Arithmétique entière

Paul Zimmermann

Arithmétique entière

Paul Zimmermann*

Our main topic here 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 Representation and Notations

We consider here 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 $0 \leq a_i \leq \beta - 1$, and a_{n-1} is sometimes assumed to be non-zero. Since the base β is usually fixed in a given program, 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, which is known as the "sign-magnitude" representation. Zero is an important special case; to simplify the

*Ces notes pour les Journées Nationales de Calcul Formel 2007 sont largement inspirées du chapitre 1 du livre *Modern Computer Arithmetic* en cours de rédaction avec Richard Brent [1]. Plutôt que de les traduire en français, j'ai préféré me concentrer sur le fond. Que le lecteur francophone et M. Toubon veuillent bien m'en excuser.

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 [6].

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 = Σ_0^{n-1} a_i β^i , B = Σ_0^{n-1} b_i β^i
3 Output: C := Σ_0^{n-1} c_i β^i and 0 ≤ d ≤ 1 such that A + B = dβ^n + C
4 d ← 0
5 for i from 0 to n − 1 do
6     s ← a_i + b_i + d
7     c_i ← s mod β
8     d ← s div β
9 Return C, d.
```

Let 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 $\beta \leq \lceil M/2 \rceil$.

The subtraction code is very similar. Step 6 simply becomes $s \leftarrow a_i - b_i + d$, where $d \in \{0, -1\}$ is the *borrow* of the subtraction, and $-\beta \leq s < \beta$ (assuming 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 §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.

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 $a = \sum_{0 \le i < n} a_i \beta^i$ and $b = \sum_{0 \le j < n} b_j \beta^j$. Multiply the polynomials $A(x) = \sum_{0 \le i < n} a_i x^i$ and $B(x) = \sum_{0 \le j < n} b_j x^j$, obtaining a polynomial $C(x)$, then evaluate $C(x)$ at $x = \beta$ to obtain ab. Note that the coefficients of $C(x)$ may be larger than β, in fact they may be of order $n\beta^2$. These examples demonstrate the analogy between operations on polynomials and integers, and also show the limits of the analogy.

3.1 Naive Multiplication

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 $\quad C \leftarrow C + \beta^j (A \cdot b_j)$
7 Return C.

Theorem 3.1 *Algorithm* ***BasecaseMultiply*** *correctly computes the product AB, and uses $\Theta(mn)$ word operations.*

The multiplication by β^j at step 6 is trivial with the chosen dense representation: it simply requires shifting by j words towards the most significant words. The main operation in algorithm **BasecaseMultiply** is the computation of $A \cdot b_j$ 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 thus the multiplication of an array of m words by one word, with accumulation of the result in another array of $m + 1$ words.

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

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.

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

Theorem 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 $C = 3^{1-\log_2 n_0} \max(K(n_0), (n_0 - 1)^2)$. □

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.

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**.

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.

Most fast multiplication algorithms can be viewed as evaluation/interpolation algorithms, from a polynomial point of view. Karatsuba's algorithm regards the inputs as polynomials $A_0 + A_1 x$ and $B_0 + B_1 x$ evaluated 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]

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

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

The case $r = 2$ corresponds to Karatsuba's algorithm (§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.

