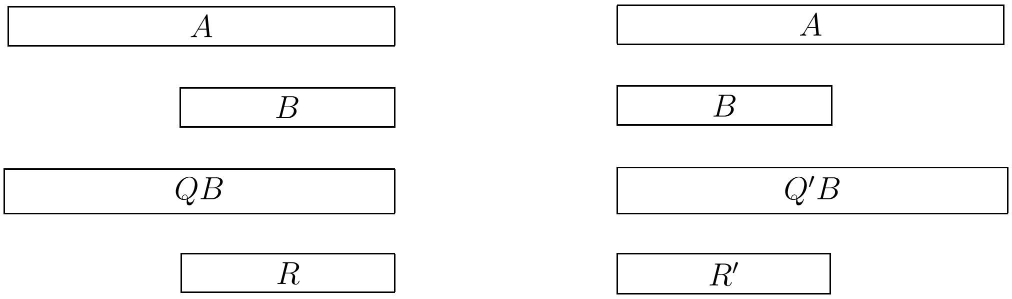


Figure 1.2: Classical/MSB division (left) vs Hensel/LSB division (right).

The LSB-quotient is uniquely defined by $Q' = A/B \bmod \beta^n$, with $0 \leq Q' < \beta^n$. 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 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.

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 in the form $s^2 + r$, taking two digits at a time of m, and finding one digit at a time of s. 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, 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 $2s$.

The following algorithm 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 Ch. 4):

```
Algorithm SqrtRem.
Input: $m = a_{n-1}\beta^{n-1} + \cdots + a_1\beta + a_0$ with $a_{n-1} \ne 0$
Output: $(s, r)$ such that $s^2 \le m = s^2 + r < (s+1)^2$
$l \leftarrow \lfloor \frac{n-1}{4} \rfloor$
if $l = 0$ then return BasecaseSqrtRem($m$)
write $m = a_3\beta^{3l} + a_2\beta^{2l} + a_1\beta^l + a_0$ with $0 \le a_2, a_1, a_0 < \beta^l$
$(s', r') \leftarrow$ SqrtRem($a_3\beta^l + a_2$)
$(q, u) \leftarrow$ DivRem($r'\beta^l + a_1, 2s'$)
$s \leftarrow s'\beta^l + q$
$r \leftarrow u\beta^l + a_0 - q^2$
if $r < 0$ then
    $r \leftarrow r + 2s - 1$
    $s \leftarrow s - 1$
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 square 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, and $R(n) \sim \frac{31}{6}M(n)$ with FFT multiplication, assuming $S(n) \sim \frac{2}{3}M(n)$.*

1.5.2 k-th Root

The above idea 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 $k{s'}^{k-1}$ 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 directly recomputing $(s'\beta + t)^k$.

Cube Root.

We illustrate with the case $k = 3$ (the cube root), where `BasecaseCbrtRem` is a naive algorithm that should deal with inputs of up to 6 words.

```
Algorithm **CbrtRem**.
**Input**: $0 \le n = n_{d-1}\beta^{d-1} + \cdots + n_1\beta + n_0$ with $0 \le n_i < \beta$
**Output**: $(s, r)$ such that $s^3 \le n = s^3 + r < (s+1)^3$
$l \leftarrow \lfloor \frac{d-1}{6} \rfloor$
**if** $l = 0$ **then** return BasecaseCbrtRem($n$)
write $n$ as $n'b^3 + a_2b^2 + a_1b + a_0$ where $b := \beta^l$
$(s', r') \leftarrow \mathrm{CbrtRem}(n')$
$(q, u) \leftarrow \mathrm{DivRem}(br' + a_2, 3s'^2)$
$r \leftarrow b^2u + ba_1 + a_0 - q^2(3s'b + q)$
$s \leftarrow bs' + q$
**while** $r < 0$ **do**
    $r \leftarrow r + 1 - 3s + 3s^2$
    $s \leftarrow s - 1$
Return $(s, r)$.
```

Exact Newton Iteration for k-th Root.

Theorem 1.5.2 *Algorithm* `RootRem` *is correct.*

Proof We prove by induction on n that the returned values s and r satisfy $n = s^k + r$ and $s^k \le n < (s+1)^k$. With the notations from the algorithm, we have $s = s'\beta + q$, thus $t = s^k$ and $r = n - t$, which proves that $n = s^k + r$.

```
1  Algorithm RootRem.
2  Input: $n \ge 0$
3  Output: $(s, r)$ such that $s^k \le n = s^k + r < (s+1)^k$
4  if $n < N$ use a naive algorithm
5  choose a base $\beta$ such that $n \ge \beta^2$
6  write $n = n_2\beta^k + n_1\beta^{k-1} + n_0$ with $n_2 \ge \beta^k$, $0 \le n_1 < \beta$, $0 \le n_0 < \beta^{k-1}$
7  $(s', r') \leftarrow$ RootRem($n_2$)
8  $q \leftarrow \lfloor \frac{r'\beta + n_1}{ks'^{k-1}} \rfloor$
9  $t \leftarrow (s'\beta + q)^k$
10 while $t > n$ do
11     $q \leftarrow q - 1$
12     $t \leftarrow (s'\beta + q)^k$
13 Return $(s'\beta + q, n - t)$.
```

Assuming the while loop exits, $t \le n$, thus $r \ge 0$ and $s^k \le n$. It only remains to prove that $n < (s+1)^k$.

If the while-loop is entered, this means that the previous value of t was larger than n, i.e. $(s+1)^k > n$. It thus suffices to prove that $n < (s'\beta+q+1)^k$. We first have $\frac{r'\beta+n_1}{ks'^{k-1}} < q+1$, thus $r'\beta + n_1 \le (q+1)ks'^{k-1} - 1$. This gives

$$\begin{aligned} n &= n_2\beta^k + n_1\beta^{k-1} + n_0 = (s'^k + r')\beta^k + n_1\beta^{k-1} + n_0 \\ &= s'^k\beta^k + (r'\beta + n_1)\beta^{k-1} + n_0 \le s'^k\beta^k + (q+1)ks'^{k-1}\beta^{k-1} - \beta^{k-1} + n_0 \\ &< s'^k\beta^k + (q+1)ks'^{k-1}\beta^{k-1} \le (s'\beta + q + 1)^k. \end{aligned}$$

□

However, the above result is not very satisfactory, since we have no bound on the number of iterations of the while-loop in Algorithm RootRem. The following lemma shows how to choose β at step 5 to ensure the while-loop is performed only once, and thus can be replaced by a if-test, as in Algorithm SqrtRem.

Lemma 1.5.1 *If $s' \ge k\beta$ at step 7 of Algorithm RootRem, then at most one correction is necessary.*

Proof Let q' be the final value of q at step 13, and q the value at step 8. By hypothesis we have $n = n_2\beta^k + n_1\beta^{k-1} + n_0 < (s'\beta + q' + 1)^k$, thus we deduce:

$$\begin{aligned} q &\le \frac{r'\beta + n_1}{ks'^{k-1}} = \frac{(n_2 - s'^k)\beta^k + n_1\beta^{k-1}}{ks'^{k-1}\beta^{k-1}} < \frac{(s'\beta + q' + 1)^k - (s'\beta)^k}{ks'^{k-1}\beta^{k-1}} \\ &= q' + 1 + \frac{(q'+1)^2}{ks'\beta} \sum_{i=0}^{k-2} \binom{k}{i+2} \left(\frac{q'+1}{s'\beta}\right)^i . \end{aligned}$$

It can be shown that $\sum_{i=0}^{k-2}\binom{k}{i+2}x^i = (1+x)^k/x^2 - 1/x^2 - k/x < (e-2)k^2$ for $x \le 1/k$. We thus conclude that for $\frac{q'+1}{s'\beta} \le 1/k$ — which is true since $s' \ge k\beta > k$ and $q' < \beta$ — we have $q < q' + 1 + \frac{(e-2)k\beta}{s'} \le q' + e - 1$. Since both q and q' are integers, and $e - 1 < 2$, it follows $q \le q' + 1$. □

This lemma shows that it suffices to choose β slightly smaller — by $\log k$ bits or by one word — to ensure there is at most one correction. In practice, especially for large operands, one may want to take a few extra bits in s'. Indeed, if we take $s' \ge 2^g k\beta$, assuming $\frac{r'\beta+n_1}{ks'^{k-1}}$ is uniformly distributed in $[q', q'+1+(e-2)k\beta/s']$, the probability of correction is less than $(e-2)2^{-g}$. With $g = 6$ for example, this is about 1%.

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 the "exact root" algorithms shown above, 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 $s^k = n \bmod \beta^l$ for any integer l. However, in the case of exact division, the equation $a = qb \bmod \beta^l$ has only one solution q as soon as b is relatively prime to β. Here, the equation $s^k = n \bmod \beta^l$ may have several solutions, so the lifting process is not unique. For example, $x^2 = 1 \bmod 2^3$ has four solutions.

Suppose we have $s^k = n \bmod \beta^l$, and we want to lift to β^{l+1}. We want $(s + t\beta^l)^k = n + n'\beta^l \bmod \beta^{l+1}$ where $0 \le t, n' < \beta$. Thus $kt = n' + \frac{n-s^k}{\beta^l} \bmod \beta$. 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. The following algorithm detects exact powers, and returns the largest exponent. To early detect non-kth powers at step 5, one may use modular algorithms when k is relatively prime to the base β (see above).

```
1 Algorithm IsPower.
2 Input: a positive integer $n$.
3 Output: $k$ if $n$ is an exact $k$th power, false otherwise.
4 for $k$ from $\lfloor \log_2 n \rfloor$ downto 2 do
5     if $n$ is a $k$th power, return $k$
6 Return false.
```

1.6 Gcd

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

- left-to-right versus right-to-left 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

We do not give Euclid's algorithm here: it can be found in many textbooks, *e.g.* Knuth [36], and we do not recommend it in its simplest form, except for testing purposes. Indeed, it is one of the slowest ways 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 Knuth [36, §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" instead — uses the *two* most significant words instead, which gives cofactors t, u, v, w of one full-word. 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 $97, 115, 231$.

```
Algorithm DoubleDigitGcd.
Input: $a := a_{n-1}\beta^{n-1} + \cdots + a_0$, $b := b_{m-1}\beta^{m-1} + \cdots + b_0$.
Output: $\gcd(a, b)$.
if $b = 0$ then return $a$
if $m < 2$ then return BasecaseGcd($a, b$)
if $a < b$ or $n > m$ then return DoubleDigitGcd($b, a \bmod b$)
$(t, u, v, w) \leftarrow$ HalfBezout($a_{n-1}\beta + a_{n-2}, b_{n-1}\beta + b_{n-2}$)
Return DoubleDigitGcd($|ta + ub|, |va + wb|$).
```

Note: in `DoubleDigitGcd`, we assume a and b are the current value of $a_{n-1}\beta^{n-1} + \cdots + a_0$ and $b_{m-1}\beta^{m-1} + \cdots + b_0$, with n and m updated after each step, so that $a_{n-1} \neq 0$ and $b_{m-1} \neq 0$. 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 $\begin{pmatrix} t & u \\ v & w \end{pmatrix}$.

Binary Gcd. A better algorithm than Euclid's one, still with an $O(n^2)$ complexity, is the *binary* algorithm. It differs from Euclid's algorithm in two ways: firstly it consider least significant bits first, and secondly it avoids expensive divisions, which most of the time give a small quotient.

