"Alexandru Ioan Cuza" University of Iaşi
Faculty of Computer Science

TECHNICAL REPORT

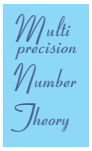
MpNT: A Multi-Precision Number Theory Package. Number-Theoretic Algorithms (I)

F.L. Ţiplea • S. Iftene
C. Hriţcu • I. Goriac
R.M. Gordân • E. Erbiceanu

TR 03-02, May 2003

ISSN 1224-9327

Universitatea "Alexandru Ioan Cuza" Iaşi
Facultatea de Informatică
Str. Berthelot 16, 6600-Iaşi, Romania
Tel. +40-32-201090, email: bibl@infoiasi.ro



MpNT: A Multi-Precision Number Theory Package

Number-Theoretic Algorithms (I)

F.L. Ţiplea • **S. Iftene**
C. Hriţcu • **I. Goriac**
R.M. Gordân • **E. Erbiceanu**

May 21, 2003

Faculty of Computer Science
"Al.I.Cuza" University of Iaşi
6600 Iaşi, Romania
E-mail: mptn@infoiasi.ro

Preface

MpNT is a multi-precision number theory package developed at the Faculty of Computer Science, "Al.I.Cuza" University of Iaşi, România. It has been started as a base for cryptographic applications, looking for both efficiency and portability without disregarding code structure and clarity. However, it can be used in any other domain which requires efficient large number computations.

The present paper is the first in a series of papers dedicated to the design of the MpNT library. It has two goals. First, it discusses some basic number-theoretic algorithms that have been implemented so far, as well as the structure of the library. The second goal is to have a companion to the courses *Algebraic Foundations of Computer Science* [64], *Coding Theory and Cryptography* [65], and *Security Protocols* [66], where most of the material of this paper has been taught and students were faced with the problem of designing efficient implementations of cryptographic primitives and protocols. From this point of view we have tried to prepare a self-contained paper. No specific background in mathematics or programming languages is assumed, but a certain amount of maturity in these fields is desirable.

Due to the detailed exposure, the paper can accompany well any course on algorithm design or computer mathematics.

The paper has been prepared as follows:

- Sections 1–6, by F.L. Ţiplea;
- Section 7, by S. Iftene;
- Section 8, by C. Hriţcu, I. Goriac, R.M. Gordân, and E. Erbiceanu.

The library has been implemented by C. Hriţcu, I. Goriac, R.M. Gordân, and E. Erbiceanu.

Iaşi, May 2003

Contents

Introduction

An integer in C is typically 32 bits, of which 31 can be used for positive integer arithmetic. Some compilers, such as GCC, offer a "long long" type, giving 64 bits capable of representing integers up to about $9 \cdot 10^{18}$. This is good for most purposes, but some applications require many more digits than this. For example, public-key encryption with the RSA algorithm typically requires 300 digit numbers or more. Computing the probabilities of certain real events often involves very large numbers; although the result might fit in a typical C type, the intermediate computations require very large numbers which will not fit into a C integer, not even a 64 bit one. Therefore, computations with large numerical data (having more than 10 or 20 digits, for example) need specific treatment. There is mathematical software, such as Maple or Mathematica, which provides the possibility to work with a non-limited precision. Such software can be used to prototype algorithms or to compute constants, but it is usually non-efficient since it is not dedicated to numerical computations. In order to overcome this shortcoming, several multi-precision libraries have been proposed, such as LIP [40], LiDIA [25], CLN [27], PARI [26], NTL [57], GMP [24] etc.

MpNT (**M**ulti-**p**recision **N**umber **T**heory) is a new multi-precision library developed at the Faculty of Computer Science, "Al.I.Cuza" University of Iaşi, România. It has been started as a base for cryptographic applications, looking for both efficiency and portability without disregarding code structure and clarity. However, it can be used in any other domain which requires efficient large number computations. The library is written in ISO C++ (with a small kernel in assembly language), so it should work on any available architecture. Special optimizations apply for the Intel IA-32 and compatible processors under Windows and Linux. Comparisons between our library and the existing ones show that it is quite efficient.

The present paper is the first in a series of papers dedicated to the design of the MpNT library. It has two goals. First, it discusses some basic number-theoretic algorithms that have been implemented so far, as well as the structure of the library. The second goal is to have a companion to the courses *Algebraic Foundations of Computer Science* [64], *Coding Theory and Cryptography* [65], and *Security Protocols* [66], where most of the material of this paper has been taught and students were faced with the problem of designing efficient implementations of cryptographic primitives and protocols. From this point of view we have tried to prepare a self-contained paper. No specific background in mathematics or programming languages is assumed, but a certain amount of maturity in these fields is desirable.

The paper is organized into 8 sections. The first section establishes the notation regarding base representation and complexity of algorithms, while the next 6 sections discuss algorithms for addition and subtraction, multiplication, division, greatest common divisor, modular reduction, and exponentiation. The last section is devoted to a short description of the MpNT library (for further details the reader is referred to the user guide and the home page of the library).

1 Preliminaries

In this section we establish the notation we use in our paper regarding base representation of integers and complexity of algorithms. For details, the reader is referred to classical textbooks on algorithmic number theory and analysis of algorithms, such as [1, 36, 23].

1.1 Base Representation of Integers

The set of *integers* is denoted by $\mathbf{Z}$. Integers $a \geq 0$ are referred to as *non-negative integers* or *natural numbers*, and integers $a > 0$ are referred to as *positive integers*. The set of natural numbers is denoted by $\mathbf{N}$.

For an integer a, $|a|$ stands for the *absolute value* of a, and $sign(a)$ stands for its sign defined by

$$sign(a) = \begin{cases} 1, & \text{if } a \geq 0 \\ -1, & \text{if } a < 0. \end{cases}$$

It is well-known that for any two integers a and $b \neq 0$ there exist unique integers q and r satisfying $a = bq + r$ and $0 \leq r < |b|$. The integer q is called the *quotient*, and the integer r the *remainder*, of a when divided by b. It is common to write $a \ div \ b$ for the quotient q, and $a \ mod \ b$ for the remainder r.

For a real number x, $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x, and $\lceil x \rceil$ denotes the least integer greater than or equal to x. The following properties hold true:

- $x - 1 < \lfloor x \rfloor \leq x$;
- $x \leq \lceil x \rceil < x + 1$;
- $\lfloor x \rfloor = -\lceil -x \rceil$;
- $\lfloor x \rfloor = \begin{cases} \lceil x \rceil = x, & \text{if } x \text{ is an integer} \\ \lceil x \rceil - 1, & \text{otherwise.} \end{cases}$

Let $\beta \geq 2$ be an integer called *base*. Each non-negative integer $a < \beta$ is called a *digit of the base* β or a *base* β *digit*.

The *free semigroup* generated by $\{0, \ldots, \beta - 1\}$ is denoted by β^+.

It is well-known that any non-negative integer a can be represented in a given base β as:

$$a = a_{k-1}\beta^{k-1} + \cdots + a_1\beta + a_0,$$

where $0 \leq a_i < \beta$ for all $0 \leq i < k$, and $a_{k-1} > 0$.

The k-ary vector

$$(a_{k-1}, \ldots, a_0)_\beta$$

is called the *representation of* a *in base* β, the numbers a_i are called the *digits of* a *in base* β, and k is the *length of the representation*. We shall also say that a is a *base* β k*-digit number*. When the base β is clear from the context (especially for base $\beta = 2$ or $\beta = 10$) we simplify the notation above to $a_{k-1} \cdots a_{0\,\beta}$ or even to $a_{k-1} \cdots a_0$, and we shall say that a *is a* k*-digit number*. The length of the representation of a in base β is denoted by $|a|_\beta$. It satisfies $|a|_\beta = \lfloor \log_\beta a \rfloor + 1$.

We identify positive integers by their representation in some base β. From this point of view we can write

$$(a_{k-1}, \ldots, a_0)_\beta = (a'_{l-1}, \ldots, a'_0)_{\beta'}$$

in order to specify that $(a_{k-1}, \ldots, a_0)_\beta$ and $(a'_{l-1}, \ldots, a'_0)_{\beta'}$ are distinct representations (in base β and β', respectively) of the same integer. For example,

$$32 = (100000)_2$$

means that 32 and $(100000)_2$ are representations in base 10 and base 2, respectively, of the same integer. From technical reasons, the notation $(a_{k-1}, \ldots, a_0)_\beta$ will be sometimes extended by allowing leading zeroes. For example, $(0, 0, 1, 1)_\beta$ also denotes the integer $0 \cdot \beta^3 + 0 \cdot \beta^2 + 1 \cdot \beta + 1 = \beta + 1$ whose base β representation is $(1, 1)_\beta$. The difference between these two notations will be clear from the context.

Recall now a few useful properties regarding base representation of integers (for details, the reader is referred to [67]). Let $a_0, \ldots, a_{k-1}$ be base β digits. Then,

1. $a_0 + a_1\beta + \cdots + a_i\beta^i < \beta^{i+1}$, for all i, $0 \leq i < k$;
2. $(a_{k-1}, \ldots, a_0)_\beta = (a_{k-1}, \ldots, a_{k-i})_\beta \cdot \beta^{k-i} + (a_{k-i-1}, \ldots, a_0)_\beta$, for all i, $1 \leq i < k$.

Sometimes, we shall be interested in decomposing the representation of an integer in blocks of digits. For example, it may be the case that the representation $(100000)_2$ be decomposed into 3 equally sized blocks, 10, 00, and 00. In this case we shall write $[10, 00, 00]_{2^2}$. In general, we write

$$[w_{l-1}, \ldots, w_0]_{\beta^m}$$

for the decomposition of $(a_{k-1}, \ldots, a_0)_\beta$ into blocks of m digits each, where $m \geq 1$ and w_i is a sequence of m β-digits, for all $0 \leq i < l$ (therefore, $k = ml$). The first block may be padded by zeroes to the left in order to have length m. For example, $[01, 01]_{2^2}$ is a decomposition of $(1, 0, 1)_2$ into blocks of length 2. When the length of blocks varies from block to block, we shall write

$$[w_{l-1}, \ldots, w_0]_{\beta^+},$$

but in this case no padding to the left of the first block is allowed. For example, $[1, 000, 00]_{2^+}$ is such a decomposition of $(1, 0, 0, 0, 0, 0)_2$.

If $a = (a_{k-1}, \ldots, a_0)_\beta$ is the base β representation of a and $k-1 \geq i \geq j \geq 0$, then $a[i : j]$ denotes the integer

$$a[i : j] = (a_i, \ldots, a_j)_\beta.$$

Using this notation, the relation at 2 can be re-written as

2'. $a = a[k-1 : k-i] \cdot \beta^{k-i} + a[k-i-1 : 0]$.

In practice, *big integers* or *multi-precision integers* are handled thanks to the representation in a suitable base β. The choice of the base β must satisfy some requirements such as:

- β should fit in a basic data type;
- β should be as large as possible to decrease the size of the representation of large integers and to decrease the cost of the basic algorithms running on them.

Typical choices for β are $\beta = 2^m$ or $\beta = 10^m$ (powers of 2 or 10).

1.2 Analysis of Algorithms

Two basic things must be taken into consideration when one describes an algorithm. The first one is to prove its correctness, i.e., that the algorithm gives the desired result when it halts. The second one is about its complexity. Since we are interested in practical implementations, we must give an estimate of the algorithm's running time (if possible, both in the worst and average case). The space requirement must also be considered. In many algorithms this is negligible, and then we shall not bother mentioning it, but in certain algorithms it becomes an important issue which has to be addressed.

The time complexity will be the most important issue we address. It will be always measured in *bit/digit operations*, i.e., logical or arithmetic operations on bits/digits. This is the most realistic model if using real computers (and not idealized ones). We do not list explicitly all the digit operations, but they will be clear from the context. For example, the addition of two digits taking into account the carry too is a digit operation. The *shifting* and *copying operations* are not considered digit operations [1]. In practice, these operations are fast in comparison with digit operations, so they can be safely ignored.

In many cases we are only interested in the dominant factor of the complexity of an algorithm, focusing on the shape of the running time curve, ignoring thus low-order terms or constant factors. This is called an *asymptotic analysis*, which is usually discussed in terms of the $\mathcal{O}$ notation.

Given a function f from the set of positive integers $\mathbf{N}$ into the set of positive real numbers $\mathbf{R}_+$, denote by $\mathcal{O}(f)$ the set

$$\mathcal{O}(f) = \{g : \mathbf{N} \to \mathbf{R}_+ | (\exists c > 0)(\exists n_0 \in \mathbf{N})(\forall n \geq n_0)(g(n) \leq cf(n))\}.$$

Notice that $\mathcal{O}(f)$ is a set of functions. Nonetheless, it is common practice to write $g(n) = \mathcal{O}(f(n))$ to mean that $g \in \mathcal{O}(f)$, or to say "$g(n)$ is $\mathcal{O}(f(n))$" than to say "g is in $\mathcal{O}(f)$". We also write $f(n) = \Theta(g(n))$ for "$f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(f(n))$".

We shall say that an algorithm is of time complexity $\mathcal{O}(f(n))$ if the running time $g(n)$ of the algorithm on an input of size n is $\mathcal{O}(f(n))$. When $f(n) = \log n$ ($f(n) = (\log n)^2$, $f(n) = P(\log n)$ for some polynomial P, $f(n) = n^\alpha$) we shall say that the algorithm is *linear* (*quadratic*, *polynomial*, *exponential*) *time*.

[1] The shifting operation shifts digits a few places to the left/right. The copying operation copies down a compact group of digits.

2 Addition and Subtraction

Addition and subtraction are two simple operations whose complexity is linear with respect to the length of the operands.

Let a and b be two integers. Their sum satisfies the following properties:

1. if a and b have the same sign, then $a + b = sign(a) \cdot (|a| + |b|)$;
2. if a and b have opposite signs and $|a| \geq |b|$, then $a + b = sign(a) \cdot (|a| - |b|)$;
3. if a and b have opposite signs and $|a| < |b|$, then $a + b = sign(b) \cdot (|b| - |a|)$.

Therefore, in order to compute the sum of two integers a and b we may focus on two basic operations: the sum of two positive integers a and b, and the difference of two different positive integers a and b.

The algorithm **Add** is a solution to the addition problem of two positive integers (we may assume that both of them have the same length of representation in base β).

Add(a, b)
`input:` $a = (a_{k-1}, \ldots, a_0)_\beta > 0$, $b = (b_{k-1}, \ldots, b_0)_\beta > 0$;
`output:` $c = a + b$;
`begin`
1. $carry := 0$;
2. `for` $i := 0$ `to` $k - 1$ `do`
   `begin`
3. $c_i := (a_i + b_i + carry)\ mod\ \beta$;
4. $carry := (a_i + b_i + carry)\ div\ \beta$;
   `end`;
5. `if` $carry > 0$ `then` $c_k := 1$;

`end`.

It is easy to see that the algorithm **Add** requires k digit operations, which shows that its time complexity is $\mathcal{O}(k)$.

In a similar way we can design an algorithm **Sub** for subtraction. Its complexity is $\mathcal{O}(k)$ as well.

Sub(a, b)
`input:` $a = (a_{k-1}, \ldots, a_0)_\beta$, $b = (b_{k-1}, \ldots, b_0)_\beta$, $a > b > 0$;
`output:` $c = a - b$;
`begin`
1. $borrow := 0$;
2. `for` $i := 0$ `to` $k - 1$ `do`
   `begin`
3. $c_i := (a_i - b_i + borrow)\ mod\ \beta$;
4. $borrow := (a_i - b_i + borrow)\ div\ \beta$;
   `end`;

`end`.

3 Multiplication

Several techniques can be used in order to compute the product ab of two integers. We shall present in this section the *school multiplication method*, the *Karatsuba method*, and the *FFT method*.

Since $a \cdot b = (sign(a) \cdot sign(b)) \cdot (|a| \cdot |b|)$ and $a \cdot 0 = 0 \cdot a = 0$, we may focus only on multiplication of non-negative integers.

3.1 School Multiplication

The technique we present here is an easy variation of the school method. It is based on computing partial sums after each row multiplication. In this way, the intermediate numbers obtained by addition do not exceed $\beta^2 - 1$ and, therefore, a basic procedure for multiplication of two base β digits can be used.

Formally, we can write

$$ab = \sum_{i=0}^{k} (a_i b)\beta^i,$$

where $a = \sum_{i=0}^{k} a_i \beta^i$. The algorithm **SchoolMul** given below exploits this property.

SchoolMul(a,b)
`input:` $a = (a_{k-1}, \ldots, a_0)_\beta \geq 0$, $b = (b_{l-1}, \ldots, b_0)_\beta \geq 0$;
`output:` $c = ab$;
`begin`

```
1.    if (a = 0 ∨ b = 0) then c := 0
         else begin
2.                for i := 0 to k − 1 do c_i := 0;
3.                for j := 0 to l − 1 do
                     begin
4.                      carry := 0;
5.                      for h := 0 to k − 1 do
                           begin
6.                            t := a_h b_j + c_{h+j} + carry;
7.                            c_{h+j} := t mod β;
8.                            carry := t div β
                           end;
9.                      c_{j+k} := carry;
                     end;
              end;
end.
```

It is easy to see that $c < \beta^{k+l}$ because $a < \beta^k$ and $b < \beta^l$. Therefore, we have $|c|_\beta \leq k + l$.