We refer the reader interested in higher order Toom-Cook implementations to [8], 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 parameters the evaluation points; if one chooses consecutive integer points, the determinant of that matrix contains all primes up to $2r - 2$. This proves that the division by 3 can not be avoided for Toom-Cook 3-way with consecutive integer points.

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

[2] An exact division can be performed from the least significant bits, which is usually more efficient: see §4.5.

Algorithm **ToomCook3**.
Input: two integers $0 \leq A, B < \beta^n$.
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 $A = a_0 + a_1 x + a_2 x^2$, $B = b_0 + b_1 x + b_2 x^2$ 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)
$t_1 \leftarrow (3v_0 + 2v_{-1} + v_2)/6 - 2v_\infty$, $t_2 \leftarrow (v_1 + v_{-1})/2$
$c_0 \leftarrow v_0$, $c_1 \leftarrow v_1 - t_1$, $c_2 \leftarrow t_2 - v_0 - v_\infty$, $c_3 \leftarrow t_1 - t_2$, $c_4 \leftarrow v_\infty$

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, since they involve $O(r^2)$ scalar operations. The Fast Fourier Transform (FFT) is a way to perform evaluation and interpolation in an efficient way for some special values of r. This explains why multiplication algorithms of best asymptotic complexity are based on the Fast Fourier Transform.

There are different flavours of FFT multiplication, depending on the ring where the operations are performed. The asymptotically best algorithm, due to Schönhage-Strassen [4], with a complexity of $O(n \log n \log\log n)$, works in the ring $\mathbb{Z}/(2^n + 1)\mathbb{Z}$.

Another method commonly used is to work with floating-point complex numbers [2, Section 4.3.3.C]; one drawback is that, due to the inexact nature of floating-point computations, a careful error analysis is required to guarantee the correctness of the implementation, assuming an underlying arithmetic with rigorous error bounds.

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:

> Consider first $k = 2$. In the FFT range, the trivial strategy costs $2M(n)$, whereas the strategy which performs one $2n \times n$ product using a FFT of size $3n$, which is equivalent to a $\frac{3}{2}n \times \frac{3}{2}n$ product, costs $M(\frac{3}{2}n) \approx \frac{3}{2}M(n)$. In

general the trivial strategy costs $kM(n)$, whereas the "big-FFT" strategy costs $\frac{k+1}{2}M(n)$.

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" requires $K(\lceil m/2 \rceil, \lceil m/2 \rceil)$ for the product of the differences (or sums) of the low and high parts from the operands, due to a "wrap around" effect when subtracting the parts from the smaller operand; this will ultimately lead to a cost similar to that of a $m \times m$ product. The "odd-even strategy" avoids this wrap around:

> 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. Design an algorithm `UnbalancedKaratsuba` using that scheme. Show that its complexity satisfies $K(m,n) = 2K(\lceil m/2 \rceil, \lceil n/2 \rceil) + K(\lfloor m/2 \rfloor, \lfloor n/2 \rfloor)$.

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.

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 im-

plementation of multiplication itself, bearing in mind that the best possible speedup is two.

For naive multiplication, Algorithm **BasecaseMultiply** (§3.1) can be modified to obtain a theoretical speedup of two, since only half of the products $a_i b_j$ need to be computed.

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

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.

We distinguish several kinds of division: *full division* computes both quotient and remainder, while in some cases only the quotient (for example when dividing two floating-point mantissas) or remainder (when multiplying two residues modulo n) is needed. Finally we discuss exact division — when the remainder is known to be zero.

4.1 Naive Division

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

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** $A \geq \beta^m B$ **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 $\quad q_j^* \leftarrow \lfloor (a_{n+j}\beta + a_{n+j-1})/b_{n-1} \rfloor$