```
Algorithm **BinaryGcd**.
**Input**: $a, b > 0$.
**Output**: $\gcd(a, b)$.
$i \leftarrow 0$
while $a \bmod 2 = b \bmod 2 = 0$ do
    $(i, a, b) \leftarrow (i + 1, a/2, b/2)$
while $a \bmod 2 = 0$ do
    $a \leftarrow a/2$
while $b \bmod 2 = 0$ do
    $b \leftarrow b/2$
while $a \neq b$ do
    $(a, b) \leftarrow (|a - b|, \min(a, b))$
    repeat $a \leftarrow a/2$ until $a \bmod 2 \neq 0$
Return $2^i \cdot a$.
```

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 *[62] 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 $ua \equiv vb \bmod m$.*

```
1  Algorithm ReducedRatMod.
2  Input: $a, b > 0$, $m > 1$ with $\gcd(a, m) = \gcd(b, m) = 1$
3  Output: $(u, v)$ such that $0 < |u|, v < \sqrt{m}$ and $ua \equiv vb \bmod m$
4  $c \leftarrow a/b \bmod m$
5  $(u_1, v_1) \leftarrow (0, m)$
6  $(u_2, v_2) \leftarrow (1, c)$
7  while $v_2 \geq \sqrt{m}$ do
8      $q \leftarrow \lfloor v_1/v_2 \rfloor$
9      $(u_1, u_2) \leftarrow (u_2, u_1 - qu_2)$
10     $(v_1, v_2) \leftarrow (v_2, v_1 - qv_2)$
11 return $(u_2, v_2)$.
```

The following algorithm, `ReducedRatMod`, finds such a pair (u, v): it is a simple variation 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 can be performed efficiently using Hensel lifting (§2.3), otherwise by an extended gcd algorithm (§1.6.2).

1.6.2 Extended Gcd

Algorithm `ExtendedGcd` (Table 1.1) 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: $a = ua_0 + vb_0$, and $b = wa_0 + xb_0$.

An important special case is modular inversion (see Ch. 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 yields u and v with $ua + vn = 1$, thus $1/a = u \bmod n$. But since v is not needed here, we can simply avoid computing v and x, by removing lines 4 and 9.

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 refer to §1.6.3 for subquadratic extended gcd.

```
**Input**: integers $a$ and $b$.
**Output**: integers $(g, u, v)$ such that $g = \gcd(a, b) = ua + vb$.
$(u, w) \leftarrow (1, 0)$
$(v, x) \leftarrow (0, 1)$
**while** $b \neq 0$ **do**
    $(q, r) \leftarrow \mathrm{DivRem}(a, b)$
    $(a, b) \leftarrow (b, r)$
    $(u, w) \leftarrow (w, u - qw)$
    $(v, x) \leftarrow (x, v - qx)$
Return $(a, u, v)$.
```

Table 1.1: Algorithm `ExtendedGcd`.

1.6.3 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 $q_i = r_{i-1}$ div r_i are usually small, and computing them is less expensive.

The main idea is thus to compute the partial quotients without computing the partial remainders. This can be seen as an 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 `HalGcd` algorithm:

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 $H^*(n) \sim H(n) - 8M(n/4)$.

```
Algorithm HalfGcd.
Input: $a \geq b > 0$
Output: a $2 \times 2$ matrix $R$ and $a', b'$ such that $[a'\, b']^t = R\,[a\, b]^t$
$n \leftarrow \mathrm{nbits}(a)$, $k \leftarrow \lfloor n/2 \rfloor$
$a := a_1 + 2^k a_0$, $b := b_1 + 2^k b_0$
$S, a_2, b_2 \leftarrow$ HalfGcd$(a_1, b_1)$
$a' \leftarrow a_2 2^k + S_{11} a_0 + S_{12} b_0$
$b' \leftarrow b_2 2^k + S_{21} a_0 + S_{22} b_0$
$l \leftarrow \lfloor k/2 \rfloor$
$a' := a'_1 2^l + a'_0$, $b' := b'_1 + 2^k b_0$
$T, a'_2, b'_2 \leftarrow$ HalfGcd$(a'_1, b'_1)$
$a'' \leftarrow a'_2 2^l + T_{11} a'_0 + T_{12} b'_0$
$b'' \leftarrow b'_2 2^l + T_{21} a'_0 + T_{22} b'_0$
Return $S \cdot T$, $a'', b''$.
```

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

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 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: see Table 1.3. The idea is to mimic the left-to-right version, by defining an appropriate right-to-left division (Algorithm `BinaryDivide`).

```
Algorithm **BinaryHalfGcd**.
**Input**: $P, Q \in \mathbb{Z}$ with $0 = \nu(P) < \nu(Q)$, and $k \in \mathbb{N}$
**Output**: a $2 \times 2$ integer matrix $R$, $j \in \mathbb{N}$, and $P', Q'$ such that
    ${}^t[P', Q'] = 2^{-j} R \cdot {}^t [P, Q]$ with $\nu(P') \le k < \nu(Q')$
$m \leftarrow \nu(Q)$,   $d \leftarrow \lfloor k/2 \rfloor$
**if** $k < m$ **then** return $R = \mathrm{Id}, j = 0, P' = P, Q' = Q$
decompose $P$ into $P_1 2^{2d+1} + P_0$, same **for** $Q$
$R, j_1, P'_0, Q'_0 \leftarrow$ BinaryHalfGcd($P_0, Q_0, d$)
$P' \leftarrow (R_{1,1} P_1 + R_{1,2} Q_1) 2^{2d+1-2j_1} + P'_0$
$Q' \leftarrow (R_{2,1} P_1 + R_{2,2} Q_1) 2^{2d+1-2j_1} + Q'_0$
$m \leftarrow \nu(Q')$, **if** $k < j_1 + m$ **then** return $R, j_1, P', Q'$
$q \leftarrow$ BinaryDivide($P', Q'$)
$P' \leftarrow P' + q 2^{-m} Q'$,   $d' \leftarrow k - (j_1 + m)$
$(P', Q') \leftarrow (2^{-m} P', 2^{-m} Q')$
decompose $P'$ into $P_3 2^{2d'+1} + P_2$, same **for** $Q'$
$S, j_2, P'_2, Q'_2 \leftarrow$ BinaryHalfGcd($P_2, Q_2, d'$)
$(P'', Q'') \leftarrow ([S_{1,1} P_3 + S_{1,2} Q_1] 2^{2d'+1-2j_2} + P'_2, [S_{2,1} P_3 + S_{2,2} Q_3] 2^{2d'+1-2j_2} + Q'_2)$
Return $S \cdot [0, 2^m; 2^m, q] \cdot R, j_1 + m + j_2, Q'', P''$.

Algorithm **BinaryDivide**.
**Input**: $P, Q \in \mathbb{Z}$ with $0 = \nu(P) < \nu(Q) = j$
**Output**: $|q| < 2^j$ such that $\nu(Q) < \nu(P + q 2^{-j} Q)$
$Q' \leftarrow 2^{-j} Q$
$q \leftarrow -P/Q' \bmod 2^{j+1}$
**if** $q < 2^j$ **then** return $q$ **else** return $q - 2^{j+1}$
```

Table 1.3: A subquadratic binary gcd algorithm.

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, naïve 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 — think of 2 or a power of 2 — and externally in base B — for example 10 or 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 therefore assume that β and B are not commensurable from now on.

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 only in base β.

1.7.1 Quadratic Algorithms

The following two algorithms respectively read and write n-word integers, with a complexity of $O(n^2)$ in both cases.

```
Algorithm **IntegerInput**.
**Input**: a string $S = s_{m-1} \ldots s_1 s_0$ **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**
    $A \leftarrow BA + \mathrm{val}(s_i)$
Return $A$.
```

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 $S = S_{\mathrm{hi}} \,||\, S_{\mathrm{lo}}$ where S_{lo} has k digits in base B, then

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

where $\mathrm{Input}(S, B)$ is the value obtained when reading the string S in the external base B. The following algorithm shows a possible way to implement

```
Algorithm IntegerOutput.
Input: $A = \sum_0^{n-1} a_i \beta^i$ of the number represented by $S$
Output: a string $S$ of characters, representing $A$ in base $B$
$m \leftarrow 0$
while $A \neq 0$
    $s_m \leftarrow \mathrm{char}(A \bmod B)$
    $A \leftarrow A \operatorname{div} B$
    $m \leftarrow m + 1$
Return $S = s_{m-1} \ldots s_1 s_0$.
```

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

this: If the output A has n words, algorithm **IntegerInput** has complexity $O(M(n) \log n)$, more precisely $\sim \frac{1}{2} M(n/2) \log_2 n$ for n a power of two (see Ex. 1.9.20).

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

If the input A has n words, algorithm **IntegerOutput** has complexity $O(M(n) \log n)$, more precisely $\sim \frac{1}{2} D(n/2) \log_2 n$ for n a power of two, 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