The number t in line 6 of the algorithm **SchoolMul** satisfies

$$t = a_h b_j + c_{h+j} + carry \leq (\beta - 1)^2 + (\beta - 1) + (\beta - 1) = \beta^2 - 1 < \beta^2,$$

because $a_h, b_j, c_{h+j}, carry < \beta$. Thus, the algorithm **SchoolMul** has the time complexity $\mathcal{O}(kl)$.

3.2 Karatsuba Multiplication

The school multiplication is far from being the best solution to the multiplication problem. In [33], Karatsuba and Ofman proposed an iterative solution to this problem. Let us assume that we want to multiply a and b, both of the same length k in base β. Moreover, assume that k is an even number, $k = 2l$. We can write:

$$a = u_1\beta^l + u_0 \quad \text{and} \quad b = v_1\beta^l + v_0$$

(u_1, u_0, v_1, and v_0 are l-digit integers obtained as in Section 1.1).

The number $c = ab$ can be decomposed as

$$c = u_1v_1\beta^{2l} + [(u_1 - u_0)(v_0 - v_1) + u_1v_1 + u_0v_0]\beta^l + u_0v_0$$

or as

$$c = u_1v_1\beta^{2l} + [(u_0 + u_1)(v_0 + v_1) - u_1v_1 - u_0v_0]\beta^l + u_0v_0.$$

No matter which of the two decompositions is chosen, the relation giving c consists only of three multiplications of base β numbers of length l and a few auxiliary operations (additions and shifts) whose complexity is linear to l. Therefore, the $M(2l)$ complexity of multiplying two base β sequences of length $k = 2l$ satisfies

$$M(2l) \leq \begin{cases} \alpha, & \text{if } l = 1 \\ 3M(l) + \alpha l, & \text{if } l > 1, \end{cases}$$

for some constant α.

By induction on i we obtain $M(2^i) \leq \alpha(3^i - 2^i)$. Indeed,

$$\begin{aligned} M(2^{i+1}) &= M(2 \cdot 2^i) \\ &\leq 3M(2^i) + \alpha 2^i \\ &\leq 3\alpha(3^i - 2^i) + \alpha 2^i \\ &= \alpha(3^{i+1} - 2^{i+1}). \end{aligned}$$

Let k and l be positive integers such that $2^{l-1} \leq k < 2^l$ (i.e., $l = \lfloor \log_2 k \rfloor + 1$). We have

$$M(k) \leq M(2^l) \leq \alpha(3^l - 2^l) < \alpha 3^l \leq \alpha 3^{1+\log_2 k} = 3\alpha k^{\log_2 3},$$

and thus $M(k) = \mathcal{O}(k^{\log_2 3})$.

We have obtained:

Theorem 3.2.1 There is an $\mathcal{O}(k^{\log_2 3})$ time complexity algorithm for multiplying two base β integers of length at most k.

The above idea leads directly to the **Karatsuba1** recursive algorithm of computing ab, when the length k of a and b is a power of 2. If k is small enough ($k \leq k_0$, for some constant k_0), then apply school multiplication; otherwise, apply a *Karatsuba step*. The constant k_0 depends on the machine and implementation, and it is assumed that school multiplication is faster than Karatsuba multiplication for integers of length less than k_0.

Karatsuba1(a, b, k)
`input:` $a, b \geq 0$ integers of length k, where k is a power of 2;
`output:` $c = ab$;
`begin`

1. `if` $k \leq k_0$ `then` **SchoolMul**(a,b)
   `else begin`
2. $l := k/2$;
3. `decompose` $a = u_1\beta^l + u_0$ and $b = v_1\beta^l + v_0$;
4. $A :=$**Karatsuba1**(u_1, v_1, l);
5. $B :=$**Karatsuba1**(u_0, v_0, l);
6. $C :=$**Karatsuba1**$(u_0 + u_1, v_0 + v_1, l)$;
7. $D := C - A - B$;
8. $c := A\beta^{2l} + D\beta^l + B$

   `end`

`end`.

The variables l and D are not really needed; they are used here only to help the comments below.

At each recursive call of **Karatsuba1**(a, b, k), where a and b have the length k and $k > k_0$, the following are true:

- step 3 is very fast: u_1, u_0, v_1, and v_0 are directly obtained from the base β representation of a and b by digit extraction (copy);
- step 8 is also very fast: multiplication by β means a left shift by a digit;
- 2 subtractions of integers of length k are needed to compute D at step 7. We notice that $D \geq 0$;
- one addition is nedeed in order to compute $c = ab$ at step 8. Indeed, if we consider the result split up into four parts of equal length (i.e., $k/2$), then it can be obtained from A, B, and D just by adding D as it is shown in Figure 1.

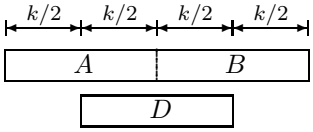


Figure 1: Getting ab in **Karatsuba1**(a, b, k)

As we have already seen, 2 subtractions and one addition of base β integers of length k are needed in order to complete a recursive call. This means 4 subtractions and 2 additions of base β integers of length $k/2$. Michel Quercia remarked in [51] (see also [72]) that a decomposition of A, B, and C before the step 7 in the algorithm above can result in a reduction of the number of additions. Indeed, let

- $A = A_1\beta^{k/2} + A_0$,
- $B = B_1\beta^{k/2} + B_0$, and

- $C = C_1\beta^{k/2} + C_0$.

Then, ab can be obtained as described in Figure 2, where the difference $A_0 - B_1$ occurs twice. More precisely, the decomposition of ab can be rewritten as follows:

$$ab = A_1\beta^{3k/2} + (C_1 - A_1 + (A_0 - B_1))\beta^k + (C_0 - B_0 - (A_0 - B_1))\beta^{k/2} + B_0.$$

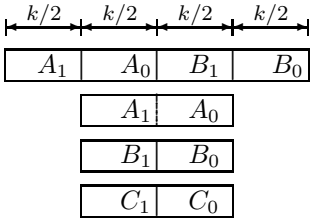


Figure 2: A new way of getting ab

In this way, only 4 subtractions and one addition of base β numbers of length $k/2$ are needed. The algorithm **Karatsuba2** exploits this property.

Karatsuba2(a, b, k)
`input:` $a, b \geq 0$ integers of length k, where k is a power of 2;
`output:` $c = ab$;
`begin`
1. `if` $k \leq k_0$ `then` **SchoolMul**(a, b)
   `else begin`
2. $l := k/2$;
3. `decompose` $a = u_1\beta^l + u_0$ and $b = v_1\beta^l + v_0$;
4. $A :=$**Karatsuba2**(u_1, v_1, l);
5. $B :=$**Karatsuba2**(u_0, v_0, l);
6. $C :=$**Karatsuba2**$(u_0 + u_1, v_0 + v_1, l)$;
7. `decompose` $A = A_1\beta^l + A_0$;
8. `decompose` $B = B_1\beta^l + B_0$;
9. `decompose` $C = C_1\beta^l + C_0$;
10. $D := A_0 - B_1$;
11. $A_0 := C_1 - A_1 + D$;
12. $B_1 := C_0 - B_0 - D$;
13. $c := A_1\beta^{3l} + A_0\beta^{2l} + B_1\beta^l + B_0$;
    `end`
`end.`

None of these two algorithms take into consideration the case of arbitrary length operands. In order to overcome this we have to consider two more cases:

- the operands a and b have an odd length k. Let $k = 2l + 1$. Then, we can write $a = u_1\beta + u_0$ and $b = v_1\beta + v_0$, which leads to

  $$ab = u_1v_1\beta^2 + (u_0v_1 + u_1v_0)\beta + u_0v_0.$$

  This relation shows that we can use a Karatsuba step to compute u_1v_1; the other products can be computed by school multiplication (whose complexity is linear to k);

- the operands a and b have different lengths. Let k be the length of a, and l be the length of b. Assume that $k > l$. Let $k = rl + k'$, where $k' < l$. We can write

$$a = u_r\beta^{(r-1)l+k'} + \cdots + u_1\beta^{k'} + u_0,$$

where $u_1, \ldots, u_r$ are of length l, and u_0 is of length k'. Then,

$$ab = u_r b\beta^{(r-1)l+k'} + \cdots + u_1 b\beta^{k'} + u_0 b.$$

Moreover, $u_i b$ can be computed by a Karatsuba step.

The algorithm **KaratsubaMul** takes into consideration these remarks and provides a complete solution to the multi-precision integer multiplication.

KaratsubaMul(a, b, k, l)
`input:` $a \geq 0$ of length k, and $b \geq 0$ of length l;
`output:` $c = ab$;
`begin`

1. `case` k, l `of`
2. $min\{k, l\} \leq k_0$: **SchoolMul**(a, b);
3. $k = l$ even:
   `begin`
4. $p := k/2$;
5. `decompose` $a = u_1\beta^p + u_0$ and $b = v_1\beta^p + v_0$;
6. $A :=$**KaratsubaMul**(u_1, v_1, p, p);
7. $B :=$**KaratsubaMul**(u_0, v_0, p, p);
8. $C :=$**KaratsubaMul**$(u_0 + u_1, v_0 + v_1, p, p)$;
9. $D := C - A - B$;
10. $c := A\beta^{2p} + D\beta^p + B$
    `end`;
11. $k = l$ odd:
    `begin`
12. `let` $k = 2k' + 1$;
13. `decompose` $a = u_1\beta + u_0$ and $b = v_1\beta + v_0$;
14. $A :=$**KaratsubaMul**(u_1, v_1, k', k');
15. $B :=$**SchoolMul**(u_0, v_1);
16. $C :=$**SchoolMul**(u_1, v_0);
17. $D :=$**SchoolMul**(u_0, v_0);
18. $c = A\beta^2 + (B + C)\beta + D$
    `end`
    `else`
    `begin`
    (let $k > l$);
19. `decompose` $k = rl + k'$, where $k' < l$;
20. `decompose` $a = u_r\beta^{(r-1)l+k'} + \cdots + u_1\beta^{k'} + u_0$;
21. `for` $i := 1$ `to` r `do` $A_i :=$**KaratsubaMul**(u_i, b, l, l);
22. $A_0 :=$**KaratsubaMul**(u_0, b, k', l);
23. $c := A_r\beta^{(r-1)l+k'} + \cdots + A_1\beta^{k'} + A_0$;
    `end`
    `end case`
    `end`.

3.3 FFT Multiplication

The main idea of the Karatsuba algorithm can be naturally generalized. Instead of dividing the sequences a and b into two subsequences of the same size, divide them into r equally sized subsequences,

$$a = u_{r-1}\beta^{(r-1)l} + \cdots + u_1\beta^l + u_0$$

and

$$b = v_{r-1}\beta^{(r-1)l} + \cdots + v_1\beta^l + v_0$$

(assuming that their length is $k = rl$). Let U and V be the polynomials

$$U(x) = u_{r-1}x^{r-1} + \cdots + u_1x + u_0$$

and

$$V(x) = v_{r-1}x^{r-1} + \cdots + v_1x + v_0$$

Thus, $a = U(\beta^l)$ and $v = V(\beta^l)$. Therefore, in order to determine ab it should be sufficient to find the coefficients of the polynomial $W(x) = U(x)V(x)$,

$$W(x) = w_{2r-2}x^{2r-2} + \cdots + w_1x + w_0.$$

The coefficients of W can be efficiently computed by the Fast Fourier Transform. This idea has been first exploited by Schönhage and Strassen [56]. Their algorithm is the fastest multiplication algorithm for arbitrary length integers.

The Discrete Fourier Transform The *discrete Fourier transform*, abbreviated DFT, can be defined over an arbitrary commutative ring $(R, +, \cdot, 0, 1)$ [2], usually abbreviated by R. The one-element ring (that is, the case $1 = 0$) is excluded from our considerations. For details concerning rings the reader is referred to [13].

Definition 3.3.1 Let R be a commutative ring. An element $\omega \in R$ is called a *primitive* or *principal* n^{th} *root of unity*, where $n \geq 2$, if the following properties are true:

(1) $\omega \neq 1$;

(2) $\omega^n = 1$;

(3) $\sum_{j=0}^{n-1}(\omega^i)^j = 0$, for all $1 \leq i < n$.

The elements $\omega^0, \omega^1, \ldots, \omega^{n-1}$ are called the n^{th} *roots of unity.*

Remark 3.3.1 Assume that R is a field and n has a multiplicative inverse n^{-1} in R [3]. Then, the third property in Definition 3.3.1 is equivalent to

(3') $\omega^i \neq 1$, for all $1 \leq i < n$.

[2] Recall that $(R, +, \cdot, 0, 1)$ is a commutative ring if $(R, +, 0)$ is an abelian group, $(R-\{0\}, \cdot, 1)$ is a commutative monoid, and the usual distributive laws hold.

[3] Integers appear in any ring, even in finite ones. Take n to be $1 + \cdots + 1$ (n times), where 1 is the multiplicative unit.

Indeed, if we assume that (3) holds true but there is an i such that $\omega^i = 1$, then the relation $\sum_{j=0}^{n-1}(\omega^i)^j = 0$ implies $n \cdot 1 = 0$, which leads to $1 = n^{-1} \cdot 0 = 0$; a contradiction.

Conversely, assume that (3') holds true. Then, for any i, $1 \leq i < n$, we have

$$0 = (\omega^n)^i - 1 = (\omega^i)^n - 1 = (\omega^i - 1)(\sum_{j=0}^{n-1}(\omega^i)^j),$$

which leads to $\sum_{j=0}^{n-1}(\omega^i)^j = 0$ because $\omega^i \neq 1$ and R is a field.

Example 3.3.1 In the field $\mathbf{C}$ of complex numbers, the element $\omega = e^{(2\pi i)/n}$ is a primitive n^{th} root of unity.

In the field $\mathbf{Z}_m^*$, where m is a prime number, any primitive element (generator of this field) is a primitive $(m-1)^{st}$ root of unity.

The following properties are obvious but they will be intensively used.

Proposition 3.3.1 Let R be a commutative ring and $\omega \in R$ be a primitive n^{th} root of unity. Then, the following properties are true:

(1) ω has a multiplicative inverse, which is ω^{n-1};

(2) $\omega^{-i} = \omega^{n-i}$, for all $0 \leq i < n$;

(3) $\omega^i\omega^j = \omega^{(i+j) \ mod \ n}$;

(4) If n is even, then $(\omega^i)^2 = (\omega^{n/2+i})^2$, for all $0 \leq i < n/2$;

(5) If $n > 2$, is even, and has a multiplicative inverse n^{-1} in R, then ω^2 is a primitive $(n/2)^{th}$ root of unity.

Proof (1) The relation $\omega^n = 1$ leads to $\omega\omega^{n-1} = 1$, which shows that ω^{n-1} is the multiplicative inverse of ω.

(2) follows from (1), and (3) uses the property $\omega^n = 1$.

(4) Let $0 \leq i < n/2$. Then, $\omega^0 = \omega^n$ leads to $\omega^{2i} = \omega^{n+2i}$ which is what we need.

(5) Clearly, $(\omega^2)^{n/2} = \omega^n = 1$. Let us prove that $\omega^2 \neq 1$. If we assume, by contradiction, that $\omega^2 = 1$, then the relation $\sum_{j=0}^{n-1}(\omega^2)^j = 0$ implies $n \cdot 1 = 0$, which leads to a contradiction as in Remark 3.3.1.

Let $1 \leq i < n/2$. We have to prove that $\sum_{j=0}^{n/2-1}((\omega^2)^i)^j = 0$. Since $2i < n$ we can write

$$\begin{aligned}
\textstyle\sum_{j=0}^{n-1}(\omega^{2i})^j &= \textstyle\sum_{j=0}^{n/2-1}(\omega^{2i})^j + \sum_{j=n/2}^{n-1}(\omega^{2i})^j \\
&= \textstyle\sum_{j=0}^{n/2-1}(\omega^{2i})^j + \sum_{j=0}^{n/2-1}(\omega^{2i})^{n/2+j} \\
&= \textstyle\sum_{j=0}^{n/2-1}(\omega^{2i})^j + \sum_{j=0}^{n/2-1}(\omega^{2i})^j \\
&= \textstyle 2\sum_{j=0}^{n/2-1}(\omega^{2i})^j \\
&= 0
\end{aligned}$$

(the property (4) and the fact that ω is a primitive n^{th} root of unity have been used). The relation $2\sum_{j=0}^{n/2-1}(\omega^{2i})^j = 0$ leads to $n\sum_{j=0}^{n/2-1}(\omega^{2i})^j = 0$, and to $\sum_{j=0}^{n/2-1}(\omega^{2i})^j = n^{-1} \cdot 0 = 0$.