7 $\quad q_j \leftarrow \min(q_j^*, \beta - 1)$
8 $\quad A \leftarrow A - q_j \beta^j B$
9 $\quad$ **while** $A < 0$ **do**
10 $\quad\quad q_j \leftarrow q_j - 1$
11 $\quad\quad A \leftarrow A + \beta^j B$
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.)

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

Theorem 4.1 *Algorithm* ***BasecaseDivRem*** *correctly computes the quotient and remainder of the division of A by a normalized B, in $O(nm)$ word operations.*

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

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

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

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

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

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

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

We conclude that when the for-loop ends, $0 \leq A < B$ holds, and since $(\sum_j^{m-1} q_i\beta^i)B + A$ 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 branch prediction algorithms available on modern processors will fail, resulting in wasted cycles. A workaround is to compute a more accurate partial quotient, for example with a division of 3 words by 2 words at step 6, and therefore decrease the proportion of corrections almost to zero.

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 [5] 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, $q_j^* \leq \beta - 1$ is then always fulfilled, thus step 7 can be avoided, and q_j^* replaced by q_j.)

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

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

With the example of section §4.1, Svoboda's algorithm would give $k = 1160$, $B' = 1000691299080$,

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

We thus get $Q' = 766440$ and $R' = 705575568644$. The final division gives $R' = 817B + 778334723$, thus we 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 $Q = q_0 + kQ'$ (or when only the quotient is needed, by dividing $A' = kA$ by $B' = kB$).

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 (§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_1 here); in such a way, we only have to subtract the product of Q_1 by the low part of the divisor.

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

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 yields a two-digit quotient 11. Even $A \leq \beta^m B$ is

not recursively fulfilled; consider $A = 55170000$ with $B = 5517$: the first recursive call will divide 5517 by 55.

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

A 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 step 11 iterations, we have $0 \le A'' < B$. □

Remark 3: Theorem 4.2 gives $D(n) \sim 2M(n)$ for Karatsuba multiplication, and $D(n) \sim 2.63M(n)$ for Toom-Cook 3-way.
Remark 4: to decrease the probability that the estimated quotients Q_1 and Q_0 are too large, one may use one extra word of the truncated dividend and divisors in the recursive calls to **RecursiveDivRem**.

A graphical view of Algorithm **RecursiveDivRem** in the case $m = 2n$ is given on Fig. 1, which represents the multiplication $Q \cdot B$: one firstly computes the lower left corner in $D(n/2)$ (step 6), secondly the lower right corner in $M(n/2)$ (step 7), thirdly the upper left corner in $D(n/2)$ (step 9), and finally the upper right corner in $M(n/2)$ (step 10).

4.3.1 Unbalanced Division

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

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

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. The p-adic version of Newton's method, also called Hensel lifting, is used in §4.5 for the exact division.

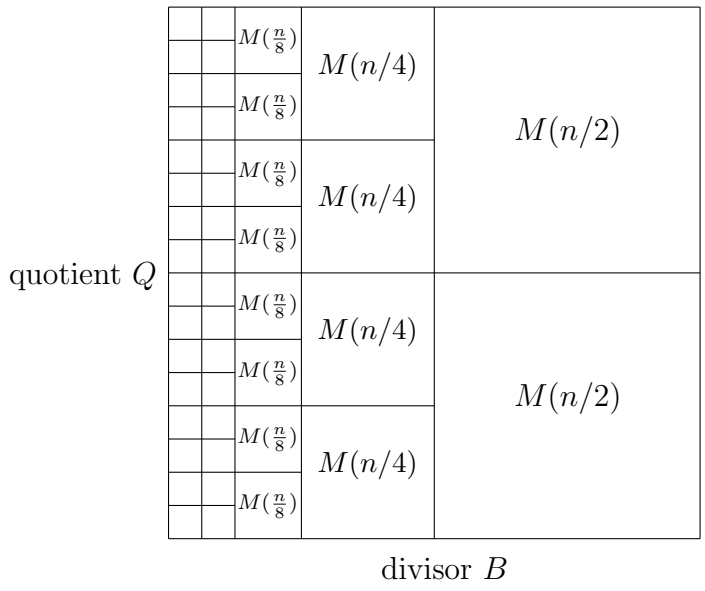

Figure 1: Divide and conquer division: a graphical view (most significant parts at the lower left corner).