```
Algorithm **IntegerOutput**.
**Input**: $A = \sum_0^{n-1} a_i \beta^i$ **of** the number represented by $S$
**Output**: a string $S$ **of** characters, representing $A$ **in** base $B$
**if** $A < B$ **then** char($A$)
**else**
    find $k$ such that $B^{2k-2} \le A < B^{2k}$
    $(Q, R) \leftarrow \mathrm{DivRem}(A, B^k)$
    **IntegerOutput**($Q$)||**IntegerOutput**($R$)
```

output may thus be 2 to 5 times slower than integer input; see however Ex. 1.9.21.

1.8 Notes and further references

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. 1.9.1.

A very good description of Toom-Cook algorithms can be found in [24, Section 9.5.1], in particular how to symbolically generate the evaluation and interpolation formulæ.

The exact division algorithm starting from least significant bits is due to Jebelean [31], who also invented with Krandick the "bidirectional" algorithm [37]. The Karp-Markstein trick to speed up Newton's iteration (or Hensel lifting over p-adic numbers) is described in [33]. The "recursive division" in §1.4.3 is from [21], although previous but not-so-detailed ideas can be found in [43] and [30].

The square root algorithm in §1.5.1 was proven in [8].

The binary gcd has been analysed by Brent [12, 15], Knuth [35, 36], Maze [41] and Vallée [59]. The double-digit gcd (which should be called double-word gcd instead) is due to Jebelean [32]. Sorenson's k-ary reduction is due to Sorenson [54], and was improved and implemented in GNU MP by Weber, who also invented algorithm `ReducedRatMod` [62]. The first subquadratic gcd algorithm was published by Knuth [34], but his analysis was suboptimal (he gave $O(n(\log n)^5(\log\log n)))$, and the correct complexity was given by Schönhage [50]: 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 [64]. The subquadratic binary gcd given here is due to Stehlé and Zimmermann [55].

1.9 Exercises

Exercise 1.9.1 [Hanrot] Prove that the number $K(n)$ of word products in Karatsuba's algorithm as defined in Th. 1.3.2 is non-decreasing for $n_0 = 2$ (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 $2^7 \le n \le 2^{10}$, and find experimentally where the maximum appears.

Exercise 1.9.2 [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 1.9.3 [Maeder [40]] Show that an auxiliary memory of $2n + 2\lfloor \log_2 n \rfloor - 2$ words is enough to implement Karatsuba's algorithm in-place.

Exercise 1.9.4 [Quercia, McLaughlin] Show that Algorithm `KaratsubaMultiply` can be implemented with only $\sim \frac{7}{2}n$ additions/subtractions. [Hint: decompose C_0, C_1 and C_2 in two parts.]

Exercise 1.9.5 Design an in-place version of Algorithm `KaratsubaMultiply` (see Ex. 1.9.3) that accumulates the result in $c_0, \ldots, c_{2n-1}$, and returns a carry bit.

Exercise 1.9.6 [Vuillemin [61]] 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 1.9.7 [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 1.9.8 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 1.9.9 For multiplication of two numbers of size kn and n, with $k > 1$ integer, show that the trivial strategy which performs k multiplications, each $n \times n$, is not always the best possible.

Exercise 1.9.10 [Hanrot] In Karatsuba's algorithm, instead of splitting the operands in high and low parts, one can split them in odd and even part. Considering the inputs as polynomials $A(\beta)$ and $B(\beta)$, this corresponds to writing $A(t) = A_0(t^2) + tA_1(t^2)$. This is known as the "odd-even" scheme [29]. Design an algorithm `UnbalancedKaratsuba` using that scheme. Show that its complexity satisfies $K(m, n) = 2K(\lceil m/2\rceil, \lceil n/2\rceil) + K(\lfloor m/2\rfloor, \lfloor n/2\rfloor)$.

Exercise 1.9.11 [Karatsuba, Zuras [65]] Assuming the multiplication has superlinear cost, show that the speedup of squaring with respect to multiplication can not exceed 2.

Now we go from a multiplication algorithm of cost cn^α to Toom-Cook r-way; get an expression for the threshold n_0, assuming Toom-Cook cost has a 2nd order term in kn. See how this threshold evolves when c is replaced by another constant, in particular show that this threshold increases for squaring ($c' < c$). Assuming Toom-Cook r-way has cost ln^β for multiplication, and $l'n^\beta$ for squaring, obtain a closed-form expression for the ratio l'/l, in terms of c, c', α, β.

Exercise 1.9.12 [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 1.9.13 In algorithm `BasecaseDivRem` (§1.4.1), prove that $q_j^* \le \beta + 1$. Can this bound be reached? In the case $q_j^* \ge \beta$, prove that the while-loop at steps 9-11 is executed at most once.

Prove that the same holds for Svoboda's algorithm, i.e. that $A \ge 0$ after step 11.

Exercise 1.9.14 [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 1.9.15 In Algorithm **RecursiveDivRem**, find inputs that require 1, 2, 3 or 4 corrections [hint: consider $\beta = 2$]. Prove that when $n = m$ and $A < \beta^m(B+1)$, at most two corrections occur.

Exercise 1.9.16 Find the asymptotic complexity of Algorithm **RecursiveDivRem** in the FFT range.

Exercise 1.9.17 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 1.9.18 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 1.9.19 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 §exactdiv. Prove that your algorithm is correct.

Exercise 1.9.20 Find the asymptotic complexity $T(n)$ of Algorithm `IntegerInput` for $n = 2^k$ (§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 1.9.21 Show that asymptotically, the output routine can be made as fast as the input routine `IntegerInput`. [Hint: use Bernstein's scaled remainder tree and the middle product.] Experiment with it on your favorite multiple-precision software.

Exercise 1.9.22 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 1.9.23 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 $O(M(n))$?

Chapter 2

The FFT, Modular Arithmetic and Finite Fields (outline)

2.1 Representation

2.1.1 Classical Representations

Non-negative, symmetric

2.1.2 Montgomery's Representation

2.1.3 MSB vs LSB Algorithms

Many classical (most significant bits first) algorithms have a p-adic (least significant bit first) equivalent:

classical (MSB)	p-adic (LSB)
Euclidean division	Montgomery reduction
Svoboda's algorithm	Montgomery-Svoboda
Euclidean gcd	2-adic gcd
Newton's iteration	Hensel lifting

2.1.4 Residue Number System

CRT, parallel/distributed algorithms.

2.1.5 Link with polynomials

Modular arithmetic on polynomials.

2.2 Multiplication

2.2.1 Barrett's Algorithm

Barrett's algorithm [4] is interesting when many divisions have to be made with the same divisor; this is in particular 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 one. A small number of corrections suffice to convert those approximations into exact values.

```
Algorithm **BarrettDivRem**.
**Input**: integers $A$, $B$ with $0 \le A < 2^n B$, $2^{n-1} < B < 2^n$.
**Output**: quotient $Q$ and remainder $R$ **of** $A$ divided by $B$.
$I \leftarrow \lfloor 2^{2n}/B \rfloor$ [precomputation]
$Q' \leftarrow \lfloor A_1 I/2^n \rfloor$ where $A = A_1 2^n + A_0$ with $0 \le A_0 < 2^n$
$R' \leftarrow A - Q'B$
**while** $R' \ge B$ **do**
    $(Q', R') \leftarrow (Q' + 1, R' - B)$
Return $(Q', R')$.
```

Theorem 2.2.1 *Algorithm **BarrettDivRem** is correct.*

Proof Since $2^{n-1} < B < 2^n$, we have $2^n < 2^{2n}/B < 2^{n+1}$, thus $2^n \le I < 2^{n+1}$. We have $Q' \le A_1 I/2^n \le A_1/2^n 2^{2n}/B \le A_1 2^n/B \le A/B$. This ensures that R' is nonnegative. Now $I > 2^{2n}/B - 1$, which gives

$$BI > 2^{2n} - B. \tag{2.1}$$

Similarly, $Q' > A_1 I/2^n - 1$ gives

$$2^n Q' > A_1 I - 2^n. \tag{2.2}$$

This gives $2^n Q' B > A_1 IB - 2^n B > A_1(2^{2n} - B) - 2^n B = 2^n A - 2^n A_0 - B(2^n + A_1) > 2^n A - 42^n B$ since $A_0 < 2^n < 2B$ and $A_1 \le B$. We conclude that $2^n A < 2^n Q'(B + 4)$, 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 $I = 113$, $A_1 = 30$, $Q' = 52$, $R' = 108 = 3B$.

Remark: the multiplications at steps 5 and 6 may be replaced by short products, that of step 5 by a high short product, and that of step 6 by a low short product.

2.2.2 Montgomery's Algorithm

```
1 Algorithm REDC.
2 Input: $0 \le C < \beta^{2n}$, $N$, $\mu \leftarrow -N^{-1} \bmod \beta$
3 Output: $0 \le R < \beta^n$ such that $R = C\beta^{-n} \bmod N$
4 Assume $C$ decomposes into $\sum_0^{2n-1} c_i\beta^i$ where the $c_i$ evolve
5 for $i$ from 0 to $n-1$ do
6     $q \leftarrow \mu c_i \bmod \beta$
7     $C \leftarrow C + qN\beta^i$
8 $R \leftarrow C\beta^{-n}$
9 if $R \ge \beta^n$ then return $R - N$ else return $R$.
```

Theorem 2.2.2 *Algorithm **REDC** is correct.*

Proof Assume that for a given i, we have $c_0 = \ldots = c_{i-1} = 0$ when entering step 6. Then since $q = c_i/N \bmod \beta$, we have $C + qN\beta^i = 0 \bmod \beta^{i+1}$ 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 well defined at step 8.

Still at step 8, 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 $R - N < \beta^n$. □

Remark: a subquadratic version of REDC is obtained by taking $n = 1$, and considering β as a "giant base". This is exactly the 2-adic counterpart of Barrett's subquadratic algorithm: step 6 can be replaced by a low short product, and step 7 by a high short product.

2.2.3 Special Moduli

$\beta^n \pm 1$, Schönhage-Strassen FFT (including extra primitive root $2^{3n/4} - 2^{n/4}$ modulo $2^n + 1$).

Multiplication modulo $a \pm b$ where a, b are highly composite.

2.3 Division/Inversion

Link to extended GCD (Ch. 1) or Fermat.

Describe here Hensel lifting for inversion mod p^k (link with division by a constant in §1.4.7). Cite paper of Shanks-Vuillemin for division mod β^n.

2.3.1 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 $M(n)\log n$, but also for smaller numbers. When several inversions are to be performed modulo the same number, the following algorithm is usually faster:

```
Algorithm MultipleInversion.
**Input**: residues $x_1, \ldots, x_k$ modulo $n$
**Output**: $y_1 = 1/x_1, \ldots, y_k = 1/x_k$ modulo $n$
$z_1 \leftarrow x_1$
for $i$ from 2 to $k$ do
    $z_i \leftarrow z_{i-1} x_i (\mathrm{mod} n)$
$q \leftarrow 1/z_k (\mathrm{mod} n)$
for $i$ from $k$ downto 2 do
    $y_i \leftarrow q z_{i-1} (\mathrm{mod} n)$
    $q \leftarrow q x_i (\mathrm{mod} n)$
$y_1 \leftarrow q$
```

Proof We have $z_i = x_1 x_2 \ldots x_i (\mathrm{mod} n)$, thus at the beginning of step i, $q = (x_1 \ldots x_i)^{-1} (\mathrm{mod} n)$, which indeed gives $y_i = 1/x_i (\mathrm{mod} n)$. □

This algorithm uses only one modular inversion, and $3(k-1)$ modular multiplications. Thus it is faster when an inversion is at least three times as expensive as a product. Fig. 2.1 shows a recursive variant of the algorithm, with the same numbers 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.

A dual case is when the number to invert is invariant, and we want to compute $1/x \bmod n_1, \ldots, 1/x \bmod n_k$. A similar algorithm works as follows: first compute $N = n_1 \ldots n_k$ using a product tree like that in Fig. 2.1. Then compute $1/x \bmod N$, and go down the tree, while reducing the residue at

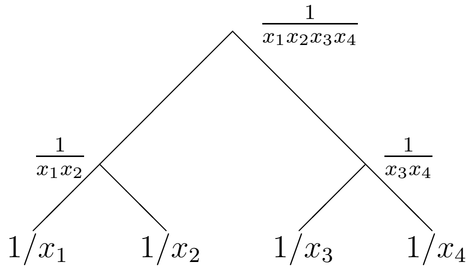


Figure 2.1: A recursive variant of Algorithm `MultipleInversion`.

each node. The main difference is that here, the residues grow while going up the tree, thus even if it performs only one modular inversion, this method might be slower for large k.

2.4 Exponentiation

Link to HAC, Ch. 14.

2.5 Conversion

integer from/to modular (CRT, FFT), 3-primes variant of FFT.

2.6 Finite Fields

FFT in finite fields. Generalization of above, trinomials, ...

2.7 Applications of FFT

Applications and other variants of FFT.

2.8 Exercises

Exercise 2.8.1 Assuming you have a 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.

2.9 Notes and further references

Applications: Pollard ρ, ECM, roots, ...

Algorithm `MultipleInversion` is due to Montgomery [44].

Chapter 3

Floating-Point Arithmetic (outline)

3.1 Introduction

3.1.1 Representation

fraction/mantissa, exponent, sign, position of the point

IEEE 754/854: special values (infinities, NaN), signed zero, rounding modes (±∞, to zero, to nearest, away).

Binary vs decimal representation.

Implicit vs explicit leading bit.

Links to other possible representations. In her PhD [42], Valérie Ménissier-Morain discusses three different representations for real numbers (Ch. V): continued fractions, redundant representation, and the classical non-redundant representation. She also considers the theory of computable reals, their representation by B-adic numbers, and the computation of algebraic or transcendental functions (Ch. III).

3.1.2 Local vs Global Precision

global precision vs one precision for each variable

3.1.3 Link to Integers

Using f-p numbers for integer word operations, and for integer arbitrary-precision (expansions: cf Priest [48]). Also use of (complex) floating-point numbers for FFT multiplication (cf Knuth vol 2, and error analysis in Colin Percival's paper [47]).

3.1.4 Error analysis

absolute vs relative vs ulp error
forward vs backward error analysis

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

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

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

3.1.5 Rounding

Assume we want to correctly round to n bits a real number whose binary expansion is $0.1b_1 \ldots b_n b_{n+1} \ldots$ It is enough to know the value 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. The following table shows how to correctly round from r and 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.

r	s	zero	nearest	away
0	0	0	0	0
0	1	0	0	1
1	0	0	0 or 1	1
1	1	0	1	1

However, in general we do not have an infinite expansion, but a finite approximation y of an unknown real value x. The problem is the following: given the approximation y, and a bound on the error $|y - x|$, is it possible to determine the correct rounding of x?