Therefore, ω^2 is a primitive $(n/2)^{th}$ root of unity. $\diamond$

Remark 3.3.2 When R is a field, more useful properties about roots of unity can be proved. For example,

- $\omega^{n/2} = -1$, provided that n is even.

Indeed,

$$0 = \omega^n - 1 = (\omega^{n/2})^2 - 1 = (\omega^{n/2} - 1)(\omega^{n/2} + 1),$$

which shows that $\omega^{n/2} + 1 = 0$ because $\omega^{n/2} \neq 1$.

This property leads to another useful property, namely

- $\omega^{n/2+i} = -\omega^i$, for all $0 \leq i < n/2$, provided that n is even.

Let R be a commutative ring and $\omega \in R$ be a primitive n^{th} root of unity. Denote by $A_{\omega,n}$, or simply by A when ω and n are clear from the context, the matrix

$$A = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & (\omega^1)^1 & (\omega^1)^2 & \dots & (\omega^1)^{n-1} \\ & & & \dots & \\ 1 & (\omega^{n-1})^1 & (\omega^{n-1})^2 & \dots & (\omega^{n-1})^{n-1} \end{pmatrix}$$

That is, the line i of A contains all the powers of ω^i, for all $0 \leq i < n$.

Proposition 3.3.2 Let R be a commutative ring and $\omega \in R$ be a primitive n^{th} root of unity. If the integer n has a multiplicative inverse n^{-1} in R, then the matrix A is non-singular and its inverse A^{-1} is given by $A^{-1}(i,j) = n^{-1}\omega^{-ij}$, for all $0 \leq i, j < n$.

Proof It is enough to show that $A \cdot A^{-1} = A^{-1} \cdot A = I_n$, where I_n is the $n \times n$ identity matrix.

Let $B = A \cdot A^{-1}$. Then,

$$B(i,j) = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{k(i-j)},$$

for all i and j. If $i = j$ then $B(i,j) = 1$, and if $i > j$ then $B(i,j) = 0$ (by Definition 3.3.1(3)). In the case $i < j$ we use the property in Proposition 3.3.1(2), which leads to $B(i,j) = 0$. Therefore, $B = I_n$. The case $A^{-1} \cdot A = I_n$ can be discussed in a similar way to this. $\diamond$

For an n-dimensional row vector $\mathbf{a} = (a_0, \dots, a_{n-1})$ with elements from R, $\mathbf{a}^t$ stands for the transpose of $\mathbf{a}$.

Definition 3.3.2 Let R be a commutative ring and $\omega \in R$ be a primitive n^{th} root of unity.

(1) The *discrete Fourier transform* of a vector $\mathbf{a} = (a_0, \dots, a_{n-1})$ with elements from R is the vector $F(\mathbf{a}) = A\mathbf{a}^t$.

(2) The *inverse discrete Fourier transform* of a vector $\mathbf{a} = (a_0, \dots, a_{n-1})$ with elements from R is the vector $F^{-1}(\mathbf{a}) = A^{-1}\mathbf{a}^t$, provided that n has a multiplicative inverse in R.

Relationship Between DFT and Polynomial Evaluation There is a close relationship between DFT and polynomial evaluation. Let

$$a(x) = \sum_{j=0}^{n-1} a_j x^j$$

be an $(n-1)^{st}$ degree polynomial over R. This polynomial can be viewed as a vector

$$\mathbf{a} = (a_0, \ldots, a_{n-1}).$$

Evaluating the polynomial $a(x)$ at the points $\omega^0, \omega^1, \ldots, \omega^{n-1}$ is equivalent to computing $F(\mathbf{a})$, i.e.,

$$F(\mathbf{a}) = (a(\omega^0), a(\omega^1), \ldots, a(\omega^{n-1})).$$

Likewise, computing the inverse discrete Fourier transform is equivalent to finding (interpolating) a polynomial given its values at the n^{th} roots of unity.

Definition 3.3.3 Let $\mathbf{a} = (a_0, \ldots, a_{n-1})$ and $\mathbf{b} = (b_0, \ldots, b_{n-1})$ be two n-ary vectors over R. The *convolution* of $\mathbf{a}$ and $\mathbf{b}$, denoted by $\mathbf{a} \odot \mathbf{b}$, is the $2n$-ary vector $\mathbf{c} = (c_0, \ldots, c_{2n-1})$ given by

$$c_i = \sum_{j=0}^{i} a_j b_{i-j},$$

for all $0 \leq i < 2n$, where $a_i = b_i = 0$ for $i \geq n$.

We remark that $c_{2n-1} = 0$ in the definition above. This term is included only for symmetry.

Now, let $a(x)$ and $b(x)$ be two $(n-1)^{st}$ degree polynomials over R. It is easy to see that the coefficients of the polynomial $a(x)b(x)$ are exactly the components of the convolution $\mathbf{c} = \mathbf{a} \odot \mathbf{b}$, if we neglect c_{2n-1} which is 0.

Theorem 3.3.1 (Convolution Theorem)
Let R be a commutative ring, $\omega \in R$ be a primitive $(2n)^{th}$ root of unity, where $2n$ has a multiplicative inverse in R, and let $\mathbf{a} = (a_0, \ldots, a_{n-1})$ and $\mathbf{b} = (b_0, \ldots, b_{n-1})$ be two n-ary vectors over R. Then,

$$\mathbf{a} \odot \mathbf{b} = F^{-1}(F(\mathbf{a}') \cdot F(\mathbf{b}')),$$

where $\mathbf{a}' = (a_0, \ldots, a_{n-1}, 0, \ldots, 0)$, $\mathbf{b}' = (b_0, \ldots, b_{n-1}, 0, \ldots, 0)$, and "$\cdot$" is the componentwise product of vectors.

Proof Just compute $F(\mathbf{a}') \cdot F(\mathbf{b}')$ and $F(\mathbf{a} \odot \mathbf{b})$ and prove they are identical using the definition of the convolution product. $\diamond$

In the view of the Convolution Theorem, the efficiency of computing the convolution product depends directly on the efficiency of computing the (inverse) discrete Fourier transform.

The Fast Fourier Transform Algorithm Computing the (inverse) discrete Fourier transform of a vector $\mathbf{a} \in R^n$ requires time complexity $\mathcal{O}(n^2)$ (if we assume that arithmetic operations on elements of R require one step each). When n is a power of 2 there is an $\mathcal{O}(n \log n)$ time complexity algorithm due to Cooley and Tukey [16]. This algorithm is based on the following simple remark. Let $a'(x)$ and $a''(x)$ be the polynomials given by

$$a'(x) = a_0 + a_2 x + \cdots + a_{n-2} x^{n/2-1}$$

and

$$a''(x) = a_1 + a_3 x + \cdots + a_{n-1} x^{n/2-1},$$

where

$$a(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}.$$

Then,

$$(*) \quad a(x) = a'(x^2) + x a''(x^2).$$

Therefore,

- to evaluate $a(x)$ at the points $\omega^0, \omega^1, \ldots, \omega^{n-1}$ reduces to evaluate $a'(x)$ and $a''(x)$ at $(\omega^0)^2, (\omega^1)^2, \ldots, (\omega^{n-1})^2$, and then to re-combine the results via $(*)$. By Proposition 3.3.1(4), $a'(x)$ and $a''(x)$ will be evaluated at $n/2$ points. That is,
$$a(\omega^i) = a'(\omega^{2i}) + \omega^i a''(\omega^{2i})$$
and
$$a(\omega^{n/2+i}) = a'(\omega^{2i}) + \omega^{n/2+i} a''(\omega^{2i}),$$
for all $0 \leq i < n/2$;
- by Proposition 3.3.1(5), $a'(x)$ and $a''(x)$ will be evaluated at the $(n/2)^{th}$ roots of unity. Therefore, we can recursively continue with $a'(x)$ and $a''(x)$.

The algorithm **FFT1** given below is based on these remarks. Lines 5 and 6 in **FFT1** are the recursion, and lines 8–10 do the re-combination via $(*)$: line 8 does the first half, from 0 to $n/2 - 1$, while line 9 does the second half, from $n/2$ to $n - 1$.

Line 9 can be replaced by

$$y_i := y'_i - \bar{\omega} y''_i,$$

whenever the property $\omega^{n/2} = -1$ holds true in the ring R (e.g., when R is a field).

The time complexity $T(n)$ of this algorithm satisfies

$$T(n) = 2T(n/2) + \Theta(n)$$

(the loop 8–10 has the complexity $\Theta(n)$). So, $T(n) = \Theta(n \log n)$.

In order to compute $F^{-1}(\mathbf{a})$ we replace ω by ω^{-1} and multiply the result by n^{-1} in the algorithm **FFT1**$(\mathbf{a}, n)$. Therefore, the inverse DFT has the same time complexity.

```
   FFT1(a, n)
   input:  a = (a_0, ..., a_{n-1}) ∈ R^n, where n is a power of 2;
   output: F(a);
   begin
1.    if n = 1 then F(a) = a
         else  begin
2.                ω̄ := 1;
3.                a' = (a_0, a_2, ..., a_{n-2});
4.                a'' = (a_1, a_3, ..., a_{n-1});
5.                y' :=FFT1(a', n/2);
6.                y'' :=FFT1(a'', n/2);
7.                for i := 0 to n/2 − 1 do
                     begin
8.                      y_i := y'_i + ω̄ y''_i;
9.                      y_{n/2+i} := y'_i + ω̄ ω^{n/2} y''_i;
10.                     ω̄ := ω̄ ω;
                     end
11.               F(a) := y;
               end
   end.
```

Let us have a closer look at **FFT1**. Figure 3 shows a tree of input vectors to the recursive calls of the procedure, with an initial vector of length 8. This tree

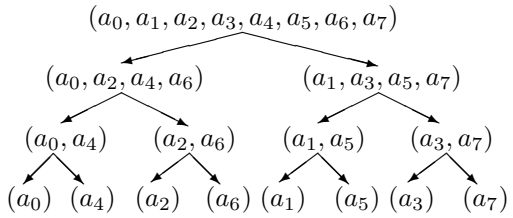


Figure 3: The tree of vectors of the recursive calls

shows where coefficients are moved on the way down. The computation starts at the bottom and traverses the tree to the root. That is, for instance, from (a_0) and (a_4) we compute $a_0 + \omega^0 a_4$, which can be stored into a_0, and $a_0 + \omega^{4+0} a_4$, which can be stored into a_4. From (a_0, a_4) and (a_2, a_6) we compute $a_0 + \omega^0 a_2$, which can be stored into a_0, $a_0 + \omega^{4+0} a_2$, which can be stored into a_2, $a_4 + \omega^2 a_6$, which can be stored into a_4, and $a_4 + \omega^{4+2} a_6$, which can be stored into a_6. Three iterations are needed to reach the root of the tree. After the third one, a_0 is in fact the evaluation of $a(x)$ at ω^0, and so on. These iterations are represented in Figure 4. Let us analyse in more details this example:

- the order $(a_0, a_4, a_2, a_6, a_1, a_3, a_5, a_7)$ can be easily obtained by reversing the binary representations of the numbers $0, 1, 2, 3, 4, 5, 6, 7$. For instance, a_4 takes the place of a_1 because the image mirror of 001 is 100, and so on;
- there are $k = log_2 8$ iterations, and the j^{th} iteration uses the $(n/(2^{j-1}))^{th}$ roots of unity, which are

$$(\omega^0)^{2^{j-1}}, \ldots, (\omega^{n-1})^{2^{j-1}};$$

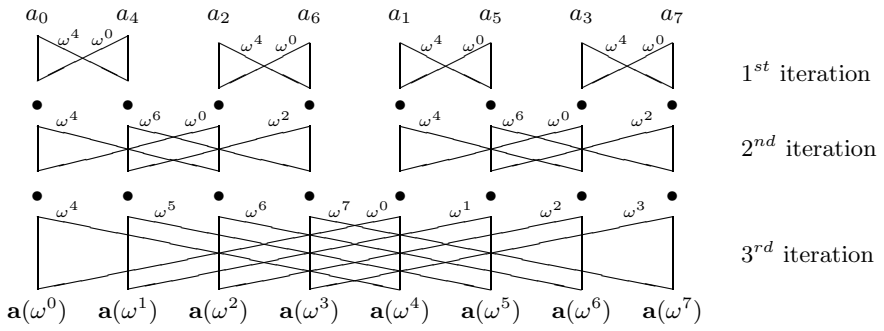


Figure 4: Iterations of **FFT1** for $n = 8$

- at each iteration j, there are 2^{k-j} subvectors of 2^j elements each. The elements of each subvector are re-combined by means of the unity roots that are used at this iteration. The re-combination respects the same rule for each subvector. For example, the subvector (a_0, a_4) at the first iteration in Figure 4 gives rise to the elements $a_0 + \omega^0 a_4$ and $a_0 + \omega^4 a_4$, the subvector (a_2, a_6) gives rise to the elements $a_2 + \omega^0 a_6$ and $a_2 + \omega^4 a_6$, and so on.

These remarks, generalized to an arbitrary integer $n = 2^k$, lead to the following algorithm.

FFT2$(\mathbf{a}, n)$
`input:` $\mathbf{a} = (a_0, \ldots, a_{n-1}) \in R^n$, where $n = 2^k$, $k \geq 1$;
`output:` $F(\mathbf{a})$;
`begin`
1. $\mathbf{a}$:=`bit-reverse`$(\mathbf{a})$;
2. `for` $j := 1$ `to` k `do`
   `begin`
3. $m := 2^j$;
4. $\omega_j := \omega^{2^{k-j}}$;
5. $\bar{\omega} = 1$;
6. `for` $i := 0$ `to` $m/2 - 1$ `do`
   `begin`
7. `for` $s := i$ `to` $n - 1$ `step` m `do`
   `begin`
8. $t := \bar{\omega} a_{s+m/2}$;
9. $u := a_s$;
10. $a_s := u + t$;
11. $a_{s+m/2} := u + \omega^{m/2} t$;
   `end`
12. $\bar{\omega} := \bar{\omega}\omega_j$;
   `end`
   `end`
`end.`

In the algorithm **FFT2**, `bit-reverse` is a very simple procedure which produces the desired order of the input vector. As we have already explained,

this order is obtained just by replacing a_i by $a_{rev(i)}$, where $rev(i)$ is the number obtained from the binary representation of i by reversing its bits.

Line 2 in the algorithm counts the iterations, and for each iteration j it gives the dimension of the subvectors at that iteration and prepares for the computation of the unity roots (used at that iteration). Line 6 prepares for making the re-combinations regarding each subvector, and lines 7–11 do them.

As with **FFT1**, line 11 can be replaced by

$$a_{s+m/2} := u - t,$$

whenever the property $\omega^{n/2} = -1$ holds true in the ring R.

In order to compute $F^{-1}(\mathbf{a})$ we replace ω by ω^{-1} and multiply the result by n^{-1} in the algorithm **FFT2**$(\mathbf{a}, n)$.

If one wants to start with the original order of the input vector, then the re-combination should be performed in a reverse order than the one used in **FFT2**. This order is obtained like in Figure 5. As we can see, the first iteration

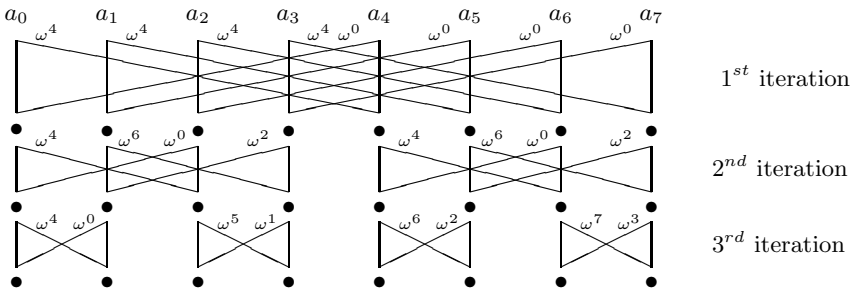


Figure 5: Starting with the original order of the input vector

combines the elements in the same order as the last iteration in Figure 4, and so on. After the last iteration, the resulting vector should be reversed. One can easily design a slight modified version of **FFT2** in order to work in this case too. We shall prefer to present another version, **FFT3**, which shows explicitly how to get, for each iteration j and element a_i, the indexes p, q, and r such that $a_i = a_p + \omega^r a_q$.