```
1  Algorithm UnbalancedDivision.
2  Input: A = Σ_0^{n+m-1} a_i β^i , B = Σ_0^{n-1} b_j β^j .
3  Output: quotient Q and remainder R of A divided by B.
4  Assumptions: m > n , B normalized.
5  Q ← 0
6  while m > n do
7      (q, r) ← RecursiveDivRem(A div β^{m-n}, B)
8      Q ← Qβ^n + q
9      A ← rβ^{m-n} + A mod β^{m-n}
10     m ← m − n
11 (q, r) ← RecursiveDivRem(A, B)
12 Return Q := Qβ^m + q , R := r .
```

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

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* 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 (MSB) strategy, and the least $n/2$ bits using the second (LSB) one. Since an exact division of size n is replaced by two exact divisions of size $n/2$, this gives a speedup up to 2 for quadratic algorithms.

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 $(a - r)/b$ (§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. Conversely, the multiplication of two numbers modulo n reduces to the remainder of their product after division by n. 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 mod B_1 reduces to a single $n/2 \times n$ multiplication. Once A is reduced into A_1 of $3n/2$ words by Svoboda's algorithm in $2M(n/2)$, use `RecursiveDivRem` on A_1 and B, which costs $D(n/2) + M(n/2)$. The total cost is thus $3M(n/2) + D(n/2)$, instead of $2M(n/2) + 2D(n/2)$ for a full division with `RecursiveDivRem`. This gives $\frac{5}{3}M(n)$ for Karatsuba and $2.04M(n)$ for Toom-Cook 3-way.

4.7 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. 2). Given a dividend A of $2n$ words and a divisor B of n words, the classical or MSB (most

Figure 2: Classical/MSB division (left) vs Hensel/LSB division (right).

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.

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 §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.

5 Roots

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 $0 \leq r' < 100$. Since $(10s+t)^2 = 100s^2 + 20st + t^2$, a good approximation to the next digit t can be found by dividing $10r$ by $2s$.

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

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

Theorem 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 $S(n) \sim \frac{2}{3}M(n)$.*

5.2 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 $0 \leq t, n' < \beta$. 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.

5.2.1 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,

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

one may use modular algorithms when k is relatively prime to the base β (see above).
REMARK: in the above algorithm, one can limit the search to prime exponents k.

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.

6.1 Naive Gcd

We do not give Euclid's algorithm here: it can be found in many textbooks, e.g., Knuth [2], 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 [2, §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 $97,115,231$.

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

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.

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

6.1.1 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 6.1 *[7] 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 $ua \equiv vb \bmod m$.*

The following algorithm, `ReducedRatMod`, finds such a pair (u, v): it is a simple variation

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

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

When m is a prime power, the inversion $1/b \bmod m$ at line 4 can be performed efficiently using Hensel lifting, otherwise by an extended gcd algorithm (§6.2).

6.2 Extended Gcd

Algorithm `ExtendedGcd` (Table 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

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

Table 1: Algorithm `ExtendedGcd`.

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: 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 §4.5).

All known algorithms for subquadratic gcd rely on an extended gcd subroutine, so we refer to §6.3 for subquadratic extended gcd.

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

The main idea is 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 **HalfGcd** algorithm.

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

Let $H(n)$ be the complexity of HalfGcd for inputs of n bits: a_1 and b_1 have $n/2$ bits,

	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 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.

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

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

6.3.1 Subquadratic binary gcd

The binary gcd can also be made fast: see Table 3. The idea is to mimic the left-to-right version, by defining an appropriate right-to-left division (Algorithm `BinaryDivide`).

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,

Algorithm **BinaryHalfGcd**.
Input: $P, Q \in \mathbb{Z}$ with $0 = \nu(P) < \nu(Q)$, and $k \in \mathbb{N}$
Output: a 2×2 integer matrix R, $j \in \mathbb{N}$, and P', Q' such that
  ${}^t[P', Q'] = 2^{-j} R \cdot {}^t[P, Q]$ with $\nu(P') \leq k < \nu(Q')$
$m \leftarrow \nu(Q)$, $d \leftarrow \lfloor k/2 \rfloor$
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')
$P' \leftarrow P' + q2^{-m}Q'$, $d' \leftarrow k - (j_1 + m)$
$(P', Q') \leftarrow (2^{-m}P', 2^{-m}Q')$
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 $\nu(Q) < \nu(P + q2^{-j}Q)$
$Q' \leftarrow 2^{-j}Q$
$q \leftarrow -P/Q' \bmod 2^{j+1}$
if $q < 2^j$ **then** return q **else** return $q - 2^{j+1}$

Table 3: A subquadratic binary gcd algorithm.

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 in base β only.

7.1 Quadratic Algorithms

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

Algorithm **IntegerInput**.
Input: a string $S = s_{m-1} \dots s_1 s_0$ **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.

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 $S = s_{m-1} \dots s_1 s_0$.

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

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

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 this: If the output A

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

has n words, algorithm **FastIntegerInput** 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.

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

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

where $\text{Output}(A, B)$ 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 $\text{Output}(A_{\rm lo}, B)$ has k digits, after possibly adding leading zeros.

```
1 Algorithm FastIntegerOutput.
2 Input: A = sum_0^{n-1} a_i β^i of the number represented by S
3 Output: a string S of characters, representing A in base B
4 if A < B then char(A)
5 else
6     find k such that B^{2k-2} ≤ A < B^{2k}
7     (Q, R) ← DivRem(A, B^k)
8     FastIntegerOutput(Q)||FastIntegerOutput(R)
```

If the input A has n words, algorithm **FastIntegerOutput** has complexity $O(M(n) \log n)$, more precisely $\sim \frac{1}{2} D(n/2) \log_2 n$ 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 output may thus be 2 to 5 times slower than integer input.

References

[1] Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Version 0.1.1, 2006. `http://www.loria.fr/~zimmerma/mca/pub226.html`.

[2] 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`.

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

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

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

[6] 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`.

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

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