```
Algorithm RoundingPossible.
Input: a f−p number $y = 0.y_1 \ldots y_m$, $y_1 = 1$, a precision $n \le m$,
       an error $\varepsilon = 2^{-k}$, a rounding mode $\circ$
Output: true iff $\circ_n(x)$ can be determined for $|y - x| \le \varepsilon$
If $\circ$ is to nearest, then $n \leftarrow n + 1$
If $k \le n$ then return false
If $\circ$ is to nearest and $y_n = y_{n+1}$ then return true
If $y_{n+1} = y_{n+2} = \ldots = y_k$ then return false
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 $k \le n$, then the error on y may change the significand, so 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, it is possible to round correctly. Otherwise it is only possible when $y_{n+1}, y_{n+2}, \ldots, y_k$ are not all identical. □

The Double Rounding Problem

This problem does not occur with all rounding modes (Ex. 3.5.1).

3.1.6 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 $m = n + k$ bits, with k relatively small. Several strategies are possible when this first approximation y is not accurate enough, or too close to a rounding boundary.

3.2 Addition/Subtraction/Comparison

Leading Zero Anticipation and Detection.

Sterbenz Theorem.

Floating-point addition and subtraction are more difficult to implement than integer addition/subtraction for two reasons:

- scaling due to the exponents necessitates shifting the mantissas 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 right to left, one may have to look at arbitrarily low bits to guarantee correct rounding.

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. The case of one or more zero operands is treated separately. In the description below we assume that both operands are nonzero.

3.2.1 Floating-Point Addition

The following algorithm adds two binary floating-point numbers b and c. 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, $b \geq c > 0$, $2^{n-1} \leq b < 2^n$ and $2^{m-1} \leq c < 2^m$. We also 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**: $b$ and $c$ two floating-point numbers, a precision $n$,
3  and a rounding mode $\circ$.
4  **Output**: a f-p number $a \cdot 2^e$ of precision $n$, equal to $\circ(b+c)$.
5  Split $b$ into $b_h + b_l$ where $b_h$ contains the $n$ most significant
6        bits of $b$.
7  Split $c$ into $c_h + c_l$ where $c_h$ contains the $\max(0, m)$ most
8        significant bits of $c$.
9  $a_h \leftarrow b_h + c_h, \quad e \leftarrow 0$
10 $(c, r, s) \leftarrow b_l + c_l$.
11 $a \leftarrow a_h + c + \mathrm{round}(\circ, r, s)$
12 if $a \geq 2^n$ then
13     $a \leftarrow \mathrm{round2}(\circ, a \bmod 2, t)$
14     $e \leftarrow e + 1$
15     if $a = 2^n$ then $(a, e) \leftarrow (a/2, e+1)$
16 Return $(a, e)$.
```

The values of $\mathrm{round}(\circ, r, s)$ and $\mathrm{round2}(\circ, a \bmod 2, t)$ are given in Table 3.1. At step 10, 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. For rounding to nearest, t is a ternary value, which is respectively positive, zero, or negative when a is larger than, equal to, or smaller than the exact sum $b + c$.

$\circ$	r	s	$\mathrm{round}(\circ, r, s)$	t
zero	any	any	0	
away	r	s	0 if $r = s = 0$, 1 otherwise	
nearest	0	any	0	$-s$
nearest	1	0	0/1 (even rounding)	$-1/1$
nearest	1	1	1	1

$\circ$	$a \bmod 2$	t	$\mathrm{round2}(\circ, a \bmod 2, t)$
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	$(a-t)/2$

Figure 3.1: Rounding rules for addition.

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

Proof With the assumptions made, 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 11, $2^{n-1} \leq a \leq 2^{n+1}$. 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 it 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) 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 Leading Zero Detection

3.2.3 Floating-Point Subtraction

3.3 Multiplication, Division, Algebraic Functions

Link to chapter 1.

Bounds, extension to different precisions.

Short multiplication/division.

Middle product (cf Newton). Application to argument reduction: Payne and Hanek method.

3.3.1 Multiplication

The exact product of two floating-point numbers $m \cdot 2^e$ and $m' \cdot 2^{e'}$ is $(mm') \cdot 2^{e+e'}$. Therefore, if no underflow or overflow occurs, the problem reduces to the multiplication of the significands m and m'.

```
Algorithm FPmultiply.
Input: $x = m \cdot \beta^e$, $x' = m' \cdot \beta^{e'}$, a precision $n$, a rounding mode $\circ$
Output: $\circ(xx')$ rounded to precision $n$
$e'' \leftarrow e + e'$
$m'' \leftarrow \circ(mm')$ to precision $n$
Return $m'' \cdot \beta^{e+e'}$.
```

The product at step 5 is a *short product*, i.e. a product whose most significand part only is wanted. 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 seems tricky. Lastly, in the FFT range, no better algorithm is known than computing the full product mm'.

Hence our advice is to perform a full product of the m and m', after possibly truncating them to $n+k$ digits if they have more than n digits. This also makes it easier to compute additional digits in case the first rounding fails.

How long can a run of zeros or ones be after the first n bits? The answer is: as long as the longest mantissa minus n bits. Indeed, take an arbitrary output m'' with k ones after the round bit, and consider an arbitrary m of n bits. Then compute $n+k$ bits of m''/m, and let m' be the corresponding mantissa. Since m''/m agrees with m' up to $n+k$ bits, then m'' agrees with mm' up to $n+k$ bits. Even if both mantissas have at most n bits, we can have a run of $n-1$ identical bits: take for example $m = 2^n - 1$ and $m' = 2^{n-1} + 1$.

Error analysis of the short product. Consider two n-word normalized mantissae A and B that we multiply using a short product algorithm:

```
Algorithm ShortProduct.
Input: $A = \sum_{i=0}^{n-1} a_i \beta^i$, $B = \sum_{i=0}^{n-1} b_i \beta^i$
Output: an approximation of $AB$ div $\beta^n$
if $n \le n_0$ then return FullProduct$(A, B)$
choose $k \ge n/2$, $l \leftarrow n - k$
$C_1 \leftarrow$ FullProduct$(A$ div $\beta^l, B$ div $\beta^l, k)$
$C_2 \leftarrow$ ShortProduct$(A$ mod $\beta^l, B$ div $\beta^k, l)$
$C_3 \leftarrow$ ShortProduct$(A$ div $\beta^k, B$ mod $\beta^l, l)$
Return $C_1$ div $\beta^{k-l} + (C_2 + C_3)$ div $\beta^l$.
```

Theorem 3.3.1 *The value C' returned by Algorithm* `ShortProduct` *differs from the exact short product $C = AB \operatorname{div} \beta^n$ by at most $n-1$, more precisely*

$$C' \leq C \leq C' + (n-1).$$

Proof We first prove by induction that the algorithm computes all products $a_i b_j$ with $i + j \geq n - 1$. For $n \leq n_0$ this is trivial since all products are computed. Now assume $n > n_0$, and consider a product $a_i b_j$ with $i+j \geq n-1$. Three cases can occur:

1. $i, j \geq l$, then $a_i b_j$ is computed in C_1;

2. $i < l$, which implies $j \geq k$ since $i + j \geq n - 1$; the index of a_i in $A \bmod \beta^l$ is $i' = i$, and that of b_j in B div β^k is $j' = j - k$, thus $i' + j' = i + j - k \geq (n-1) - k = l - 1$, which proves that $a_i b_j$ is computed in C_2;

3. similarly, $j < l$ implies $i \geq k$, and $a_i b_j$ is computed in C_3.

The case $i, j < l$ cannot occur since we would have $i + j \leq n - 2$ (recall that $l \leq n/2$).

The neglected part with respect to the full product AB is thus at most $\sum_{i+j \leq n-2} a_i b_j \beta^{i+j} \leq \sum_{i+j \leq n-2} (\beta - 1)^2 \beta^{i+j} = (n-1)\beta^n - n\beta^{n-1} + 1 \leq (n-1)\beta^n$.

The product D' — before division by β^n — computed by the algorithm thus satisfies $D' \leq AB \leq D' + (n-1)\beta^n$, from which the theorem follows, after division by β^n. □

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

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

Remark 2: assuming β is a power of two, if A and B are normalized, i.e. $\beta/2 \leq a_{n-1}, b_{n-1} < \beta$, then $C' \geq \beta^n/4$, and the error is bounded by $2(n-1)$ ulps.

Integer Multiplication via Complex Floating-Point FFT

To multiply n-bit integers, the algorithms using Fast Fourier Transform (FFT for short) fall into two classes: those using number theoretical properties, and those based on complex floating-point computations. The latter, while not giving the best known asymptotic complexity $O(n \log n \log\log n)$ of the former, have a good practical behaviour, because they take advantage of the efficiency of floating-point hardware. The drawback of the complex FFT 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 factorization, a small probability of error is acceptable because the result (a purported factorization) can easily be checked and discarded if incorrect.

Until very recently, all rigorous error analyses of complex FFT gave very pessimistic bounds. The following theorem from Percival [47] changes this situation:

Theorem 3.3.2 *[47] The FFT allows computation of the cyclic convolution $z = x * y$ of two vectors of length $N = 2^n$ of complex values such that*

$$|z' - z|_\infty < |x| \cdot |y| \cdot ((1+\varepsilon)^{3n}(1+\varepsilon\sqrt{5})^{3n+1}(1+\eta)^{3n} - 1), \tag{3.1}$$

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, $\eta > |(w^k)' - (w^k)|$, $0 \le k < N$, $w = e^{\frac{2\pi i}{N}}$, and $(\cdot)'$ refers to the computed (stored) value of $\cdot$ for each expression.

The proof given in Percival's paper [47] is incorrect, but we have a correct proof (see [19] and Ex. 3.5.3). For the double precision format of IEEE 754, with rounding to nearest, we have $\varepsilon = 2^{-53}$, and if the w^k are correctly rounded, we can take $\eta = \varepsilon/\sqrt{2}$. For a fixed FFT size $N = 2^n$, Eq. (3.1) 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 3.1).

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 3.5.4.

n	bits/float	m-bit mult.
1	25	25
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
11	18	18432
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 3.1: Maximal number of bits per floating-point number, and maximal m for a plain $m \times m$ bit integer product, for a given FFT size 2^n, with signed coefficients, and 53-bit floating-point mantissa.

3.3.2 Reciprocal

The following algorithm computes a floating-point inverse with time complexity $2M(n)$, considering multiplication as a black-box.