FFT3$(\mathbf{a}, n)$
`input:` $\mathbf{a} = (a_0, \ldots, a_{n-1}) \in R^n$, where $n = 2^k$, $k \geq 1$;
`output:` $F(\mathbf{a})$;
`begin`
1. `for` $i := 0$ `to` $2^k - 1$ `do` $y_i := a_i$;
2. `for` $j := 0$ `to` $k - 1$ `do`
`begin`
3. `for` $i := 0$ `to` $2^k - 1$ `do` $z_i := y_i$;
4. `for` $i := 0$ `to` $2^k - 1$ `do`
`begin`
5. `let` $i = (d_{k-1} \cdots d_0)_2$;
6. $p := (d_{k-1}, \ldots, d_{k-j+1}, 0, d_{k-j-1}, \ldots, d_0)_2$;
7. $q := (d_{k-1}, \ldots, d_{k-j+1}, 1, d_{k-j-1}, \ldots, d_0)_2$;

8. $r := (d_{k-j}, \ldots, d_0, 0, \ldots, 0)_2$;
9. $y_i := z_p + \omega^r z_q$;
end
end;
10. $F(\mathbf{a})$:=`bit-reverse`(y);
`end.`

Usually, in implementations, the powers of ω are pre-computed.

FFT in the Complex Field C FFT is intensively used with the field of complex numbers. We recall that any complex number $z = x + iy$ can be written as

$$z = |z|(\cos\theta + i\sin\theta)$$

or

$$z = |z|e^{i\theta},$$

where $|z| = \sqrt{x^2 + y^2}$ and $\tan\theta = y/x$.

The equation $z^n = 1$ has exactly n complex roots, which are $e^{i\frac{2\pi j}{n}}$, for all $0 \leq j < n$. These solutions form a cyclic group under multiplication, and the complex number $\omega = e^{(2\pi i)/n}$ generates this group. It is easy to see that ω is a primitive n^{th} root of unity, and the solutions of the equation $z^n = 1$ are exactly the n^{th} roots of unity (in the sense of Definition 3.3.1). These numbers can be located in the complex plane as the vertices of a regular polygon of n sides inscribed in the unit circle $|z| = 1$ and with one vertex at the point $z = 1$ (see Figure 6 for the case $n = 8$).

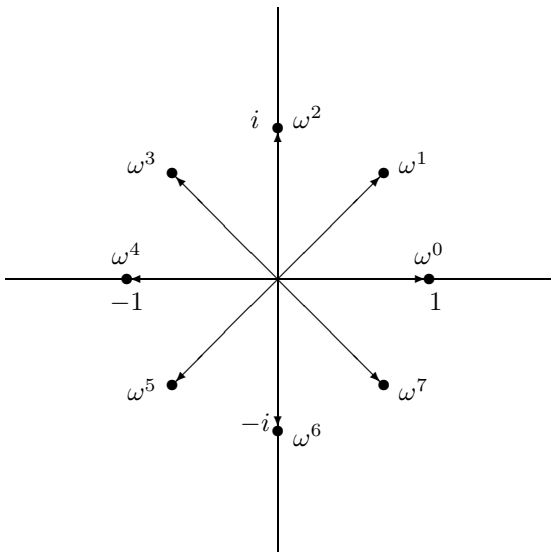


Figure 6: The values $\omega^0, \ldots, \omega^7$, where $\omega = e^{(2\pi i)/8}$

Since $\mathbf{C}$ is a field, we may use all the properties of the n^{th} roots of unity mentioned above, such as

- $\omega^{-1} = \omega^{n-1}$;

- $\omega^{n/2} = -1$, provided that n is even;
- $\omega^{n/2+i} = -\omega^i$ and $(\omega^{n/2+i})^2 = (\omega^i)^2$, for all $0 \leq i < n/2$, provided that n is even;
- ω^2 is a primitive $(n/2)^{th}$ root of unity, provided that n is even.

The FFT algorithms developed in the above subsections work in this case too. Moreover, they may use the property $\omega^{n/2+i} = -\omega^i$, which gives some efficiency in implementations.

Let us consider now the problem of computing ω^j, for all $0 \leq j < n$, where n is a power of 2 (for example, $n = 2^k$).

Denote $\omega_r = e^{(2\pi i)/2^r}$, for all $1 \leq r \leq k$. That is,

$$\omega_r = \omega^{2^{k-r}},$$

for all r. Remark that $\omega_k = \omega$.

If these values have been already computed, then ω^j can be computed by

$$\omega^j = \underbrace{\omega_r \cdots \omega_r}_{q \text{ times}},$$

where $j = 2^{k-r}q$, for some r and odd q. This method could be very inefficient because it implies many multiplications.

We can compute ω^j recursively. For example, if we assume that $\omega^{j'}$ has been computed, for all $j' < j$, then we can write

$$\omega^j = \omega^{j'}\omega_r,$$

where $j' = 2^{k-r}(q-1) < j$. Now, $q-1$ is even and, therefore, j' can be written as $j' = 2^{k-r'}q'$, for some $r' < r$ and odd $q' < q$. This shows that ω_r is not used in computing $\omega^{j'}$. In other words, we can say that there is a sequence

$$r_j > \cdots > r_1 \geq 1$$

such that

$$\omega^j = \omega_{r_1} \cdots \omega_{r_j}.$$

In this way, no more than k multiplications are needed to compute ω^j.

Let us focus now on computing ω_r, for all $1 \leq r \leq k$. Simple computations show that $\omega_1 = -1$ and $\omega_2 = i$.

Denote $\omega_r = x_r + iy_r$, where $x_r = cos(2\pi/2^r)$ and $x_r = sin(2\pi/2^r)$. There are many pairs of recurrence equations for getting ω_{r+1} from ω_r. One of them, proposed in [36], is

$$x_{r+1} = \sqrt{\frac{1+x_r}{2}}, \quad y_{r+1} = \sqrt{\frac{1-x_r}{2}}.$$

In [62], Tate has shown that the equation for y_{r+1} is not stable because x_r approaches 1 rather quickly. Moreover, he proposed an alternative pair of equations,

$$x_{r+1} = \frac{1}{2}\sqrt{2(1+x_r)}, \quad y_{r+1} = \frac{y_r}{\sqrt{2(1+x_r)}}$$

and proved that these equations are stable.

FFT in Finite Fields If we are working in the field of real (or complex) numbers, we must approximate real numbers with finite precision numbers. In this way we introduce errors. These errors can be avoided if we work in a finite ring (or field), like the ring $\mathbf{Z}_m$ of integers modulo m.

The FFT in a finite ring is usually called the *number theoretic transform*, abbreviated NTT.

Theorem 3.3.2 Let n and ω be powers of 2, and $m = \omega^{n/2} + 1$. Then, the following are true:

(1) n has a multiplicative inverse in $\mathbf{Z}_m$;

(2) $\omega^{n/2} \equiv -1 \ mod \ m$;

(3) ω is a principal n^{th} root of unity in $\mathbf{Z}_m$.

Proof (1) and (2) follows directly from the way n, ω, and m have been chosen.

(3) Let $n = 2^k$. By induction on k one can easily prove that the following relation holds in any commutative ring R:

$$\sum_{i=0}^{n-1} a^i = \prod_{i=0}^{k-1} (1 + a^{2^i}),$$

for all $a \in R$. Therefore, it holds in $\mathbf{Z}_m$.

Let $1 \leq i < n$. In order to prove that

$$\sum_{j=0}^{n-1} (\omega^i)^j \equiv 0 \ mod \ m$$

it suffices to show that $1 + \omega^{2^s i} \equiv 0 \ mod \ m$, for some $0 \leq s < k$.

Let $i = 2^t i'$, where i' is odd. By choosing s such that $s + t = k - 1$, one can easily show that $1 + \omega^{2^s i} \equiv 0 \ mod \ m$. $\diamond$

The importance of Theorem 3.3.2 is that the convolution theorem is valid in rings like $\mathbf{Z}_{\omega^{n/2}+1}$, where n and ω are powers of 2. Moreover, the property $\omega^{n/2} \equiv -1 \ mod \ m$ holds true, which is very important for implementations.

Let us assume that we wish to compute the convolution of two n-dimensional vectors $\mathbf{a} = (a_0, \ldots, a_{n-1})$ and $\mathbf{b} = (b_0, \ldots, b_{n-1})$, where n is a power of 2. Then, we have to

- define $\mathbf{a}' = (a_0, \ldots, a_{n-1}, 0, \ldots, 0)$ and $\mathbf{b}' = (b_0, \ldots, b_{n-1}, 0, \ldots, 0)$, which are 2n-ary vectors;
- choose ω as a power of 2, for instance $\omega = 2$. Then, 2 is a primitive $(2n)^{th}$ root of unity in $\mathbf{Z}_{2^n+1}$;
- compute $F^{-1}(F(\mathbf{a}') \cdot F(\mathbf{b}'))$ in $\mathbf{Z}_{2^n+1}$. If the components of $\mathbf{a}$ and $\mathbf{b}$ are in the range 0 to 2^n, then the result is exactly $\mathbf{a} \odot \mathbf{b}$; otherwise, the result is correct modulo $2^n + 1$.

Let us focus now on computing $F(\mathbf{a})$ in $\mathbf{Z}_{\omega^{n/2}+1}$, where $\mathbf{a}$ is an n-dimensional vector over $\mathbf{Z}_{\omega^{n/2}+1}$, and n and ω are powers of 2. Clearly, we can use any FFT

algoritm presented in the subsections above, where the addition and multiplication should be replaced by addition and multiplication modulo $\omega^{n/2}+1$.

Let $m=\omega^{n/2}+1$. Any number $a \in \mathbf{Z}_m$ can be represented by a string of b bits, where

$$b=\frac{n}{2}\log\omega+1.$$

There are $log\, n$ iterations in **FFT2**, and each iteration computes n values by addition and multiplication modulo m. Therefore, there are $\mathcal{O}(n\log n)$ operations.

Addition modulo m is of time complexity $\mathcal{O}(b)$. The only multiplication used is by ω^p, where $0 \leq p < n$. Such a multiplication is equivalent to a left shift by $p\log\omega$ places (remember that ω is a power of 2). The resulting integer has at most $3b-2$ bits because

$$p\log\omega < n\log\omega = 2(b-1)$$

(see the relation above) and the original integer has at most b bits.

The modular reduction $x\ mod\ m$ we have to perform, where x is an integer, is based on the fact that x can be written in the form $\sum_{i=0}^{l-1} x_i(\omega^{n/2})^i$ (as usual, by splitting the binary representation of x into blocks of $(n/2)\log\omega$ bits). Then, we use the following lemma.

Lemma 3.3.1 $\sum_{i=0}^{l-1} x_i(\omega^{n/2})^i \equiv \sum_{i=0}^{l-1} x_i(-1)^i\ mod\ m.$

Proof Remark that $\omega^{n/2} \equiv -1\ mod\ m$. $\diamond$

Therefore, the direct Fourier transform is of time complexity $\mathcal{O}(bn\log n)$, i.e. $\mathcal{O}(n^2(\log n)(\log\omega))$.

Let us take into consideration the inverse Fourier transform. It requires multiplication by ω^{-p} and by n^{-1} (these are computed modulo m).

Since

$$\omega^p\omega^{n-p} \equiv 1\ mod\ m,$$

we have that $\omega^{-p}=\omega^{n-p}$. Therefore, multiplication by ω^{-p} is multiplication by ω^{n-p}, which is a left shift of $(n-p)\log p$ places (the resulting integer still has at most $3b-2$ bits).

Assume $n=2^k$ (we know that n is a power of 2). In order to find y such that

$$2^k y \equiv 1\ mod\ m$$

we start from the remark that $\omega^n \equiv 1\ mod\ m$. Therefore, we may choose y such that $2^k y=\omega^n$, i.e., $y=2^{n\log\omega-k}$. The integer y is the inverse of n modulo m. Multiplying by n^{-1} is equivalent to a left shift of $n\log\omega-k$ places.

The time complexity of the inverse transform is the same as for the direct transform.

4 Division

Division deals with the problem of finding, for two integers a and $b \neq 0$, a representation $a = bq + r$, where $0 \leq r < |b|$. Such a representation always exists and it is unique. The integer q is the *quotient*, and the integer r is the *remainder*, of the division of a by b.

Division is one of the most difficult operations. It can be proven that division has the same complexity as multiplication, but in practice it is slower than multiplication. Much effort has been devoted to find efficient division algorithms. In practice, efficiency depends highly on the length of the operands. For small integers, *school division* is the best. For resonable large integers, *recursive division* is the best known method.

In what follows we shall describe both the school and recursive division. We shall only consider positive dividents $a > 0$ and divisors $b > 0$ because, for all the other cases we have:

- if $a > 0$ and $b < 0$, then $\lfloor a/b \rfloor = -\lfloor a/(-b) \rfloor$ and $a \ mod \ b = a \ mod \ (-b)$;
- if $a < 0$ and $b > 0$, then:
  - if $a \ mod \ b = 0$, then $\lfloor a/b \rfloor = -\lfloor (-a)/b \rfloor$;
  - if $a \ mod \ b > 0$, then $\lfloor a/b \rfloor = -\lfloor (-a)/b \rfloor - 1$ and $a \ mod \ b = b - ((-a) \ mod \ b)$;
- if $a < 0$ and $b < 0$, then
  - if $a \ mod \ b = 0$, then $\lfloor a/b \rfloor = \lfloor (-a)/(-b) \rfloor$;
  - if $a \ mod \ b > 0$, then $\lfloor a/b \rfloor = \lfloor (-a)/(-b) \rfloor - 1$ and $a \ mod \ b = b - ((-a) \ mod \ (-b))$.

4.1 School Division

In order to obtain the quotient and the remainder of the division of a by b we can use repeated subtraction, as follows:

- if $a < b$, then $q = 0$ and $r = a$;
- if $a \geq b$, then $q = 1 + ((a - b) \ div \ b)$ and $r = (a - b) \ mod \ b$.

The *repeated subtraction method* leads to a time-consuming algorithm.

The well-known *school method* for dividing integers uses, as a basic step, the division of a $(k + 1)$-digit integer by a k-digit integer, for some k. In such a case, the main problem is to "guess" efficiently the quotient. Therefore, we can focus on the division of a $(k+1)$-digit integer a by a k-digit integer b satisfying $\lfloor a/b \rfloor < \beta$, where β is the base.

Definition 4.1.1 An integer $b = (b_{k-1}, \ldots, b_0)_\beta$ is called *normalized* if its leading digit b_{k-1} satisfies $b_{k-1} \geq \lfloor \beta/2 \rfloor$.

It is easy to see that a k-digit integer b is normalized iff $\beta^k/2 \leq b < \beta^k$.

The following theorems helps us to estimate efficiently the quotient $q = \lfloor a/b \rfloor$.

Theorem 4.1.1 Let $a = (a_k, a_{k-1}, \ldots, a_0)_\beta$ and $b = (b_{k-1}, \ldots, b_0)_\beta$ be integers such that $q = \lfloor a/b \rfloor < \beta$. Then, the integer

$$\hat{q} = min \left\{ \left\lfloor \frac{a_k\beta + a_{k-1}}{b_{k-1}} \right\rfloor, \beta - 1 \right\}.$$

satisfies the following:

(1) $q \leq \hat{q}$;

(2) $\hat{q} - 2 \leq q$, providing that b is normalized.

Proof (1) Let $a = qb + r$, where $r < b$. If $\hat{q} = \beta - 1$, then $q \leq \hat{q}$ because $q < \beta$. Otherwise, $\hat{q} = \left\lfloor \frac{a_k\beta + a_{k-1}}{b_{k-1}} \right\rfloor$ and, therefore,

$$a_k\beta + a_{k-1} = \hat{q}b_{k-1} + \hat{r},$$

where $\hat{r} < b_{k-1}$. Thus,

$$a_k\beta + a_{k-1} = \hat{q}b_{k-1} + \hat{r} \leq \hat{q}b_{k-1} + b_{k-1} - 1.$$

We shall prove that $a - \hat{q}b < b$, which shows that $q \leq \hat{q}$. We have:

$$\begin{aligned} a - \hat{q}b &\leq a - \hat{q}(b_{k-1}\beta^{k-1}) \\ &= (a_k\beta^k + \cdots + a_0) - (\hat{q}b_{k-1})\beta^{k-1} \\ &\leq (a_k\beta^k + \cdots + a_0) - (a_k\beta + a_{k-1} - b_{k-1} + 1)\beta^{k-1} \\ &= a_{k-2}\beta^{k-2} + \cdots + a_0 + b_{k-1}\beta^{k-1} - \beta^{k-1} \\ &< b_{k-1}\beta^{k-1} \\ &\leq b. \end{aligned}$$

(2) Assume that $b_{k-1} \geq \lfloor b/2 \rfloor$, but $q < \hat{q} - 2$. Then,

$$\hat{q} \leq \frac{a_k\beta + a_{k-1}}{b_{k-1}} = \frac{a_k\beta^k + a_{k-1}\beta^{k-1}}{b_{k-1}\beta^{k-1}} \leq \frac{a}{b_{k-1}\beta^{k-1}},$$

which leads to

$$a \geq \hat{q}b_{k-1}\beta^{k-1} > \hat{q}(b - \beta^{k-1})$$

(the integer b cannot be β^{k-1} because, otherwise, $\hat{q} = q$). Therefore,

$$\hat{q} < \frac{a}{b - \beta^{k-1}}.$$

The relation $q < \hat{q} - 2$ leads to:

$$3 \leq \hat{q} - q < \frac{a}{b - \beta^{k-1}} - q < \frac{a}{b - \beta^{k-1}} - (\frac{a}{b} - 1) = \frac{a}{b}(\frac{\beta^{k-1}}{b - \beta^{k-1}}) + 1$$

which implies

$$\frac{a}{b} > 2\frac{b - \beta^{k-1}}{\beta^{k-1}} \geq 2(b_{k-1} - 1).$$

Next, using $\beta - 1 \geq \hat{q}$, we obtain

$$\beta - 4 \geq \hat{q} - 3 \geq q = \left\lfloor \frac{a}{b} \right\rfloor \geq 2(b_{k-1} - 1),$$

which shows that $b_{k-1} \leq \beta/2 - 1$; a contradiction. $\diamond$

The following theorem follows easily from definitions.

Theorem 4.1.2 Let a and $b = (b_{k-1}, \ldots, b_0)_\beta$ be integers. Then, db is normalized and $\lfloor a/b \rfloor = \lfloor (da)/(db) \rfloor$, where $d = \lfloor \beta/(b_{k-1} + 1) \rfloor$.

If b is normalized, then the integer d in Theorem 4.1.2 is 1.

Let $a = (a_k, a_{k-1}, \ldots, a_0)_\beta$ and $b = (b_{k-1}, \ldots, b_0)_\beta$ be integers such that $q = \lfloor a/b \rfloor < \beta$. The theorems above say that, in order to determine q, one can do as follows:

- replace a by da and b by db (b is normalized now and the quotient is not changed);
- compute $\hat{q}$ and try successively $\hat{q}$, $\hat{q} - 1$ and $\hat{q} - 2$ in order to find q (one of them must be q). If $a - b\hat{q} \geq 0$, then $q = \hat{q}$; otherwise, we compute $a - b(\hat{q} - 1)$. If $a - b(\hat{q} - 1) \geq 0$ then $q = \hat{q} - 1$ else $q = \hat{q} - 2$.

There is a better *quotient test* than the above one, which avoids making too many multiplications. It is based on the following theorem.

Theorem 4.1.3 Let $a = (a_k, a_{k-1}, \ldots, a_0)_\beta > 0$ and $b = (b_{k-1}, \ldots, b_0)_\beta > 0$ be integers, and $q = \lfloor a/b \rfloor$. Then, for any positive integer $\hat{q}$ the following hold:

(1) If $\hat{q}b_{k-2} > (a_k\beta + a_{k-1} - \hat{q}b_{k-1})\beta + a_{k-2}$, then $q < \hat{q}$;

(2) If $\hat{q}b_{k-2} \leq (a_k\beta + a_{k-1} - \hat{q}b_{k-1})\beta + a_{k-2}$, then $q > \hat{q} - 2$.

Proof (1) Assume that $\hat{q}b_{k-2} > (a_k\beta + a_{k-1} - \hat{q}b_{k-1})\beta + a_{k-2}$. Then,

$$\begin{aligned} a - \hat{q}b &\leq a - \hat{q}(b_{k-1}\beta^{k-1} + b_{k-2}\beta^{k-2}) \\ &< a_k\beta^k + a_{k-1}\beta^{k-1} + a_{k-2}\beta^{k-2} + \beta^{k-2} - \hat{q}(b_{k-1}\beta^{k-1} + b_{k-2}\beta^{k-2}) \\ &= \beta^{k-2}((a_k\beta + a_{k-1} - \hat{q}b_{k-1})\beta + a_{k-2} - \hat{q}b_{k-2} + 1) \\ &\leq 0 \end{aligned}$$

(the last inequality follows from the hypothesis). Therefore, $q < \hat{q}$.

(2) Assume that $\hat{q}b_{k-2} \leq (a_k\beta + a_{k-1} - \hat{q}b_{k-1})\beta + a_{k-2}$, but $q \leq \hat{q} - 2$. Then,

$$\begin{aligned} a &< (\hat{q} - 1)b \\ &< \hat{q}(b_{k-1}\beta^{k-1} + b_{k-2}\beta^{k-2}) + \beta^{k-1} - b \\ &\leq \hat{q}b_{k-1}\beta^{k-1} + ((a_k\beta + a_{k-1} - \hat{q}b_{k-1})\beta + a_{k-2})\beta^{k-2} + \beta^{k-1} - b \\ &= a_k\beta^k + a_{k-1}\beta^{k-1} + a_{k-2}\beta^{k-2} + \beta^{k-1} - b \\ &\leq a_k\beta^k + a_{k-1}\beta^{k-1} + a_{k-2}\beta^{k-2} \\ &\leq a, \end{aligned}$$

which is a contradiction (the third inequality follows from hypothesis). $\diamond$

Before describing a division algorithm based on the above ideas, one more thing has to be treated carefully: the requirement $\lfloor a/b \rfloor < \beta$.

Let $a = (a_k, a_{k-1}, \ldots, a_0)_\beta$ and $b = (b_{k-1}, \ldots, b_0)_\beta$. We have two cases:

1. If b is normalized, then the integers $a' = (0, a_k, \ldots, a_1)_\beta$ and b satisfy $\lfloor a'/b \rfloor < \beta$;

2. If b is not normalized and d is chosen as in Theorem 4.1.2, then the integers $(da)[k+1:1]$ (consisting of the $k+1$ most significant digits of da) and db satisfy $\lfloor (da)[k+1:1]/(db) \rfloor < \beta$.

The algorithm **SchoolDiv1** applies all these ideas.

SchoolDiv1(a, b)
`input:` $a = (a_{k+m-1}, \ldots, a_0)_\beta > 0$ and $b = (b_{k-1}, \ldots, b_0)_\beta > 0$,
where $k \geq 2$ and $m \geq 1$;
`output:` $q = (q_m, \ldots, q_0)_\beta$ and r such that $a = bq + r$ and $0 \leq r < b$;
`begin`

```
 1.   d := ⌊β/(b_{k-1} + 1)⌋;
 2.   a := da;
 3.   let a = (a_{k+m}, ..., a_0)_β   (a_{k+m} = 0 if d = 1);
 4.   b := db;
 5.   let b = (b_{k-1}, ..., b_0)_β;
 6.   x := (a_{k+m}, ..., a_m)_β;
 7.   for i := m downto 0 do
        begin
 8.        if b_{k-1} = x_k then q̂ := β − 1 else q̂ := ⌊(x_k β + x_{k-1})/b_{k-1}⌋;
 9.        if b_{k-2} q̂ > (x_k β + x_{k-1} − b_{k-1} q̂)β + x_{k-2}
10.           then  begin
11.                    q̂ := q̂ − 1;
12.                    if b_{k-2} q̂ > (x_k β + x_{k-1} − b_{k-1} q̂)β + x_{k-2}
13.                       then q_i := q̂ − 1
14.                       else if x − b q̂ < 0 then q_i := q̂ − 1 else q_i := q̂
15.                 end
16.           else  if x − b q̂ < 0 then q_i := q̂ − 1 else q_i := q̂;
17.        let x − b q_i = (r_{k-1}, ..., r_0)_β;
18.        if i > 0 then x := (r_{k-1}, ..., r_0, a_{i-1})_β;
        end
19.   r := ⌊(x − b q_0)/d⌋;
end.
```

In algorithm **SchoolDiv1**, line 8 computes the estimate $\hat{q}$, and lines 9–16 performes the quotient test (and outputs the real value of the quotient at each step). Notice that the requirement $\lfloor x/b \rfloor < \beta$ is fulfilled at each iteration of the loop 8–18.

The variable x has been only introduced for the sake of clarity. By a careful implementation, its value at the iteration i in loop 8–18 can be computed directly in $(a_{k+i}, \ldots, a_i)_\beta$.

The algorithm **SchoolDiv1** does not take into consideration the case of one digit divisors. For this case we consider the algorithm **SchoolDiv2**.

SchoolDiv2(a, b)
`input:` $a = (a_{k-1}, \ldots, a_0)_\beta > 0$ and $0 < b < \beta$, where $k \geq 1$;
`output:` $q = (q_{k-1}, \ldots, q_0)_\beta$ and r such that $a = bq + r$ and $0 \leq r < b$;
`begin`
1. $r := 0$;
2. `for` $i := k - 1$ `downto` 0 `do`
   `begin`
3. $q_i := \lfloor (r\beta + a_i)/b \rfloor$;
4. $r := (r\beta + a_i)\ mod\ b$;
   `end`

`end.`

Let us discuss now the time complexity of division. Denote by $D(n)$ the cost of division of a $2n$-bit integer by an n-bit integer.

In order to obtain the quotient $\lfloor a/b \rfloor$, where a is a $2n$-bit integer and b is an n-bit integer b, we write

$$\left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor \frac{a}{2^{2n-1}} \cdot \frac{2^{2n-1}}{b} \right\rfloor.$$

$a/2^{2n-1}$ can be easily obtained by left shifts. Therefore, we have to estimate the cost of getting the quotient $\lfloor 2^{2n-1}/b \rfloor$. Denote by $C(n)$ this cost.

We may assume, without loss of the generality, that n is even, and let $n = 2m$ and $b = 2^m b' + b''$, where $b' \geq 2^{m-1}$. Then, we have

$$\frac{2^{2n-1}}{b} = \frac{2^{4m-1}}{2^m b' + b''} = \frac{2^{4m-1}}{2^m b'} \cdot \frac{2^m b'}{2^m b' + b''} = \frac{2^{3m-1}}{b'} \cdot \frac{2^m b'}{2^m b' + b''}.$$

Let $\alpha = 2^m b'/(2^m b' + b'')$ and $x = b''/(2^m b')$. It is easy to see that $\alpha = 1/(x + 1)$, which can be written as

$$\alpha = 1 - x + x^2 - \cdots$$

Thus,

$$\frac{2^{2n-1}}{b} = \frac{2^{3m-1}}{b'}(1 - x + x^2 - \cdots)$$

Since $0 \leq x < 2^{-m+1}$, we have

$$0 \leq \frac{2^{2n-1}}{b} < \frac{2^{3m-1}}{b'}(1 - x) + 4.$$

We look now to an estimate for $(2^{3m-1}/b')(1 - x)$. Let $2^{3m-1}/b' = a + \epsilon$, for some integer a and $0 \leq \epsilon < 1$. Then,

$$\frac{2^{3m-1}}{b'}(1 - x) = 2^m a + 2^m \epsilon - \frac{2^{2m-1} b''}{b'^2},$$

where

$$2^m \epsilon = (2^{2m-1} - ab') \cdot \frac{2^{2m-1}}{b'} \cdot 2^{-m+1} = (2^{2m-1} - ab') \cdot a \cdot 2^{-m+1} + \epsilon'$$

and $0 \leq \epsilon' < 2$.

According to these formulas, the quotient q can be obtained from a, ab', $(2^{2m-1} - ab') \cdot a$, and $\lfloor (2^{2m-1}b'')/(b'^2) \rfloor$, and some extra simple operations. The cost of these auxiliary operations is linear to n. Therefore, we can write

$$C(n) \leq 2C(n/2) + 3M(n/2) + cn,$$

for some positive constant c. Assuming $M(2k) \geq 2M(k)$ for all k, we obtain

$$C(n) = \mathcal{O}(M(n)) + \mathcal{O}(n \ln n).$$

If we add the hypothesis $M(n) \geq c'n \ln n$ for some positive constant c', the previous estimate leads to

$$C(n) = \mathcal{O}(M(n)).$$

Therefore, $D(n) = \mathcal{O}(M(n))$, which shows that the cost of the division is dominated by the cost of multiplication, up to a multiplicative constant.

4.2 Recursive Division

The Karatsuba multiplication is based on partitioning the integers and performing more multiplications with smaller integers. A similar idea can be applied to division as well, in the sense that the quotient $q = \lfloor a/b \rfloor$ can be obtained by $q = q_1\beta^k + q_0$, where $q_1 = \lfloor a_1/b \rfloor$, $q_0 = \lfloor (r_1\beta^k + a_0)/b \rfloor$, and $a = a_1\beta^k + a_0$, for some k. This method has been discovered in 1972 by Moenck and Borodin [43], who espressed it only in the case of $\mathcal{O}(n^{1+\epsilon})$ multiplication. Later, in 1997, Jebelean [32] expressed the same algorithm in the Karatsuba case, and finally, in 1998, Burnizel and Ziegler [10] worked out the details for a practical implementation. This algorithm, which should be called the *Moenck-Borodin-Jebelean-Burnikel-Ziegler division algorithm* as Zimmermann pointed out in [71], provides the fastest solution to the division problem.

Let $a > 0$ and $b > 0$ be integers. Then, for any $k \geq 1$ there are integers a_1, a_0, q_1, q_0, r_1, and r_0 such that the following are true:

$$\begin{aligned} a &= a_1\beta^k + a_0 && 0 \leq a_0 < \beta^k \\ &= (q_1 b + r_1)\beta^k + a_0 && 0 \leq r_1 < b \\ &= q_1\beta^k b + (r_1\beta^k + a_0) \\ &= q_1\beta^k b + q_0 b + r_0 && 0 \leq r_0 < b \\ &= (q_1\beta^k + q_0)b + r_0. \end{aligned}$$

Therefore, the quotient of the division of a by b can be obtained as follows: decompose a into two blocks a_1 and a_0 ($a = a_1\beta^k + a_0$), find $q_1 = \lfloor a_1/b \rfloor$ and $q_0 = \lfloor (r_1\beta^k + a_0)/b \rfloor$, and then combine these two quotients via β^k.

The algorithm $\mathbf{D}_{2/1}$ exploits this simple idea. It finds the quotient of the division of an at most $2m$-digit integer by an m-digit integer (that is, the base β representation of the divident is at most double than that of the divisor). The constant m_0 used in this algorithm depends on the machine and implementation, and it is assumed that school division is fast for $m \leq m_0$. The algorithm $\mathbf{D}_{3/2}$, also used in $\mathbf{D}_{2/1}$, outputs the quotient and the remainder of the division of two integers x and y such that $2|y|_\beta \geq 3|x|_\beta$ (it will be described later).

$\mathbf{D}_{2/1}(a,b)$
input: $a > 0$ and $b > 0$ such that $a/b < \beta^m$ and $\beta^m/2 \le b < \beta^m$;
output: $q = \lfloor a/b \rfloor$ and $r = a - qb$;
begin
1. if (m is odd) or ($m \le m_0$) then **SchoolDiv**(a,b)
   else begin
2. decompose $a = a_1\beta^{m/2} + a_0$, where $a_0 < \beta^{m/2}$;
3. $(q_1, r_1) := \mathbf{D}_{3/2}(a_1, b)$;
4. $(q_0, r_0) := \mathbf{D}_{3/2}(r_1\beta^{m/2} + a_0, b)$;
5. $q := q_1\beta^{m/2} + q_0$;
6. $r := r_0$;
   end
end.

Both requiremets "$a/b < \beta^m$" and "$\beta^m/2 \le b < \beta^m$" in the algorithm $\mathbf{D}_{2/1}$ are particularly important for the algorithm $\mathbf{D}_{3/2}$ (see the theorem below).

In order to describe the algorithm $\mathbf{D}_{3/2}$ we need the following theoretical result, whose proof can be found in [10].

Theorem 4.2.1 Let $a > 0$, $b > 0$, and $0 < k < |b|_\beta$ be integers, and let

(1) $a = a_1\beta^k + a_0$, where $a_0 < \beta^k$,

(2) $b = b_1\beta^k + b_0$, where $b_0 < \beta^k$.

If $a/b < \beta^{|b_1|_\beta}$ and b is normalized, then

$$\hat{q} - 2 \le q \le \hat{q},$$

where $q = \lfloor a/b \rfloor$ and $\hat{q} = \lfloor a_1/b_1 \rfloor$. Moreover, if $q < \beta^{|b_1|_\beta}/2$, then

$$\hat{q} - 1 \le q \le \hat{q}.$$

This theorem says that for any integer a and any $(l+k)$-digit integer b, if b is normalized and the quotient $q = \lfloor a/b \rfloor$ fits into l digits, then we can make a good guess about this quotient if we divide all but the last k digits of a by the first l digits of b.

$\mathbf{D}_{3/2}(a,b)$
input: $a > 0$ and $b > 0$ such that $a/b < \beta^m$ and $\beta^{2m}/2 \le b < \beta^{2m}$;
output: $q = \lfloor a/b \rfloor$ and $r = a - qb$;
begin
1. decompose $a = a_2\beta^{2m} + a_1\beta^m + a_0$, where $a_2, a_1, a_0 < \beta^m$;
2. decompose $b = b_1\beta^m + b_0$, where $b_0 < \beta^m$;
3. if $a_2 < b_1$ then $(q,r) := \mathbf{D}_{2/1}(a_2\beta^m + a_1, b_1)$;
4. else begin $q := \beta^m - 1$; $r := (a_2\beta^m + a_1) - qb_1$ end;
5. $d := qb_0$;
6. $r := r\beta^m + a_0 - d$;
7. while $r < 0$ do
   begin
8. $r := r + b$;
9. $q := q - 1$;
   end
end.