```
Algorithm **Invert**.
**Input**: $1 \le A \le 2$, a $p$-bit f-p number $x$ with $0 \le 1/A - x \le 2^{1-p}$
**Output**: a $p'$-bit f-p number $x'$ with $p' = 2p - 5$, $0 \le 1/A - x' \le 2^{1-p'}$
$v \leftarrow \circ(A)$ [$2p$ bits, rounded up]
$w \leftarrow vx$ [$3p$ bits, exact]
**if** $w \ge 1$ **then** return $x$
$y \leftarrow \circ(1 - w)$ [$p$ bits, towards zero]
$z \leftarrow \circ(xy)$ [$p$ bits, towards zero]
$x' \leftarrow \circ(x + z)$ [$p'$ bits, towards zero]
```

Theorem 3.3.3 *Algorithm **Invert** is correct, and yields an inversion algorithm of complexity* $2M(n)$.

Proof First suppose we have no roundoff error. Newton's iteration for the inverse of A is $x_{k+1} = x_k + x_k(1 - Ax_k)$. If we denote $\varepsilon_k := x_k - 1/A$, we have $\varepsilon_{k+1} = -A\varepsilon_k^2$. This shows that if $x_0 \le 1/A$, then all x_j's are less or equal to $1/A$.

Now consider rounding errors. The hypothesis $0 \leq 1/A - x \leq 2^{1-p}$ can be written $0 \leq 1 - Ax \leq 2^{2-p}$ since $A \leq 2$.

If $w \geq 1$ at step 6, we have $Ax \leq 1 \leq vx$; since $v - A \leq 2^{1-2p}$, this shows that $1 - Ax \leq vx - Ax \leq 2^{1-2p}$, thus $1/A - x \leq 2^{1-2p} \leq 2^{1-p'}$. Otherwise, we can write $v = A + \varepsilon_{1-2p}$, where ε_i denotes a positive quantity less that 2^i. Similarly, $y = 1 - w - \varepsilon_{2-2p}$, $z = xy - \varepsilon'_{2-2p}$, and $x' = x + z - \varepsilon_{5-2p}$. This gives $x' = x + x(1 - Ax) - \varepsilon_{1-2p}x^2 - \varepsilon_{2-2p}x - \varepsilon'_{2-2p} - \varepsilon_{5-2p}$. Since the difference between $1/A$ and $x + x(1 - Ax)$ is bounded by $A(x - 1/A)^2 \leq 2^{3-2p}$, we get

$$|x' - 1/A| \leq 2^{3-2p} + 2^{1-2p} + 2 \cdot 2^{2-2p} + 2^{5-2p} = 50 \cdot 2^{-2p} \leq 2^{6-2p} = 2^{1-p'}.$$

The complexity bound of $2M(n)$ is obtained using a "negacyclic convolution" for $w = vx$, assuming we use a multiplication algorithm that computes mod $2^n \pm 1$. Indeed, the product vx has $3p$ bits, but we know the p most significant bits give 1, so it can be obtained by multiplication mod $2^{2p} \pm 1$. □

If we keep the FFT-transform of x from step 5 to step 8, we can save $\frac{1}{3}M(n)$ — assuming the term-to-term products have negligible cost —, which gives $\frac{5}{3}M(n)$ as noticed by Bernstein, who also proposes a "messy" $\frac{3}{2}M(n)$ algorithm.
Remark: Schönhage's $1.5M(n)$ algorithm is better [51].

3.3.3 Division

Theorem 3.3.4 *Assume we divide an m-bit floating-point number by an n-bit floating-point number, with $m < 2n$. Then the (infinite) binary expansion of the quotient can have at most n consecutive zeros after its first n bits, if not exact.*

Proof Without loss of generality, we can assume that the nth significand bit of the quotient q corresponds to 1, and similarly for the divisor d. If the quotient has more than n consecutive zeros, we can write it $q = q_1 + 2^{-n}q_0$ with q_1 an n-bit integer and either $q_0 = 0$ if the division is exact, or an infinite expansion $0 < q_0 < 1$. Thus $qd = q_1d + 2^{-n}q_0d$, where q_1d is an integer of $2n - 1$ or $2n$ bits, and $0 < 2^{-n}q_0d < 1$. This implies that qd has at least $2n$ bits. □

```
Algorithm Divide (h,f,2n).
1. Compute $g_0 = Invert(f,n)$ [2M(n)]
2. $q_0 = h g_0$ truncated to $n$ bits [M(n)]
3. $e = MP(q_0, f)$ [M(n)]
4. $q = q_0 - g_0 e$ [M(n)]
```

The total cost is therefore $5M(n)$ for precision $2n$, or $\frac{5}{2}M(n)$ for precision n. This is the sharpest bound known. (Bernstein replaces $\frac{5}{2}$ with $\frac{7}{3}$ or even $\frac{13}{6}$ but with special assumptions.)

As for the reciprocal, if we cache FFT transforms, we get $\frac{5}{3}M(n)$ for step 1, and a further gain of $\frac{1}{3}M(n)$ by saving the transform of g_0 between steps 2 and 4, which gives $\frac{25}{12}M(n) = 2.0833\cdots M(n)$.

Lemma 3.3.1 *Let A and B be two positive integers, and $\beta \geq 2$ a positive integer. Let $Q = \lfloor A/B \rfloor$, $A_1 = \lfloor A/\beta \rfloor$, $B_1 = \lfloor B/\beta \rfloor$, $Q_1 = \lfloor A_1/B_1 \rfloor$. Assuming $Q_1 \leq 2B_1$, then*

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

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

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

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

For the second inequality,

$$\begin{aligned} A/B &\geq \frac{A_1\beta}{B_1\beta + (\beta - 1)} = (Q_1 B_1 + R_1)\beta B_1\beta + (\beta - 1) \\ &= Q_1 + \frac{R_1\beta - Q_1(\beta - 1)}{B_1\beta + (\beta - 1)} \geq Q_1 - \frac{Q_1}{B_1}. \end{aligned}$$

□

This lemma is useful when replacing β by β^n.

```
Algorithm ShortDivision.
Input: $A = \sum_{i=0}^{2n-1} a_i\beta^i$, $B = \sum_{i=0}^{n-1} b_i\beta^i$ with $\beta/2 \le b_{n-1} < \beta$
Output: an approximation of $A/B$
if $n \le n_0$ then return ExactQuotient($A, B$)
choose $k \ge n/2$, $l \leftarrow n - k$
$(A_1, A_0) \leftarrow (A \text{ div } \beta^{2l}, A \bmod \beta^{2l})$
$(B_1, B_0) \leftarrow (B \text{ div } \beta^{l}, B \bmod \beta^{l})$
$(Q_1, R_1) \leftarrow$ DivRem($A_1, B_1$)
$A' \leftarrow R_1\beta^{2l} + A_0 - Q_1B_0\beta^l$
$Q_0 \leftarrow$ ShortDivision($A' \text{ div } \beta^k, B \text{ div } \beta^k$)
Return $Q_1\beta^l + Q_0$.
```

Theorem 3.3.5 *The approximate quotient Q' returned by* ***ShortDivision*** *differs at most by n from the exact quotient $Q = A/B$, more precisely:*

$$Q \le Q' \le Q + 2\log_2 n.$$

Proof If $n \le n_0$, $Q = Q'$ so the statement holds. Assume $n > n_0$. We have $A = A_1\beta^{2l} + A_0$ and $B = B_1\beta^l + B_0$, thus since $A_1 = Q_1B_1 + R_1$, $A = (Q_1B_1 + R_1)\beta^{2l} + A_0 = A' + Q_1B\beta^l$. Let $A' = A'_1\beta^k + A'_0$, and $B = B'_1\beta^k + B'_0$. We have seen (Chapter 1) that 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 $Q_0 \ge A'/B$ too. Since $A/B = Q_1\beta^l + A'/B$, this proves that $Q' \ge Q$.

Now by induction $Q_0 \le \frac{A'_1}{B'_1} + 2\log_2 l$, and $\frac{A'_1}{B'_1} \le A'/B + 2$ (see Chapter 1), so $Q_0 \le A'/B + 2\log_2 n$, and $Q' \le A/B + 2\log_2 n$. □

Barrett's division

We already mentioned Barrett's algorithm for division in §2.2.1. Assume we want to divide a by b of n bits, assuming the quotient has exactly n bits. Barrett's algorithm is as follows:

0. Precompute the inverse i of b on n bits [nearest]
1. $q \leftarrow \circ(ai)$ [nearest]
2. $r \leftarrow a - bq$.

Lemma 3.3.2 *At step 2, we have $|a - bq| \le \frac{3}{2}|b|$.*

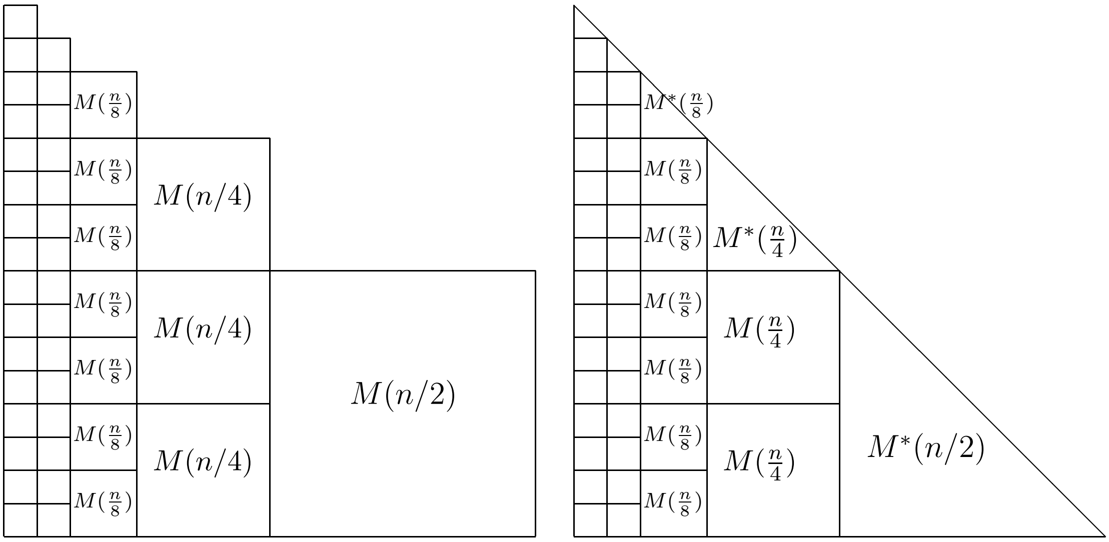


Figure 3.2: Divide and conquer short division: a graphical view. Left: with plain multiplication; right: with short multiplication. See also Fig. 1.1.

Proof We can assume without loss of generality that a is an integer $< 2^{2n}$, that b is an integer $2^{n-1} \leq b < 2^n$. We have $i = 1/b + \varepsilon$ with $|\varepsilon| \leq \frac{1}{2}\mathrm{ulp}(1/b) \leq 2^{-2n}$. And $q = ai + \varepsilon'$ with $|\varepsilon'| \leq \frac{1}{2}\mathrm{ulp}(q) \leq \frac{1}{2}$ since $q < 2^n$. Thus $q = a(1/b + \varepsilon) + \varepsilon' = a/b + a\varepsilon + \varepsilon'$, and $|bq - a| = |b||a\varepsilon + \varepsilon'| \leq \frac{3}{2}|b|$. □

As a consequence, after at most one correction, q is correct (for rounding to nearest).

Remark: if $a < 2^{2n-1}$, then the bound becomes $|a - bq| \leq |b|$, thus r is exact.

3.3.4 Square Root

The following algorithm assumes an integer significand m, and a directed rounding mode.