The multiplication in line 5 can be done by the Karatsuba algorithm. Lines 3 and 4 estimate the quotient, and lines 5–9 test it for the real value.

Let us now analyze the running time of algorithm $\mathbf{D}_{2/1}$. Let $M(m)$ denote the time to multiply two m-digit integers, and $D(m)$ the time to divide a $2m$-digit integer by an m-digit integer. The operations in lines 1 and 2 are done in $\mathcal{O}(m)$ (or even in time $\mathcal{O}(1)$ if we use clever pointer arithmetic and temporary space management). $\mathbf{D}_{3/2}$ makes a recursive call in line 3 which takes $D(m/2)$. The backmultiplication qb_0 in line 5 takes $M(m/2)$. The remaining additions and subtractions take $\mathcal{O}(m)$. Let c be a convenient constant so that the overall time for all splitting operations, additions, and subtractions is less than cm. $\mathbf{D}_{3/2}$ is called twice by $\mathbf{D}_{2/1}$, so we have the recursion

$$\begin{aligned} D(m) &\leq 2D(m/2) + 2M(m/2) + cm \\ &\leq 2[2D(m/4) + 2M(m/4) + cm/2] + 2M(m/2) + cm \\ &= 4D(m/4) + 4M(m/4) + 2M(m/2) + 2cm/2 + cm \\ &\leq \ldots \\ &\leq 2^{\log m} D(1) + \textstyle\sum_{i=1}^{\log m} 2^i M(m/2^i) + \sum_{i=1}^{\log m} cm \\ &= \mathcal{O}(m) + \textstyle\sum_{i=1} \log m 2^i M(m/2^i) + \mathcal{O}(m \log m) \end{aligned}$$

At this point we see that the running time depends on the multiplication method we use. If we use ordinary school method, we have $M(m) = m^2$ and

$$D(m) = m^2 + \mathcal{O}(m \log m),$$

which is even a little bit worse than school division.

If we use Karatsuba multiplication with $M(m) = K(m) = \mathcal{O}(m^{\log_2 3})$, the above sum has the upper bound

$$D(m) = 2K(m) + \mathcal{O}(m \log m)$$

because in this case $K(m/2^i) = K(m)/3^i$.

We consider now the problem of dividing an arbitrary n-digit integer a by an arbitrary m-digit integer b. The first idea would be to pad a by zeros to the left such that its length becomes a multiple of m. Let n' be the length of a after padding with zeros, and let $[u_{k-1}, \ldots, u_0]_{\beta^m}$ be its decomposition into blocks of length m each, where $n' = mk$. Then, the idea is to apply $\mathbf{D}_{2/1}$ to $(u_{k-1})_\beta \beta^m + (u_{k-2})_\beta$ and b getting a quotient q_{k-2} and a remainder r_{k-2}, then to apply $\mathbf{D}_{2/1}$ to $(r_{k-2})_\beta \beta^m + (u_{k-3})_\beta$ and b getting a quotient q_{k-3} and a remainder r_{k-3}, and so on. But, we notice that $\mathbf{D}_{2/1}(x, y)$ calls $\mathbf{D}_{3/2}(x', y)$ which, in returns, calls $\mathbf{D}_{2/1}(x'', y_1)$, for some x', x'', and y, where y_1 is the high-order half of y. Therefore, the length of y should be even. The same holds for y_1 too. That is, a call of $\mathbf{D}_{2/1}$ with y_1 is a divisor leads to a call of $\mathbf{D}_{3/2}$ and then to a call of $\mathbf{D}_{2/1}$ with the divisor y_{11}, the high-order half of y_1. Therefore, the length of y_1 should be even. The recursive calls stop when the divisor length is less than or equal to the division limit m_0, when school division is faster.

As a conclusion, the divisor b should be padded with zeros to the left until its length becomes a power of two multiple of m_0. Then, a should be padded with zeros to the left until its length becomes a multiple of m'.

All these lead to the algorithm **RecursiveDiv**.

RecursiveDiv(a, b)
`input:` $a > 0$ of length n and $b > 0$ of length m;
`output:` $q = \lfloor a/b \rfloor$ and $r = a - qb$;
`begin`

1. $j := min\{i | 2^i m_0 > m\}$;
2. $l := \lceil m/2^j \rceil$;
3. $m' := 2^j l$;
4. $d := max\{i | 2^i b < \beta^{m'}\}$;
5. $b := b2^d$;
6. $a := a2^d$;
7. $k := min\{i \geq 2 | a < \beta^{im'}/2\}$;
8. `decompose` $a = a_{k-1}\beta^{m'(k-1)} + \ldots + a_1\beta^{m'} + a_0$;
9. $x := a_{k-1}\beta^{m'} + a_{k-2}$;
10. `for` $i := k - 2$ `downto` 0 `do`
    `begin`
11. $(q_i, r_i) := \mathbf{D}_{2/1}(x, b)$;
12. `if` $i > 0$ `then` $x := r_i\beta^{m'} + a_{i-1}$;
    `end`;
13. $q := q_{k-2}\beta^{m'(k-2)} + \ldots + q_1\beta^{m'} + q_0$;
14. $r := \lfloor r_0/2^d \rfloor$;

`end`.

Line 5 in algorithm **RecursiveDiv** normalizes b, while line 6 shifts a by the same amount as b.

Let us analyze the running time of **RecursiveDiv**. First of all we notice that $k \leq max\{2, \lceil (n+1)/m \rceil\}$ (using the notation in the algorithm). Indeed, we may assume that the highest digit b_{s-1} of b is nonzero. Then, our choice of k ensures $k \leq \lceil (n+m'-m+1)/m' \rceil$ because, after shifting, $a < \beta^{n+m'-m+1}/2$. If $n \leq m+1$, we clearly have $k = 1$. Otherwise, it is easy to see that the following is true

$$\frac{n+m'-m+1}{m'} \leq \frac{n+1}{m},$$

which proves that $k \leq max\{2, \lceil (n+1)/m \rceil\}$.

The algorithm $\mathbf{D}_{2/1}$ is called $k-1$ times and, because $m' \leq 2m$, every execution of $\mathbf{D}_{2/1}$ takes time

$$D(m') \leq D(2m) \leq K(2m) + \mathcal{O}(2m \log 2m).$$

With $K(m) = c \cdot m^{\log 3}$, for some constant c, we conclude that the overall running time is

$$\mathcal{O}((n/m)(K(m) + \mathcal{O}(m \log m))) = \mathcal{O}(nm^{\log 3 - 1} + n \log m)$$

which is a remarkable asymptotic improvement over the $\Theta(nm)$ time bound for ordinary school division.

5 The Greatest Common Divisor

The *greatest common divisor* (GCD) of two integers a and b not both zero, denoted by $gcd(a, b)$, is the largest integer d such that d divides both a and b. In particular, $gcd(a, 0) = gcd(0, a) = |a|$, and $gcd(a, b) = gcd(|a|, |b|)$. Hence, we can always assume that a and b are non-negative.

The most well-known algorithm for computing GCD is the Euclidean algorithm [4]. In 1938, Lehmer [39] proposed a method to apply the Euclidean algorithm to large integers. Then, in 1967 Stein [60] discovered a GCD algorithm well suited for single-precision integers employing binary arithmetic. All of these algorithms have $\mathcal{O}(n^2)$ time complexity on two n-bit integers. In 1971, Schönhage [55] described an $\mathcal{O}(n(\log n)^2(\log\log n))$ time complexity GCD algorithm using FFT, but the algorithm is not considered practical. With the emphasis on parallel algorithms, Sorenson [58] generalized in 1994 the binary algorithm proposing the k-ary GCD algorithm. This algorithm (and its variants) has practical and efficient sequential versions of time complexity $\mathcal{O}(n^2/(\log n))$.

5.1 The Euclidean GCD Algorithm

Euclid's algorithm for computing the greatest common divisor of two integers is probably the oldest and most important algorithm in number theory. In [36], Knuth "classifies" this algorithm as

> "... the grand daddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day."

Based on the remark that $gcd(a, b) = gcd(b, a \bmod b)$, the Euclidean algorithm consists in performing successive divisions until the remainder becomes 0. The last non-zero remainder is $gcd(a, b)$. The algorithm is given below.

```
   Euclid(a, b)
   input:  a ≥ b > 0;
   output: gcd(a, b);
   begin
1.    while b > 0 do
         begin
2.          r := a mod b;  a := b;  b := r;
         end
3.    gcd(a, b) := a
   end.
```

The number of division steps performed by the algorithm **Euclid** was the subject of some very interesting research. Finally, in 1844, Lamé [38] was able to determine the inputs which elicit the worst-case behavior of the algorithm.

Theorem 5.1.1 (Lamé, 1844 [38])
Let $a \geq b > 0$ be integers. The number of division steps performed by **Euclid**(a, b) does not exceed 5 times the number of decimal digits in b.

[4] Described in Book VI of Euclid's *Elements*.

Proof Let $r_0 = a$, $r_1 = b$, and

$$\begin{aligned} r_0 &= r_1 q_1 + r_2, & 0 < r_2 < r_1 \\ r_1 &= r_2 q_2 + r_3, & 0 < r_3 < r_2 \\ &\cdots \\ r_{n-2} &= r_{n-1} q_{n-1} + r_n, & 0 < r_n < r_{n-1} \\ r_{n-1} &= r_n q_n + r_{n+1}, & r_{n+1} = 0 \end{aligned}$$

be the division steps performed by the algorithm **Euclid**(a, b). Then, $gcd(a, b) = r_n$. We remark that $q_n \geq 2$ and $q_i \geq 1$, for all $1 \leq i \leq n-1$.

Let $(F_i)_{i \geq 1}$ be the Fibonacci sequence, given by $F_1 = F_2 = 1$, and

$$F_{i+2} = F_{i+1} + F_i,$$

for all $i \geq 1$. Then, $r_n \geq 1 = F_2$ and $r_{n-1} \geq 2 = F_3$ which lead to

$$r_{n-2} \geq r_{n-1} + r_n \geq F_2 + F_3 = F_4.$$

By induction on i we can prove easily that $r_i \geq F_{n-i+2}$, for all i. Therefore, $b = r_1 \geq F_{n+1}$. Now, using the fact that

$$F_{i+2} > R^i,$$

for all $i \geq 1$, where $R = (1 + \sqrt{5})/2$, we get

$$b = r_1 \geq F_{n+1} > R^{n-1},$$

which leads to

$$\log_{10} b > (n-1) \log_{10} R > \frac{n-1}{5}$$

by the fact that $\log_{10} R > 1/5$. Therefore, $n < 5 \log_{10} + 1$, which proves the theorem. $\diamond$

Let $N \geq 0$ be a natural number. The Euclidean algorithm performs $\mathcal{O}(\log N)$ division steps on inputs a and b with $0 < b \leq a \leq N$. The naive bit complexity of a division step with integers less than N is $\mathcal{O}((\log N)^2)$. Therefore, the time complexity of Euclid's algorithm is $\mathcal{O}((\log N)^3)$.

A careful analysis shows that the Euclidean algorithm performs better. Dividing r_i by r_{i+1} to get a quotient of q_{i+1} and a remainder of r_{i+2} can be done in $\mathcal{O}((\log r_{i+1})(\log q_{i+1}))$ (using the notation in the proof of Theorem 5.1.1). Then,

$$\begin{aligned} \textstyle\sum_{i=0}^{n-1} (\log r_{i+1})(\log q_{i+1}) &\leq (\log b) \textstyle\sum_{i=0}^{n-1} \log q_{i+1} \\ &\leq (\log b)(\log q_1 \cdots q_n) \end{aligned}$$

It is not hard to see that

$$q_1 \cdots q_n \leq a,$$

which leads to

Theorem 5.1.2 Let $a \geq b > 0$. The time complexity of Euclid's algorithm on inputs a and b is $\mathcal{O}((\log a)(\log b))$.

It is well-known that the greatest common divisor of two numbers a and b can be written as a linear combination $gcd(a,b) = ua + vb$, for some $u, v \in \mathbf{Z}$. Two integers u and v satisfying this property can be easily computed using Euclid's algorithm. This is very important in practice, for example in solving linear diophantine equations $ax + by = c$, where $a, b, c \in \mathbf{Z}$, or in computing the modular multiplicative inverse. Let us give a few details about these two facts.

It is well-known that a linear diophantine equation $ax + by = c$ with integer coefficients has solutions (in $\mathbf{Z}$) if and only if $gcd(a,b)|c$. If this happens, then $x = uc'$ and $y = vc'$ is a solution, where $\gcd(a,b) = ua + vb$ and $c = gcd(a,b)c'$.

Let $a \in \mathbf{Z}$ and $m \geq 2$. It is known that a has an inverse modulo m if and only if $gcd(a,m) = 1$. Finding an inverse modulo m of a, when it exists, reduces to solving the equation

$$ax \equiv 1 \ mod \ m$$

or, equivalently,

$$ax - my = 1$$

in nedeterminates x and y. This equation has solutions (in $\mathbf{Z}$) if and only if $gcd(a,m) = 1$ and, if this happens, any solution $x = x_0$ of it is an inverse modulo m of a.

Let us go back now to the problem of finding a linear combination of $gcd(a,b)$. If we analyze the remainder sequence in Euclid's algorithm we notice that:

$$\begin{aligned}
r_0 &= 1 \cdot a + 0 \cdot b \\
r_1 &= 0 \cdot a + 1 \cdot b \\
r_2 &= r_0 - r_1 q_1 &&= 1 \cdot a + (-q_1) \cdot b \\
r_3 &= r_1 - r_2 q_2 &&= (-q_2) \cdot a + (1 + q_1 q_2) \cdot b
\end{aligned}$$

and so on (using the notation in the proof of Theorem 5.1.1). In other words, along with getting the remainder (at a step) we can also find a linear combination of it (with respect to a and b).

What needs to be done next is to find an elegant way of writing the remainder's linear combination based on the ones in the previous steps. If to each element x that appears in the remainder sequence above we associate a 2-ary vector $V_x = (u, v)$ that gives the linear combination of it with respect to a and b, i.e., $x = ua + vb$, then the linear combination of the remainders can be determined by:

$$\begin{aligned}
& & & & V_{r_0} &= (1,0) \\
& & & & V_{r_1} &= (0,1) \\
r_2 &= r_0 - r_1 q_1 & & & V_{r_2} &= V_{r_0} - q_1 V_{r_1} \\
r_3 &= r_1 - r_2 q_2 & & & V_{r_3} &= V_{r_1} - q_2 V_{r_2} \\
&\dots & & & & \\
r_n &= r_{n-2} - r_{n-1} q_{n-1} & & & V_{r_n} &= V_{r_{n-2}} - q_{n-1} V_{r_{n-1}}
\end{aligned}$$

In this way we can determine $gcd(a,b)$ as well as a linear combination (with respect to a and b) for it. Thus, we have obtained *Euclid's Extended Algorithm*, **EuclidExt**.

EuclidExt(a, b)
`input:` $a \geq b > 0$;
`output:` $gcd(a, b)$ and $V = (u, v)$ such that $gcd(a, b) = ua + vb$;
`begin`
1. $V_0 := (1, 0)$;
2. $V_1 := (0, 1)$;
3. `while` $b > 0$ `do`
   `begin`
4. $q := a \; div \; b$; $r := a \; mod \; b$; $a := b$; $b := r$;
5. $V := V_0$; $V_0 := V_1$; $V_1 := V - qV_1$;
   `end`
5. $gcd(a, b) := a$;
6. $V := V_0$;
`end`.

The correctness of **EuclidExt** can be proved immediately, based on the specifications above. It has the same complexity as Euclid's algorithm. More precisely, at each step, besides a division, we perform two multiplications (a multiplication having the same complexity as a division) and two subtractions (the complexity of a subtraction being linear to the maximum length of the binary representation of the operands).

The sequences $(u_i)_{i\geq 0}$ and $(v_i)_{i\geq 0}$ given by $V_{r_i} = (u_i, v_i)$ for all $0 \leq i \leq n$, are called the *cosequences* associated to $r_0 = a$ and $r_1 = b$ (using the notation above). These sequences can be computed independently one of the other due to the recurrence equation they satisfy, that is

$$V_{r_{i+2}} = V_{r_i} - q_{i+1}V_{r_{i+1}},$$

for all $0 \leq i \leq n - 2$. This is important because, in some cases, computing only one of the cosequences can speed up the whole computation. For example, computing only the first cosequence $(u_i)_{i\geq 0}$ we get u as the last but one element of it. Then, $v = (gcd(a, b) - ua)/b$.

We close the section by mentioning once more that three very important kinds of sequences are associated with two integers $a \geq b > 0$:

- the *quotient sequence* $(q_i)_{i\geq 1}$,
- the *remainder sequence* $(r_i)_{i\geq 0}$, and
- two *cosequences* $(u_i)_{i\geq 0}$ and $(v_i)_{i\geq 0}$.

These sequences are intensively used in order to improve the Euclidean algorithm.

5.2 Lehmer's GCD Algorithm

For multi-precision inputs, the Euclidean algorithm is not the best choice in practice. This is because a multi-precision division operation is relatively expensive. In 1938, Lehmer [39] proposed an interesting method by which consecutive

multi-precision division steps are replaced by consecutive single-precision division steps. In order to understand his method let us recall that the Euclidean GCD algorithm is based on computing a *remainder sequence*,

$$\begin{aligned} r_0 &= a \\ r_1 &= b \\ r_i &= r_{i-2} \ mod \ r_{i-1} \end{aligned}$$

for all $2 \leq i \leq n+1$ and any inputs $a \geq b > 0$, where $r_n \neq 0$ and $r_{n+1} = 0$. Then, $gcd(a, b) = r_n$.

A *quotient sequence* is implicitly computed along with this remainder sequence,

$$q_{i+1} = \left\lfloor \frac{r_i}{r_{i+1}} \right\rfloor,$$

for all $1 \leq i \leq n$.

In general, if a and b are multi-precision numbers, many of the computations above are multi-precision division steps, and in most cases quotients are single-precision integers. Finding the quotient q of two multi-precision integers a and b is much more expensive than computing $a - qb$, assuming that q is a single-precision integer.

Therefore, if we are able to find the quotient sequence $q_1, \ldots, q_n$ in a less expensive way, then we have the chance to significantly improve the speed of the Euclidean algorithm.

Given two multi-precision integers a and b, Lehmer had the idea to consider two single-precision approximations $\hat{r}_0$ and $\hat{r}_1$ of a and b, and to compute a quotient sequence $\hat{q}_1, \ldots, \hat{q}_i$ as long as $q_1 = \hat{q}_1, \ldots, q_i = \hat{q}_i$. Remark that computing such a quotient sequence involves single-precision division steps. If this quotient sequence is computed, then the integers a and b can be replaced by corresponding integers and the procedure can be repeated. Three main problems are to be settled up:

1. how to choose $\hat{r}_0$ and $\hat{r}_1$;
2. how to test the equality $\hat{q}_j = q_j$ without computing q_j (a test for doing that will be called a *quotient test*);
3. how to compute new a and b values when the quotient sequence $\hat{q}_1, \ldots, \hat{q}_i$ cannot be extended any longer (i.e., $\hat{q}_{i+1} \neq q_{i+1}$).

Lehmer's approach to the problems above is the following. Let $a \geq b > 0$ be two multi-precision base β integers. Assume that h is chosen such that $\hat{a} = \lfloor a/\beta^h \rfloor < \beta^p$ and $\hat{b} = \lfloor b/\beta^h \rfloor < \beta^p$ are single-precision integers.

Then, consider $\hat{r}_0 = \hat{a} + A_1$ and $\hat{r}_1 = \hat{b} + C_1$, where $A_1 = 1$ and $C_1 = 0$. In order to check that the quotient $\hat{q}_1 = \lfloor \hat{r}_0/\hat{r}_1 \rfloor$ is the same with the quotient q_1 of r_0 and r_1, we make use of the inequalities

$$\frac{\hat{a} + B_1}{\hat{b} + D_1} < \frac{a}{b} = \frac{r_0}{r_1} < \frac{\hat{a} + A_1}{\hat{b} + C_1} = \frac{\hat{r}_0}{\hat{r}_1},$$

where $B_1 = 0$ and $D_1 = 1$. That is, we compute $\hat{q}'_1 = \lfloor (\hat{a} + B_1)/(\hat{b} + D_1) \rfloor$ and check whether or not $\hat{q}_1 = \hat{q}'_1$. If the test passes, then we know that $\hat{q}_1 = q_1$. Assuming that this happens, we compute further

$$\hat{r}_2 = \hat{r}_0 - \hat{q}_1 \hat{r}_1 = (\hat{a} - \hat{q}_1 \hat{b}) + (A_1 - \hat{q}_1 C_1),$$

and continue with $\hat{r}_1$ and $\hat{r}_2$. This time, the following inequalities hold

$$\frac{\hat{r}_1}{\hat{r}_2} = \frac{\hat{b} + C_1}{(\hat{a} - \hat{q}_1 \hat{b}) + (A_1 - \hat{q}_1 C_1)} < \frac{b}{a - q_1 b} = \frac{r_1}{r_2} < \frac{\hat{b} + D_1}{(\hat{a} - \hat{q}_1 \hat{b}) + (B_1 - \hat{q}_1 D_1)}$$

If we denote $A_2 = C_1$, $B_2 = D_1$, $C_2 = A_1 - \hat{q}_1 C_1$, and $D_2 = B_1 - \hat{q}_1 D_1$, then we can write

$$\frac{\hat{r}_1}{\hat{r}_2} = \frac{\hat{b} + A_2}{(\hat{a} - \hat{q}_1 \hat{b}) + C_2} < \frac{b}{a - q_1 b} = \frac{r_1}{r_2} < \frac{\hat{b} + B_2}{(\hat{a} - \hat{q}_1 \hat{b}) + D_2}$$

and

$$\begin{pmatrix} A_2 & B_2 \\ C_2 & D_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -\hat{q}_1 \end{pmatrix} \begin{pmatrix} A_1 & B_1 \\ C_1 & D_1 \end{pmatrix}$$

We compute again $\hat{q}_2 = \lfloor \hat{r}_1/\hat{r}_2 \rfloor$ and $\hat{q}'_2 = \lfloor (\hat{b} + B_2)/((\hat{a} - \hat{q}_1 \hat{b}) + D_2) \rfloor$, and check whether or not $\hat{q}_2 = \hat{q}'_2$. If the test passes, we know that $\hat{q}_2 = q_2$, and we get

$$\frac{(\hat{a} - \hat{q}_1 \hat{b}) + B_3}{\hat{b} - \hat{q}_2 (\hat{a} - \hat{q}_1 \hat{b}) + D_3} < \frac{a - q_1 b}{b - q_2 (a - q_1 b)} = \frac{r_2}{r_3} < \frac{(\hat{a} - \hat{q}_1 \hat{b}) + A_3}{\hat{b} - \hat{q}_2 (\hat{a} - \hat{q}_1 \hat{b}) + C_3} = \frac{\hat{r}_2}{\hat{r}_3},$$

where

$$\begin{pmatrix} A_3 & B_3 \\ C_3 & D_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -\hat{q}_2 \end{pmatrix} \begin{pmatrix} A_2 & B_2 \\ C_2 & D_2 \end{pmatrix}$$

If the test "$\hat{q}_2 = \hat{q}'_2$" fails, then we compute the quotient q_2 by multi-precision division, $q_2 = \lfloor r_1/r_2 \rfloor$, and re-iterate the above procedure with r_2 and $r_3 = r_1 - q_2 r_2$ as long as both are multi-precision integers.

The following lemma can be easily proved by induction.

Lemma 5.2.1 With the notation above, the following are true, for all $i \geq 0$:

(1) $\binom{r_{i+1}}{r_{i+2}} = \left(\begin{smallmatrix} A_{i+2} & B_{i+2} \\ C_{i+2} & D_{i+2} \end{smallmatrix}\right) \binom{r_i}{r_{i+1}}$, providing that $\hat{q}_j = q_j$ for all $j \leq i + 1$;

(2) At most one of $\hat{r}_{i+1} + C_{i+1}$ and $\hat{r}_{i+1} + D_{i+1}$ can be zero;

(3) $0 \leq \hat{r}_i + A_{i+1} \leq \beta^p$, $0 \leq \hat{r}_i + B_{i+1} < \beta^p$, $0 \leq \hat{r}_{i+1} + C_{i+1} < \beta^p$, and $0 \leq \hat{r}_{i+1} + D_{i+1} \leq \beta^p$;

(4) $B_{i+1} = 0$ if and only if $i = 0$;

(5) $A_{i+1} B_{i+1} \leq 0$ and $C_{i+1} D_{i+1} \leq 0$.

The algorithm **Lehmer1** is a solution to the GCD problem using Lehmer's approach. It is similar to the one in [36]. In this algorithm, X and Y are multi-precision variables, while A, B, C, D, and x are signed single-precision variables (therefore, one bit of each of these variables is needed for sign). $\hat{a}$, $\hat{b}$, q, and q' are single-precision variables.

Computing the quotients q and q' at lines 6 and 7 may result in overflow because $0 \leq \hat{a} + A \leq \beta^p$ and $0 \leq \hat{b} + D \leq \beta^p$ (Lemma 5.2.1(3)). This can be accommodated by reserving two bits more for $\hat{a}$ and $\hat{b}$, one for sign and one for the possible overflow.

Lehmer1(a, b, p)
`input:` $a \geq b > 0$ two multi-precision base β integers, and $p > 0$ such that all $x < \beta^p$ are single-precision integers;
`output:` $gcd(a, b)$;
`begin`
1. `while` $b > \beta^p$ `do`
`begin`
2. `let` h be the smallest integer such that $\hat{a} := \lfloor a/\beta^h \rfloor < \beta^p$;
3. $\hat{b} := \lfloor b/\beta^h \rfloor < \beta^p$;
4. $A := 1;\ B := 0;\ C := 0;\ D := 1;$
5. `while` $\hat{b} + C \neq 0$ `and` $\hat{b} + D \neq 0$ `do`
`begin`
6. $q := \lfloor (\hat{a} + A)/(\hat{b} + C) \rfloor;\ q' := \lfloor (\hat{a} + B)/(\hat{b} + D) \rfloor;$
7. `if` $q \neq q'$ `then break`;
8. $x := A - qC;\ A := C;\ C := x;$
9. $x := B - qD;\ B := D;\ D := x;$
10. $x := \hat{a} - q\hat{b};\ \hat{a} := \hat{b};\ \hat{b} := x;$
`end`;
11. `if` $B = 0$
`then begin`
12. $X := a\ mod\ b;\ a := b;\ b := X;$
`end`
`else begin`
13. $X := Aa + Bb;\ Y := Ca + Db;\ a := X;\ b := Y;$
`end`
`end`
14. $gcd(a, b) :=$**Euclid**(a, b)
`end.`

The loop in line 5 computes $(\hat{r}_{i+1}, \hat{r}_{i+2})$ from $(\hat{r}_i, \hat{r}_{i+1})$, where $i \geq 0$ (using the notation above). The computation halts when one of the two $\hat{q}_i$ and $\hat{q}'_i$ values cannot be computed (because $\hat{r}_{i+1} + C_{i+1} = 0$ or $\hat{r}_{i+1} + D_{i+1} = 0$) or $\hat{q}_i \neq \hat{q}'_i$ (line 7 in the algorithm).

If one of these two cases happens, the loop in line 1 should be re-iterated with new a and b values. These values are obtained as in Lemma 5.2.1(1) if the loop in line 5 has been completely processed at least once (that is, the matrix given by A_2, B_2, C_2, and D_2 has been computed). Otherwise, the new a and b values are computed by multi-precision division (line 12). The distinction between these two cases is made by the test "$B = 0$" which passes only if no iteration of the loop in line 5 could be completed (Lemma 5.2.1(4)).

We also remark that X and Y in line 13 are computed by subtraction (Lemma 5.2.1(5)).

Lehmer's algorithm can be easily extended to get both $gcd(a, b)$ and a linear combination of it with respect to a and b. The algorithm **Lehmer1Ext** is a solution to this problem. It exploits the advantage of computing only the first cosequence.

Lehmer1Ext(a, b, p)
`input:` $a \geq b > 0$ two multi-precision base β integers, and
$p > 0$ such that all $x < \beta^p$ are single-precision;
`output:` $gcd(a, b)$ and $V = (u, v)$ such that $gcd(a, b) = ua + vb$;
`begin`
1. $u_0 := 1$, $u_1 := 0$;
2. $a' := a$, $b' := b$;
3. `while` $b' > \beta^p$ `do`
`begin`
4. `let` h be the smallest integer such that $\hat{a} := \lfloor a'/\beta^h \rfloor < \beta^p$;
5. $\hat{b} := \lfloor b'/\beta^h \rfloor < \beta^p$;
6. $A := 1$; $B := 0$; $C := 0$; $D := 1$;
7. `while` $\hat{b} + C \neq 0$ `and` $\hat{b} + D \neq 0$ `do`
`begin`
8. $q := \lfloor (\hat{a} + A)/(\hat{b} + C) \rfloor$; $q' := \lfloor (\hat{a} + B)/(\hat{b} + D) \rfloor$;
9. `if` $q \neq q'$ `then break`;
10. $x := A - qC$; $A := C$; $C := x$;
11. $x := B - qD$; $B := D$; $D := x$;
12. $x := \hat{a} - q\hat{b}$; $\hat{a} := \hat{b}$; $\hat{b} := x$;
`end`;
13. `if` $B = 0$
`then begin`
14. $X := \lfloor a'/b' \rfloor$; $Y := a' \ mod \ b'$; $a' := b'$; $b' := Y$;
15. $x := u_0 - Xu_1$; $u_0 := u_1$; $u_1 := x$;
`end`
`else begin`
16. $X := Aa' + Bb'$; $Y := Ca' + Db'$; $a' := X$; $b' := Y$;
17. $X := Au_0 + Bu_1$; $Y := Cu_0 + Du_1$; $u_0 := X$; $u_1 := Y$;
`end`
`end`
18. $(gcd(a, b), (u', v')) :=$**EuclidExt**(a', b');
19. $u := u'u_0 + v'u_1$; $v := (gcd(a, b) - ua)/b$
`end.`

Collins' Approach Lehmer's quotient test is based on computing one more quotient (line 7 in **Lehmer1**). In 1980, Collins [14] developed a better test which requires only the computation of the sequences $(\hat{q}_i)$, $(\hat{r}_i)$, and $(\hat{v}_i)$ associated to $\hat{r}_0 = \hat{a}$ and $\hat{r}_1 = \hat{b}$. The test is

– if $(\forall i \leq k)(\hat{r}_{i+1} \geq |\hat{\beta}_{i+1}|$ and $\hat{r}_{i+1} - \hat{r}_i \geq |\hat{v}_{i+1} - \hat{v}_i|)$, then $(\forall i \leq k)(q_i = \hat{q}_i)$.

Collins' test has the advantage that only one quotient has to be computed, and only one of the cosequences.

Jebelean's Approach In 1993, Jebelean [31] refined Collins' quotient test getting an exact quotient test based on the sequences $(\hat{q}_i)$, $(\hat{r}_i)$, $(\hat{u}_i)$, and $(\hat{v}_i)$ associated to $\hat{r}_0 = \hat{a}$ and $\hat{r}_1 = \hat{b}$. The test is

– $q_i = \hat{q}_i$ for all $1 \leq i \leq k$ iff for all $1 \leq i \leq k$ the following are true:

1. $\hat{r}_{i+1} \geq -\hat{u}_{i+1}$ and $\hat{r}_i - \hat{r}_{i+1} \geq \hat{v}_{i+1} - \hat{v}_i$, if i is even, and
2. $\hat{r}_{i+1} \geq -\hat{v}_{i+1}$ and $\hat{r}_i - \hat{r}_{i+1} \geq \hat{u}_{i+1} - \hat{u}_i$, if i is odd

(the proof can be found in [31]).

The algorithm **Lehmer2** below is a version of Lehmer's algorithm based on Jebelean's quotient test. The algorithm is taken from [59].

Lehmer2(a, b, p)
`input:` $a \geq b > 0$ two multi-precision base β integers, and
$p > 0$ such that all $x < \beta^p$ are single-precision;
`output:` $gcd(a, b)$;
`begin`

```
1.    while b ≠ 0 do
         begin
2.          if |a|_β − |b|_β ≤ p/2 then
               begin
3.                h := ⌊a/β^h⌋;
4.                â := ⌊a/β^h⌋;
5.                b̂ := ⌊b/β^h⌋;
6.                (û_0, v̂_0) := (1, 0);
7.                (û_1, v̂_1) := (0, 1);
8.                i := 0;
9.                done:=false;
10.               repeat
11.                  q̂ := ⌊â/b̂⌋;
12.                  (x, y) := (û_0, v̂_0) − q̂(û_1, v̂_1);
13.                  (â, b̂) := (b̂, â − q̂b̂);
14.                  i := i + 1;
15.                  if i is even
16.                     then done := b̂ < −x or â − b̂ < y − v̂_1
17.                     else done := b̂ < −y or â − b̂ < x − û_1;
18.                  (û_0, v̂_0) := (û_1, v̂_1);
19.                  (û_1, v̂_1) := (x, y);
20.               until done;
21.               (a, b) := (û_0 a + v̂_0 b, û_1 a + v̂_1 b);
               end;
22.         r := a mod b;  a := b;  b := r;
         end
23.   gcd(a, b) := a;
end.
```

5.3 The Binary GCD Algorithm

In 1967, Stein [60] discovered a "non-Euclidean" algorithm for computing GCD. It is based on the following easy remarks:

1. If a and b are both even, then $gcd(a, b) = 2 \cdot gcd(a/2, b/2)$;
2. If a is even and b is odd, then $gcd(a, b) = gcd(a/2, b)$;
3. If a and b are both odd, then $gcd(a, b) = gcd(|a - b|/2, b)$;
4. $gcd(a, 0) = |a|$.