```
Algorithm FPSqrt.
Input: $x = m \cdot 2^e$, a target precision $n$, a rounding mode $\circ$
Output: $y = \circ_n(\sqrt{x})$
If $e$ is odd, $(m', f) \leftarrow (2m, e-1)$, else $(m', f) \leftarrow (m, e)$
Write $m' := m_1 2^{2k} + m_0$, $m_1$ having $2n$ or $2n-1$ bits, $0 \le m_0 < 2^{2k}$
$(s, r) \leftarrow$ SqrtRem$(m_1)$
If round to zero or down or $r = m_0 = 0$, return $s \cdot 2^{k+f/2}$
else return $(s+1) \cdot 2^{k+f/2}$.
```

Theorem 3.3.6 *Algorithm* `FPSqrt` *returns the correctly-rounded square root of x.*

Proof Since m_1 has $2n$ or $2n-1$ bits, s has exactly n bits, and we have $x \ge s^2 2^{2k+f}$, thus $\sqrt{x} \ge s 2^{k+f/2}$. On the other hand, `SqrtRem` ensures that $r \le 2s$, thus $x 2^{-f} = (s^2+r)2^{2k}+m_0 < s^2+r+1 \le (s+1)^2$. Since $y := s \cdot 2^{k+e/2}$ and $y^+ = (s+1) \cdot 2^{k+e/2}$ 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.

To compute a square root on $2n$ bits, the asymptotically best known method is the following:

```
Algorithm SquareRoot(h, 2n).
1. Compute $g_0 = h^{-1/2}$ on $n$ bits.
2. $f_0 = h g_0$ truncated to $n$ bits [M(n)]
3. $e = h - f_0^2$ [1/2M(n)]
4. $f = f_0 + \frac{g_0}{2} e$ [M(n)]
```

Since the n most significant bits of f_0^2 are known to match those of h in step 3, we can do a transform mod $x^n - 1$. The inverse square root g_0 is computed via Newton's iteration:

```
Algorithm InverseSquareRoot(h, 2n).
1. $g_0 = InverseSquareRoot(h,n)$
2. $g = g_0 + \frac{1}{2}(g_0 - h g_0^3)$
```

At step 2, hg_0^3 has $5n$ bits, and we want only bits n to $2n$ — the low n bits are known to match those of g_0 —, thus we can compute $hg_0^3 \bmod x^{4n} - 1$ which costs $2M(n)$.

The total cost for $SquareRoot(2n)$ is thus $4.5M(n)$, and thus $2.25M(n)$ for $SquareRoot(n)$.

3.4 Conversion

Cf Chapter 1.

3.4.1 Floating-Point Output

Consider the problem of printing a floating-point number, represented internally in base b, in another base B. We distinguish here two kinds of floating-point output:

- 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 needs only one digit in base $b = 3$. 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 according to the given rounding mode, to give back the initial number. Here the number of printed digits may depend on the input number. This property is useful when storing data in a file, while guaranteeing that reading them 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 $|x - X| < \mathrm{ulp}(x)$ (we consider here directed rounding).

```
1  Algorithm **PrintFixed**.
2  **Input**: $f, e, p, b, B, P$ integers, $b^{p-1} \le |f| < b^p$, a rounding mode $\circ$.
3  **Output**: $F, E$ such that $B^{P-1} \le |F| < B^P$, and $X = F \cdot B^{E-P}$ is the
4      closest floating-point number **of** $x = f \cdot b^{e-p}$ according **to** $\circ$.
5  $\lambda \leftarrow \circ(\frac{\log b}{\log B})$
6  $E \leftarrow 1 + \lfloor (e-1)\lambda \rfloor$
7  $q \leftarrow \lceil P/\lambda \rceil$
8  $y \leftarrow xB^{P-E}$ with precision $q$
9  **If** one cannot round $y$ **to** an integer, increase $q$ and **goto** 8.
10 $F \leftarrow \mathrm{Integer}(y, \circ)$.
11 **If** $|F| \ge B^P$ **then** $E \leftarrow E + 1$
12  and **goto** 8.
13 Return $F, E$.
```

We assume here that we have precomputed values of $\lambda_B = \circ(\frac{\log b}{\log B})$. Assuming the input exponent e is bounded, it is possible — see Ex. 3.5.7 — to choose those values precise enough so that

$$E = 1 + \lfloor (e-1)\frac{\log b}{\log B} \rfloor. \tag{3.2}$$

Theorem 3.4.1 *Algorithm* **PrintFixed** *is correct.*

Proof First assume that the algorithm terminates. Eq. (3.2) implies $B^{E-1} \le b^{e-1}$, thus $|x|B^{P-E} \ge B^{P-1}$, which implies that $|F| \ge B^{P-1}$ at step 10. Thus $B^{P-1} \le |F| < B^P$. 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 xB^{P-E}, if not an integer, cannot be arbitrarily close to an integer. If $P - E \ge 0$, let k be the number of bits of B^{P-E}, then xB^{P-E} can be represented exactly with $p+k$ bits. If $P-E < 0$, let $g = B^{E-P}$, of k bits. 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 $|\varepsilon| \ge 1/g \ge 2^{-k}$.

The case $|F| \ge B^P$ at step 12 can occur for two reasons. Either $xB^{P-E} \ge B^P$, thus its rounding also satisfies this inequality; or $xB^{P-E} < B^P$, but its rounding equals B^P (this can only occur for rounding away from zero or to nearest). In the former case one we have $xB^{P-E} \ge B^{P-1}$ at the next step 8, while in the latter case the rounded value F equals B^{P-1} and the algorithm terminates. □

Now return to the free-format output. For a directed rounding mode (resp. rounding to nearest), we want that $|x - X| < \mathrm{ulp}(x)$ (resp. $|x - X| \leq \frac{1}{2}\mathrm{ulp}(x)$) knowing that $|x - X| < \mathrm{ulp}(X)$ (resp. $|x - X| \leq \frac{1}{2}\mathrm{ulp}(X)$). It is easy to see that a sufficient condition is that $\mathrm{ulp}(X) \leq \mathrm{ulp}(x)$, or equivalently $B^{E-P} \leq b^{e-p}$. In summary, 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 sufficient condition becomes:

$$P \geq 1 + p\,\frac{\log b}{\log B}.$$

For example, with $p = 53$, $b = 2$, $B = 10$, this gives $P \geq 17$. As a consequence, if a double-precision floating-point number is printed with at least 17 significant digits, it can be read back without any discrepancy, assuming input and output are performed with correct rounding to nearest.

3.4.2 Floating-Point Input

Assume we have a floating-point number x with a mantissa of n digits in base β, that we convert to x' with n' digits in base β', and then back to x'' with n digits in base β (both conversions with rounding to nearest). How large must be n' with respect to n, β, β' so that $x = x''$? For $\beta = 2, \beta' = 10$, we should have $n' \geq 9$ for single precision ($n = 24$), and $n' \geq 17$ in double precision ($n = 53$).

3.5 Exercises

Exercise 3.5.1 (Kidder,Boldo) The "rounding to odd" mode [10, 38] is defined as follows: in case the exact value is not representable, it rounds to the unique adjacent with an odd mantissa (assuming a binary representation). Prove that if $y = \mathrm{round}(x, p + k, \mathrm{odd})$ and $z = \mathrm{round}(y, p, \mathrm{nearest_even})$, and $k > 1$, then $z = \mathrm{round}(x, p, \mathrm{nearest_even})$, i.e. the double-rounding problem does not occur.

Exercise 3.5.2 Adapt Mulders' short product algorithm [45] to floating-point numbers. In case the first rounding fails, can you compute additional digits without starting again from scratch?

Exercise 3.5.3 (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 $|z - z_0 z_1| \le c 2^{-n} |z_0 z_1|$. What is the best possible c?

Exercise 3.5.4 Show that, if $\eta = O(\varepsilon)$ 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 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 favorite 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 3.5.5 (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 3.5.6 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 $\frac{x}{\sqrt{x^2+y^2}}$ lies in $[-1, 1]$, except in some special case.

Exercise 3.5.7 Show that the computation of E in Algorithm `PrintFixed` is correct as long as there is no integer n such that $|\frac{n}{e-1} \frac{\log B}{\log b} - 1| < \varepsilon$, where ε is the relative precision when computing λ: $\lambda = \frac{\log B}{\log b}(1 + \theta)$ with $|\theta| \le \varepsilon$. For a fixed range of exponents $-e_{\max} \le e \le e_{\max}$, deduce a working precision ε. Application: for $b = 2$, and $e_{\max} = 2^{31}$, compute the required precision for $3 \le B \le 36$.

Exercise 3.5.8 (Lefèvre) The IEEE 754 standard requires binary to decimal conversions to be correctly rounded in the range $m \cdot 10^n$ for $|m| \le 10^{17} - 1$ and $|n| \le 27$ 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` function of your system.

Exercise 3.5.9 Same question as the above, for the decimal to binary conversion, and the `atof` function.

3.6 Notes and further references

The link between usual multiplication and the middle product using trilinear forms was already mentioned by Victor Pan in [46] 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 $\cdots$"

Interval arithmetic (endpoint vs middle/error representation).

Complex arithmetic (pointers).

Fixed-point arithmetic.

Chapter 4

Newton's Method and Unrestricted Algorithms for Elementary and Special Function Evaluation (outline)

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 §4.2.6). 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 (§4.11).

4.2 Newton's Method

4.2.1 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, see [reference-to-be-added].

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

$$f(x_0) = f(\zeta) + (x_0 - \zeta)f'(\zeta) + \frac{(x_0 - \zeta)^2}{2} f"(\xi)$$

for some point ξ in an interval including $\{x_0, \zeta\}$. 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

$$x_{n+1} = x_n - f_n/f'_n \ , \quad n = 0, 1, \ldots \ ,$$

where we have abbreviated $f_n = f(x_n)$ and $f'_n = f'(x_n)$. 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 [58] 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*).

4.2.2 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

$$x_{j+1} = x_j + x_j(1 - x_j^m y)/m \ . \tag{4.1}$$

This iteration converges to $\zeta = y^{-1/m}$ provided the initial approximation x_0 is sufficiently close to ζ. It is perhaps surprising that (4.1) does not involve divisions, except for a division by the integer constant m. Thus, we can easily compute reciprocals (the case $m = 1$) and inverse square roots (the case $m = 2$) by Newton's method. These cases are sufficiently important that we discuss them separately in the following subsections.

4.2.3 Newton's method for reciprocals

Taking $m = 1$ in (4.1), we obtain the iteration

$$x_{j+1} = x_j + x_j(1 - x_j y) \tag{4.2}$$

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 (4.2) by y, we get

$$1 - u_{j+1} = (1 - u_j)(1 + u_j) \ ,$$

which simplifies to

$$u_{j+1} = u_j^2 \; . \tag{4.3}$$

Thus

$$u_j = (u_0)^{2^j} \; . \tag{4.4}$$

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 on the right-hand-side of (4.4).

The iteration (4.2) is sometimes implemented in hardware to compute reciprocals of floating-point numbers, see for example [33]. The sign and exponent of the floating-point number are easily handled, so we can assume that $y \in [0.5, 1.0)$. 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[1].

4.2.4 Newton's method for inverse square roots

Taking $m = 2$ in (4.1), we obtain the iteration

$$x_{j+1} = x_j + x_j(1 - x_j y)/2 \; , \tag{4.5}$$

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[2] iteration

$$x_{j+1} = \frac{1}{2}\left(x_j + \frac{y}{x_j}\right) \tag{4.6}$$

[1] In the case of the infamous *Pentium* `fdiv` *bug* [28], 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.

[2] Heron of Alexandria, *circa* 10–75 AD.

for the square root of y. This requires a division by x_j at iteration j, so it is essentially different from the iteration (4.5). Although both iterations have second-order convergence, we expect (4.5) to be more efficient (we consider such issues in more detail below).

4.2.5 Newton's method for power series

Newton's method can be applied to functions of power series as well as to functions of a real or complex variable. For simplicity we consider power series of the form

$$A(z) = a_0 + a_1 z + a_2 z^2 + \cdots$$

where $a_i \in R$ (or any field of characteristic zero) and $\mathrm{ord}(A) = 0$, *i.e.* $a_0 \neq 0$.

For example, if we replace y in (4.2) by $1 - z$, and take initial approximation $x_0 = 1$, we obtain a quadratically-convergent iteration for the power series

$$(1-z)^{-1} = \sum_{n=0}^{\infty} z^n \ .$$

In the case of power series, "quadratically convergent" means that $\mathrm{ord}(e_j) \to +\infty$ like 2^j. In our example, $u_0 = 1 - x_0 y = z$, so $u_j = z^{2^j}$ and

$$x_j = \frac{1-u_j}{1-z} = \frac{1}{1-z} + O\left(z^{2^j}\right) \ .$$

Another example: if we replace y in (4.5) by $1 - 4z$, and take initial approximation $x_0 = 1$, we obtain a quadratically-convergent iteration for the power series

$$(1-4z)^{-1/2} = \sum_{n=0}^{\infty} \binom{2n}{n} z^n \ .$$

Some operations on power series have no analogue for integers. For example, given a 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

$$\sum_{j=0}^{n} a_j \beta^j \ .$$

For more on Newton's method for power series, we refer to [11, 17, 20, 36].

4.2.6 Newton's method for exp and log

Newton's method to evaluate e^h is $f \leftarrow f_0 + f_0(h - \log f_0)$.

```
Algorithm Improve-Exp.
Input: $h$, $n$, $f_0$ an $n$-bit approximation to $\exp(h)$.
Output: $f :=$ a $2n$-bit approximation to $\exp(h)$.
$g \leftarrow \log f_0$ [computed to $2n$-bit accuracy]
$e \leftarrow h - g$
$f \leftarrow f_0 + f_0 e$
Return $f$.
```

4.3 Argument Reduction

See Brent [16].

4.4 Power Series

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

$$f(x) = \sum_{j=0}^{k-1} (x-c)^j \ f^{(j)} \ (c)/j! + R_k(x, c) \ .$$

As a simple but instructive example we consider the evaluation of $\exp(x)$ for $|x| \le 1$, using

$$\exp(x) = \sum_{j=0}^{k-1} x^j/j! + R_k(x) \ , \tag{4.7}$$

where $|R_k(x)| \le |x|^k \exp(|x|)/k! \le e/k!$

Using Stirling's approximation for $k!$, we see that $k \ge K(n) \sim n/\lg n$ is sufficient to ensure that $|R_k(x)| = O(2^{-n})$. Thus the time required is $O(nM(n)/\ln n)$.

In practice it is convenient to sum the series in the forward direction ($j = 0, 1, \ldots, k-1$). The terms $T_j = x/j!$ 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 $|T_k| < 2^{-n}$. Thus, it is not necessary to estimate k in advance, as it would be if the series were summed by Horner's rule in the backward direction ($j = k-1, k-2, \ldots, 0$).

We now consider the effect of rounding errors, under the assumption that floating-point operations satisfy

$$fl(x \text{ op } y) = (x \text{ op } y)(1 + \delta) \ ,$$

where $|\delta| \le \varepsilon$ and "op" = "+", "−", "×" or "/". Here $\varepsilon = \frac{1}{2}\beta^{1-t}$ is the "machine-precision" [63]. 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

$$\begin{aligned} |\widehat{S}_k - S_k| \quad &\le \quad ke\varepsilon + \sum_{j=1}^{k} 2j\varepsilon|T_j| + O(\varepsilon^2) \\ &\le \quad (k+2)e\varepsilon + O(\varepsilon^2) = O(n\varepsilon) \ . \end{aligned}$$

Thus, to get $|\widehat{S}_k - S_k| = O(2^{-n})$ 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 k has to be decided in advance. If we sum in the forward direction then k 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 ε fixed. We can profitably reduce the working precision when computing T_k from T_{k-1} if $|T_{k-1}| \ll 1$, 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 (4.7) is at least of order $|x|$. Thus, the method is slow for large $|x|$ (see §4.3 for faster methods in this case).

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 $2|x|/\ln\beta$ guard digits are required to compensate for what Lehmer called "catastrophic cancellation" [25]. 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.

In the following sections 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 [23]. Here *unrestricted* means that there is no *a priori* bound on $|x|$, the precision or the exponent range.

To conclude this section we give a less trivial example where power series expansions are useful. To compute the error function

$$\mathrm{erf}(x) = 2\pi^{-1/2} \int_0^x e^{-u^2} \, du \, ,$$

we may use the series

$$\mathrm{erf}(x) = 2\pi^{-1/2} \sum_{j=0}^{\infty} \frac{(-1)^j \, x^{2j+1}}{j!(2j+1)} \tag{4.8}$$

or

$$\mathrm{erf}(x) = 2\pi^{-1/2} \exp(-x^2) \sum_{j=0}^{\infty} \frac{2^j \, x^{2j+1}}{1 \cdot 3 \cdot 5 \cdots (2j+1)} \, . \tag{4.9}$$

The series (4.9) is preferable to (4.8) 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 $\mathrm{erf}c(x) = 1 - \mathrm{erf}(x)$: see §§4.5–4.6.

4.5 Asymptotic Expansions

Example: Ei, erf (cut-off point with power series).

4.6 Continued Fractions

Examples: Ei, erf, Bessel functions

4.7 Recurrence Relations

Linear and/or nonlinear recurrence relations (an example of nonlinear is considered in §4.8). Example: Bessel functions

4.8 Arithmetic-Geometric Mean

[Another nonlinear recurrence, important enough to treat separately.]

References: Brent [11, 14, 16], Borweins' book [9], Salamin, Hakmem, Bernstein (`http://cr.yp.to/papers.html#logagm`), etc.

Quadratic convergence but lacks self-correcting property of Newton's method.

Take care over sign for complex case ($\Re(\sqrt{z}) \geq 0$), or avoid it if possible.

log → exp → sin, cos → inverse sin, cos, tan

(Landen transformations (reference [14] and exercise), more efficient methods [11].)

Can improve constants in [11, 14] by using faster square root algorithm, see Chapter 3.

Ex: π (using Lagrange identity), $\log 2$, elliptic integrals and functions.

References: Borwein's book [9], Arndt and Haenel [3].

4.9 Binary Splitting

Cf [27] and the CLN implementation.

Used together with argument reduction (e.g. Brent's algorithm for exp).

The following is an extension of Brent's algorithm to sin and cos (Theorem 6.2 of [13] states a $O(M(n)\log^2 n)$ bound for sin, but the method is not given in detail; it seems the author had in mind computing $\exp z$ for complex z). We can avoid that by considering the real and imaginary parts separately:

```
Input: floating-point $|x| < 1/2$ in precision $p$
Output: an approximation of $\sin x$ and $\cos x$ to precision $p$
1. Write $x = \sum_{i=0}^{k} r_i \cdot 2^{2^{i+1}}$, where $r_i$ is an integer of at most $2^i$ bits
2. Let $x_j = \sum_{i=j}^{k} r_i \cdot 2^{2^{i+1}}$, and $y_i = r_i \cdot 2^{2^{i+1}}$, thus $x_j = y_i + x_{j+1}$
3. Compute $\sin y_i$ and $\cos y_i$ using binary splitting
4. Reconstruct $\sin x_j$ and $\cos x_j$ using
```

$$\begin{aligned}\sin x_j &= \sin y_i \cos x_{j+1} + \cos y_i \sin x_{j+1}\\ \cos x_j &= \cos y_i \cos x_{j+1} + \sin y_i \sin x_{j+1}\end{aligned}$$

4.10 Holonomic Functions

A function $f(x)$ is said to be *holonomic* iff it satisfies a linear differential equation with polynomial coefficients in x. Equivalently, the Taylor coefficients u_n of f satisfy a linear recurrence with polynomial coefficients in n. For example, the $\exp, \log, \sin, \cos$ functions are holonomic, but tan is not.

We describe here the "bit-burst" algorithm invented by the Chudnovsky brothers [22]:

Theorem 4.10.1 *If f is holonomic (or 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 in time $O(M(n)\log^3 n)$ to n bits.

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 $r_1 = 0.b_1$, $r_2 = 0.0b_2 b_3$, $r_3 = 0.000b_4 b_5 b_6 b_7$: 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 $x_i = r_1 + \cdots + r_i$. The idea of the algorithm is to transfer the Taylor series of f from x_i to x_{i+1}, which since f is holonomic reduces to converting the recurrence. 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 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:

$$f(x) = f_0^*(r_1) + \cdots + f_i^*(r_{i+1}) + \cdots + f_{k-1}^*(r_k) = \sum_{i=0}^{k-1} f_i^*(r_{i+1}) \ .$$

To conclude the proof, it suffices to show that each term $f_i^*(r_{i+1})$ can be evaluated to n bits in $O(M(n) \log^2 n)$.

Now r_{i+1} is a rational whose numerator has 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 holonomic. Assume f satisfies the following linear differential equation with polynomial coefficients:

$$c_m(t) f^{(m)}(t) + \cdots + c_1(t) f'(t) + c_0(t) f(t) = 0.$$

Substituting $x_i + t$ for x, 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 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 kth Taylor coefficient from $f_i(t)$ has size $O(k(2^i + \log k))$

[the $k \log k$ term comes from the polynomials in k in the recurrence]. Since k goes to $n/2^i + O(\log n)$ at most, this is $O(n \log n)$.

However we do not want to evaluate the kth Taylor coefficient u_k of $f_i(t)$, but the series

$$s_k = \sum_{j=1}^{k} u_j r_{i+1}^j.$$

Noticing that $u_j = (s_j - s_{j-1})/r_{i+1}^j$, and substituting that value in the recurrence for (u_j), say of order l, we obtain a recurrence of order $l+1$ for (s_k). Putting this latter recurrence in matrix form $S_k = M(k)S_{k-1}$, where S_k is the vector $(s_k, s_{k-1}, s_{k-l+1})$, we obtain $S_k = M(k)M(k-1)\cdots M(l)S_{l-1}$, where the matrix product $M(k)M(k-1)\cdots M(l)$ can be evaluated in $O(M(n)\log^2 n)$ using binary splitting. □

We illustrate the above theorem with the *arctan* function, which satisfies the differential equation:

$$f'(t)(1+t^2) = 0.$$

This equation evaluates at $x_i + t$ into $f_i'(t)(1+(x_i+t)^2) = 0$, which gives the recurrence

$$(1+x_i^2)ku_k + 2x_i(k-1)u_{k-1} + (k-2)u_{k-2} = 0.$$

This recurrence translates to

$$(1+x_i^2)kv_k + 2x_i r_{i+1}(k-1)v_{k-1} + r_{i+1}^2(k-2)v_{k-2} = 0$$

for $v_k = u_k r_{i+1}^k$, and to

$$(1+x_i^2)k(s_k - s_{k-1}) + 2x_i r_{i+1}(k-1)(s_{k-1} - s_{k-2}) + r_{i+1}^2(k-2)(s_{k-2} - s_{k-3}) = 0$$

for $s_k = \sum_{j=1}^{k} v_j$.

4.11 Contour Integration

Example (Bernoulli numbers):

$$\frac{z}{e^z - 1} + \frac{z}{2}\ .$$

4.12 Other Special Functions

$\Gamma(x)$, $\Psi(x)$, $\zeta(x)$, $\zeta(\frac{1}{2}+iy)$, etc. Bombieri conjecture re evaluation of $\zeta(s)$ on critical line. Borwein algorithm for $\zeta(s)$ with small $\Im(s)$.

4.13 Constants

Ex: exp, π, γ [18], etc. Cf `http://cr.yp.to/1987/bernstein.html` for π and e. Cf also [26].

4.14 Summary of Best Methods

Table giving for each function the best method known (for asymptotic complexity, perhaps with a comment on the corresponding region), together with the corresponding complexity.

4.15 Notes and Further References

For more about holonomic or D-finite functions, see for example [49].

Abramowitz & Stegun [1]. Book of Nico Temme [57].

Cf `http://remote.science.uva.nl/~thk/specfun/compalg.html` and `http://numbers.computation.free.fr/Constants/constants.html`

For history, see Bernstein [5, 6].

4.16 Exercises

Exercise 4.16.1 If A(z) $= \sum_{j\ge 0} a_j z^j$ is a formal power series over $\mathbb{R}$ with $a_0 = 1$, show that $\log(A(x))$ can be computed with error $O(z^n)$ 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 z is replaced by $1/2$?

Exercise 4.16.2 (Brent) Assuming we 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 $\sim T(n)$.

Appendix: Implementations and Pointers

A few words about implementations? Pointers to mailing-lists (perhaps only on the web page or CD, with errata)

Bibliography

[1] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions.* Dover, 1973.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] Jörg Arndt and Christoph Haenel. *π Unleashed.* Springer-Verlag, 2001.

[4] 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.

[5] Dan J. Bernstein. Computing logarithm intervals with the arithmetic-geometric-mean iteration. `http://cr.yp.to/arith.html#logagm`, 2003. 8 pages.

[6] Dan J. Bernstein. Removing redundancy in high-precision Newton iteration. `http://cr.yp.to/fastnewton.html`, 2004. 13 pages.

[7] R. Bernstein. Multiplication by integer constants. *Software, Practice and Experience*, 16(7):641–652, 1986.

[8] 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.

[9] J. M. Borwein and P. B. Borwein. *Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity.* Wiley, 1998.

[10] 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`.

[11] 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`.

[12] 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`.

[13] 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`.

[14] 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`.

[15] 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`.

[16] Richard P. Brent. Fast algorithms for high-precision computation of elementary functions. `http://wwwmaths.anu.edu.au/~brent/talks.html#RNC7`, 2006. Invited talk presented at the Real Numbers and Computation Conference (RNC7), Nancy, France, July 2006.

[17] 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`.

[18] 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`.

[19] Richard P. Brent, Colin Percival, and Paul Zimmermann. Error bounds on complex floating-point multiplication. *Mathematics of Computation*, to appear. `http://wwwmaths.anu.edu.au/~brent/pub/pub221.html`.

[20] 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`.

[21] Christoph Burnikel and Joachim Ziegler. Fast recursive division. Research Report MPI-I-98-1-022, MPI Saarbrücken, October 1998. urlhttp://data.mpi-sb.mpg.de/internet/reports.nsf/NumberView/1998-1-022.

[22] D. V. Chudnovsky and G. 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.

[23] C. W. Clenshaw and F. W. J. Olver. An unrestricted algorithm for the exponential function. *SIAM J. Numerical Analysis*, 17:310–331, 1980.

[24] Richard E. Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective.* Springer-Verlag, 2001.

[25] George E. Forsythe. Pitfalls in computation, or why a math book isn't enough. *Amer. Math. Monthly*, 77:931–956, 1970.

[26] X. Gourdon and P. Sebah. Numbers, constants and computation. `http://numbers.computation.free.fr/Constants/constants.html`.

[27] B. Haible and T. Papanikolaou. Fast multiprecision evaluation of series of rational numbers. Technical Report TI-7/97, Darmstadt University of Technology, 1997. `http://www.informatik.th-darmstadt.de/TI/Mitarbeiter/papanik/`.

[28] Tom R. Halfhill. The truth behind the Pentium bug. *Byte*, March 1995.

[29] Guillaume Hanrot and Paul Zimmermann. A long note on Mulders' short product. *Journal of Symbolic Computation*, 2003. To appear.

[30] T. Jebelean. Practical integer division with Karatsuba complexity. In W. W. Küchlin, editor, *Proc. ISSAC'97*, pages 339–341, Maui, Hawaii, 1997.

[31] Tudor Jebelean. An algorithm for exact division. *Journal of Symbolic Computation*, 15:169–180, 1993.

[32] Tudor Jebelean. A double-digit Lehmer-Euclid algorithm for finding the GCD of long integers. *Journal of Symbolic Computation*, 19:145–157, 1995.

[33] Alan H. Karp and Peter Markstein. High-precision division and square root. *ACM Transactions on Mathematical Software*, 23(4):561–589, 1997.

[34] D. Knuth. The analysis of algorithms. In *Actes du Congrès International des Mathématiciens de 1970*, volume 3, pages 269–274, Paris, 1971. Gauthiers-Villars.

[35] Donald E. Knuth. *The Art of Computer Programming*, volume 2 : Seminumerical Algorithms. Addison-Wesley, second edition, 1981.

[36] 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`.

[37] Werner Krandick and Tudor Jebelean. Bidirectional exact integer division. *Journal of Symbolic Computation*, 21(4–6):441–456, 1996.

[38] H. Kuki and W. J. Cody. A statistical study of the accuracy of floating-point number systems. *Communications of the ACM*, 16:223–230, 1973.

[39] 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`.

[40] Roman Maeder. Storage allocation for the Karatsuba integer multiplication algorithm. DISCO, 1993. preprint.

[41] Gérard Maze. Existence of a limiting distribution for the binary gcd algorithm. *Journal of Discrete Algorithms*, to appear.

[42] 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`.

[43] 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.

[44] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[45] T. Mulders. On short multiplications and divisions. *Applicable Algebra in Engineering, Communication and Computing*, 11(1):69–88, 2000.

[46] Victor Pan. *How to Multiply Matrices Faster*, volume 179 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[47] Colin Percival. Rapid multiplication modulo the sum and difference of highly composite numbers. *Mathematics of Computation*, 72(241):387–395, 2003.

[48] 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`.

[49] 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.

[50] Arnold Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.

[51] Arnold Schönhage. Variations on computing reciprocals of power series. *Information Processing Letters*, 74:41–46, 2000.

[52] Arnold Schönhage, A. F. W. Grotefeld, and E. Vetter. *Fast Algorithms, A Multitape Turing Machine Implementation.* BI-Wissenschaftsverlag, 1994.

[53] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[54] Jonathan P. Sorenson. Two fast GCD algorithms. *Journal of Algorithms*, 16:110–144, 1994.

[55] Damien Stehlé and Paul Zimmermann. A binary recursive gcd algorithm. In *Proceedings of the Algorithmic Number Theory Symposium (ANTS VI)*, 2004.

[56] A. Svoboda. An algorithm for division. *Information Processing Machines*, 9:25–34, 1963.

[57] Nico M. Temme. *Special Functions: An Introduction to the Classical Functions of Mathematical Physics.* Wiley, John and Sons, Inc., 1996. `http://www.addall.com/Browse/Detail/0471113131.html`.

[58] J. F. Traub. *Iterative Methods for the Solution of Equations.* Prentice-Hall, Englewood Cliffs, New Jersey, 1964.

[59] Brigitte Vallée. Dynamics of the binary Euclidean algorithm: Functional analysis and operators. *Algorithmica*, 22:660–685, 1998.

[60] 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`.

[61] Jean Vuillemin. private communication, January 2004.

[62] Kenneth Weber. The accelerated integer GCD algorithm. *ACM Transactions on Mathematical Software*, 21(1):111–122, 1995.

[63] James H. Wilkinson. *Rounding Errors in Algebraic Processes.* HMSO, London, 1963.

[64] C. K. Yap. *Fundamental Problems in Algorithmic Algebra.* Oxford University Press, 2000.

[65] Dan Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, 1994.