The idea is to apply the first identity until one of a and b is odd. Then, identities 2 and 3 are applied, making a and b smaller each time. Since a or b is always odd from this point on, we shall never apply the first identity again. Eventually, one of a and b becomes zero, allowing the application of the last identity.

An explicit description of the algorithm is given below.

Binary(a, b)
`input:` $a \geq b > 0$ integers;
`output:` $gcd(a, b)$;
`begin`
1. x:=1;
2. `while` $(a \ mod \ 2 = 0)$ `and` $(b \ mod \ 2 = 0)$ `do`
   `begin`
3. $a := a/2;\ \ b := b/2;\ \ x := 2x;$
   `end`
4. `while` $a \neq 0$ `do`
   `begin`
5. `while` $(a \ mod \ 2 = 0)$ `do` $a := a/2;$
6. `while` $(b \ mod \ 2 = 0)$ `do` $b := b/2;$
7. $y := |a - b|/2;$
8. `if` $a \geq b$ `then` $a := y$ `else` $b := y;$
   `end`
9. $gcd(a, b) := xb;$

`end.`

The main loop of the algorithm is that at line 4. Since each iteration of this loop reduces the product ab by a factor of 2 or more, the total number of iterations of the main loop is $\mathcal{O}(\log ab)$. As each iteration can be performed in $\mathcal{O}(\log ab)$ time, the total time complexity of the algorithm is $\mathcal{O}((\log ab)^2)$.

The binary GCD algorithm can be extended in order to find a linear combination of GCD, based on the following remarks:

1. let $a \geq b > 0$ be integers;
2. the algorithm **Binary**(a, b) transforms a and b until one of them becomes 0. Then, the other one multiplied by a suitable power of two is $gcd(a, b)$;
3. if we start the algorithm with a linear combination of a and a linear combination of b, usually $(1, 0)$ and $(0, 1)$ respectively, and transform these linear combinations in accordance with the transformations applied to a and b, then we get finally a linear combination of the non-zero factor (which is a linear combination of $gcd(a, b)$).

   More precisely:

(a) start initially with $(1,0)$ and $(0,1)$ as linear combinations for a and b, respectively;

(b) if both a and b are even, then divide them by 2 without changing their linear combination. This is based on the fact that any linear combination of $gcd(a/2, b/2)$ is a linear combination of $gcd(a,b)$, and vice-versa;

(c) assume that a is even and b is odd, and their linear combinations are (u_0, v_0) and (u_1, v_1), respectively. Then,

  i. $(u_0/2, v_0/2)$ is a linear combination of $a/2$, if u_0 and v_0 are even;

  ii. $((u_0+b)/2, (v_0-a)/2)$ is a linear combination of $a/2$, if u_0 or v_0 is odd.

(d) assume that a and b are odd, $a \geq b$, and their linear combinations are (u_0, v_0) and (u_1, v_1), respectively. Then, $(u_0 - u_1, v_0 - v_1)$ is a linear combination of $a - b$.

These remarks lead to the algorithm **BinaryExt** given below.

BinaryExt(a,b)
`input:` $a \geq b > 0$ integers;
`output:` $gcd(a,b)$ and $V = (u,v)$ such that $gcd(a,b) = ua + vb$;
`begin`

```
1.   x:=1;
2.   while (a mod 2 = 0) and (b mod 2 = 0) do
       begin
3.       a := a/2;  b := b/2;  x := 2x;
       end
4.   u0 := 1;  v0 := 0;  u1 := 0;  v1 := 1;
5.   a' := a;  b' := b;
6.   while a' ≠ 0 do
       begin
7.       while (a' mod 2 = 0) do
           begin
8.             a' := a'/2;
9.             if (u0 mod 2 = 0) and (v0 mod 2 = 0)
10.              then begin u0 := u0/2;  v0 := v0/2 end
11.              else begin u0 := (u0 + b)/2;  v0 := (v0 - a)/2 end;
           end
12.      while (b' mod 2 = 0) do
           begin
13.            b' := b'/2;
14.            if (u1 mod 2 = 0) and (v1 mod 2 = 0)
15.              then begin u1 := u1/2;  v1 := v1/2 end
16.              else begin u1 := (u1 + b)/2;  v1 := (v1 - a)/2 end;
           end
17.      if a' ≥ b'
18.        then begin a' := a' - b';  u0 := u0 - u1;  v0 := v0 - v1 end
19.        else begin b' := b' - a';  u1 := u1 - u0;  v1 := v1 - v0 end
       end
20.  gcd(a,b) := xb';  V := (u1, v1)
end.
```

6 Modular Reductions

Arithmetic in $\mathbf{Z}_m$, the set of integers modulo m, is usually called *modular arithmetic*. Basic modular operations (addition, subtraction, multiplication, exponentiation) are obtained from basic operations on integers plus a *modular reduction*. For example,

$$a \oplus b = (a + b) \ mod \ m,$$

for all $a, b \in \mathbf{Z}_m$, where $\oplus$ is the addition in $\mathbf{Z}_m$ and $x \ mod \ m$ is the remainder of the division of x by m (also called the *modular reduction* of x).

Modular addition and subtraction can be easily implemented taking into account the following remarks:

1. $a + b < 2m$,
2. if $a \geq b$, then $0 \leq a - b < m$, and
3. if $a < b$ then $a + m - b < m$,

for all $a, b \in \mathbf{Z}_m$.

Things are more involved in case of modular multiplication or exponentiation, where efficient modular reduction techniques are required.

The most straightforward method for performing modular reduction is to compute the remainder of division by m using a multi-precision division algorithm. This method is commonly referred to as the *classical reduction*. There are also other reduction techniques, like Barrett or Montgomery reduction, which can speed up modular operations in certain circumstancies.

6.1 Barrett Reduction

The modular reduction technique we present in this section has been introduced by Paul Barrett [2] in order to obtain a high speed implementation of the RSA encryption algorithm on an "off-the-shelf" digital signal processing chip.

Let us assume that the representation base is β and

$$m = m_{k-1}\beta^{k-1} + \cdots + m_1\beta + m_0$$

$$a = a_{l-1}\beta^{l-1} + \cdots + a_1\beta + a_0,$$

where $k < l \leq 2k$. Therefore, $\beta^{k-1} \leq m < \beta^k \leq a < \beta^{2k}$.

We can write

$$\left\lfloor \frac{a}{m} \right\rfloor = \left\lfloor \frac{a}{\beta^{k-1}} \cdot \frac{\beta^{2k}}{m} \cdot \frac{1}{\beta^{k+1}} \right\rfloor$$

which shows that quotient $q = \lfloor a/m \rfloor$ can be estimated by computing

$$\hat{q} = \left\lfloor \frac{u}{\beta^{k+1}} \right\rfloor,$$

where

$$u = \mu \left\lfloor \frac{a}{\beta^{k-1}} \right\rfloor = \mu a[l-1 : k-1] \ \text{ and } \ \mu = \left\lfloor \frac{\beta^{2k}}{m} \right\rfloor.$$

The estimation $\hat{q}$ satisfies:

Theorem 6.1.1 Using the notations above, if $k < l \leq 2k$, then $q - 2 \leq \hat{q} \leq q$.

Proof Clearly, $\hat{q} \leq q$. Using $x/y \geq \lfloor x/y \rfloor > x/y - 1$, we obtain

$$\begin{aligned}
\frac{a}{m} &\geq \hat{q} \\
&> \frac{1}{\beta^{k+1}} \left(\frac{a}{\beta^{k-1}} - 1 \right) \left(\frac{\beta^{2k}}{m} - 1 \right) - 1 \\
&= \frac{a}{m} - \frac{a}{\beta^{2k}} - \frac{\beta^{k-1}}{m} + \frac{1}{\beta^{k+1}} - 1 \\
&\geq q - \left(\frac{a}{\beta^{2k}} + \frac{\beta^{k-1}}{m} - \frac{1}{\beta^{k+1}} + 1 \right) \\
&> q - 3,
\end{aligned}$$

which proves the theorem. $\diamond$

The theorem above shows that the error in estimating q by $\hat{q}$ is at most two. It is mentioned in [2] that for about 90% of the values of a and m, the initial value of $\hat{q}$ will be the right one, and that only in 1% of cases $\hat{q}$ will be two in error.

The computation of $\hat{q}$ requires:

- a pre-computation (of μ),
- two divisions by a power of β (which are right-shifts of the base β representation), and
- a multi-precision multiplication of μ and $a[l-1:k-1]$.

Moreover, we remark that there is no need for the whole multiplication of μ and $a[l-1:k-1]$ because we have to divide the result by β^{k+1} and, therefore, we do not need the lower $k+1$ digits of the product. But, we must take into consideration the influence of the carry from position k to position $k+1$. This carry can be accurately estimated by only computing the digits from the position $k-1$. Therefore, we can obtain an approximation of the correct value of $\hat{q}$ by computing

$$\sum_{i+j \geq k-1} \mu_i a_{j+k-1} \beta^{i+j}$$

and then dividing the result by β^{k+1}. This approximation may be at most one in error.

An estimate for $a \ mod \ q$ is then $\hat{r}$ given by $\hat{r} = a - \hat{q}m$. The real remainder $r = a - mq$ satisfies $r < m < \beta^k$. The remainder $\hat{r}$ may be larger than β^k because $\hat{q}$ could be smaller than q. But, if $\beta > 2$, we have $\hat{r} < \beta^{k+1}$. Therefore, we can find $\hat{r}$ by

$$\hat{r} = (a \ mod \ \beta^{k+1} - (\hat{q}m) \ mod \ \beta^{k+1}) \ mod \ \beta^{k+1},$$

which means that the product $\hat{q}m$ can be done partially by computing only the lower $k+1$ digits

$$\sum_{i+j \leq k} \hat{q}_i m_j \beta^{i+j}.$$

As a conclusion, we can obtain $a \ mod \ m$ by subtracting $\hat{q}m \ mod \ \beta^{k+1}$ from $a[k:0]$, and then adjusting the result by at most two subtractions of m.

The algorithm **BarrettRed** takes into account all these remarks.

BarrettRed(a, m)
`input:` a base β l-digit number a and a base β k-digit number m such that $k < l \leq 2k$;
`output:` $r = a \ mod \ m$;
`pre-computation:` $\mu = \lfloor \beta^{2k}/m \rfloor$;
`begin`

1. $u := \sum_{i+j \geq k-1} \mu_i a_{j+k-1} \beta^{i+j}$;
2. $\hat{q} := \lfloor u/\beta^{k+1} \rfloor$;
3. $v := \sum_{i+j \leq k} \hat{q}_i m_j \beta^{i+j}$;
4. $r := a[k:0] - v[k:0]$;
5. `if` $r < 0$ `then` $r := r + \beta^{k+1}$;
6. `while` $r \geq m$ `do` $r := r - m$;

`end.`

Since μ is at most $k+1$ digits and $l \leq 2k$, the number of multiplications in line 1 is at most $\frac{1}{2}k(k+5)+1$. Indeed, for each i, $0 \leq i \leq k-1$, there are $i+2$ possible choices for j because the inequalities $i+j \geq k-1$ and $j+k-1 \leq 2k-1$ must be fulfilled. For $i = k$ there are $k+1$ choices for j (by the same reasoning). Therefore, there are at most

$$\sum_{i=0}^{k-1}(i+2) + (k+1) = \frac{1}{2}k(k+5)+1$$

multiplications in line 1.

Similarly, since $\hat{q}$ and m are k-digit numbers, at most $\frac{1}{2}k(k+3)-1$ multiplications are needed in line 3. Therefore, the total number of multiplications required by the Barrett's algorithm is at most $k(k+4)$.

Barrett's reduction requires pre-computation and, therefore, it may be advantageous when many reductions are performed with the same modulus.

6.2 Montgomery Reduction and Multiplication

Montgomery reduction and multiplication [46] has been introduced to speed up modular multiplication and squaring required during the exponentiation process. Montgomery reduction replaces a division by a modulus m with a multiplication and a division by a special modulus R. The modulus R is chosen such that computations via R are easy. Usually, R is a power of the base β.

Montgomery reduction Let m be a modulus and R such that $R > m$ and $gcd(m, R) = 1$. The *Montgomery reduction* of an integer a w.r.t. m and R is defined as $aR^{-1} \ mod \ m$, where R^{-1} is the inverse of R modulo m (it exists because R is relatively prime to m).

Theorem 6.2.1 Let m and R be positive integers such that $m < R$ and $gcd(m, R) = 1$. Then, for all integers a the following are true:

(1) $a + tm$ is an integer multiple of R,

(2) $(a + tm)/R \equiv aR^{-1} \ mod \ m$,

(3) $(a + tm)/R < 2m$, provided that $a < mR$,

where $t = am' \ mod \ R$, $m' = -m^{-1} \ mod \ R$, and m^{-1} is the inverse of m modulo R.

Proof (1) follows from the remark that $tm \equiv -a \ mod \ R$, which leads to $a + tm = \alpha R$, for some integer α.

(2) It is enough to show that α in the relation above is congruent to aR^{-1} modulo m. Indeed, we have $aR^{-1} + tmR^{-1} = \alpha$, which shows that $aR^{-1} \equiv \alpha \ mod \ m$.

(3) follows from $a < mR$ and $t < R$. $\diamond$

Theorem 6.2.1 shows that, in order to compute the Montgomery reduction of a one can compute the estimate $\hat{a} = (a+tm)/R$ which is at most one in error, provided that $a < mR$. This means that the Montgomery reduction of a reduces to computing $(a+tm)/R$, which means the computation of $m^{-1} \ mod \ R$, of t (by a multiplication), and finally to perform a division by R. These computations can be performed efficiently if we choose $R = \beta^k$, where $\beta^{k-1} \leq m < \beta^k$. Moreover, Dussé and Kaliski [18] remarked that the main idea in Theorem 6.2.1 is to make a a multiple of R by adding multiples of m. Instead of computing all of t at once, one can compute one digit t_i at a time, add $t_i m \beta^i$ to a, and repeat. This change, combined with the particular choice of R, allows to compute $m'_0 = -m_0^{-1} \ mod \ \beta$ instead of m', where m_0 is the least digit of m (providing that $gcd(m_0, \beta) = 1$).

A Montgomery reduction can be implemented according to the algorithm **MonRed**.

MonRed(a, m)
`input:` $m = (m_{k-1} \cdots m_0)_\beta$ and a such that $0 \leq a < mR$
and $gcd(m_0, \beta) = 1$, where $R = \beta^k$;
`output:` $r = aR^{-1} \ mod \ m$;
`pre-computation:` $m'_0 = -m_0^{-1}$ modulo β;
`begin`
1. `for` $i := 0$ to $k - 1$ `do`
   `begin`
2. $t_i := a_i m'_0 \ mod \ \beta$;
3. $a := a + t_i m \beta^i$;
   `end`
4. $a := \lfloor a/\beta^k \rfloor$;
5. `if` $a \geq m$ `then` $a := a - m$;
6. $r := a$

`end.`

As can be seen from the algorithm, the Montgomery reduction can be performed in $k(k + 1)$ multiplications, which is superior to Barrett's algorithm. However, we note that the number of multiplications does not depend by the size of a. This means that the Montgomery reduction is not efficient for small numbers, where the Barrett reduction is better to use.

Montgomery multiplication Let m and R be as in the Montgomery reduction. It is straightforward to show that the set $\{aR \ mod \ m | a < m\}$ is a complete residue system, i.e., it contains all numbers between 0 and $m - 1$. The number $\bar{a} = aR \ mod \ m$ is called the *m-residue* of a.

Thus, there is a one-to-one correspondence between the numbers in the range 0 and $m-1$ and the numbers in the set above, given by $a \mapsto \bar{a} = aR \ mod \ m$.

The *Montgomery product* of two m-residues $\bar{a}$ and $\bar{b}$ is defined as the m-residue

$$\bar{c} = \bar{a}\bar{b}R^{-1} \ mod \ m.$$

It is easy to see that $\bar{c}$ is the m-residue of the product $c = ab \ mod \ m$.

The computation of the Montgomery product is achieved as in the algorithm **MonPro**.

MonPro$(\bar{a}, \bar{b}, m)$
`input:` two m-residues $\bar{a}$ and $\bar{b}$;
`output:` $\bar{c} = \bar{a}\bar{b}R^{-1} \ mod \ m$;
`pre-computation:` $m'_0 = -m_0^{-1}$ modulo β, provided that $gcd(m_0, \beta) = 1$;
`begin`
1. $x := \bar{a}\bar{b}$;
2. $\bar{c}$:=**MonRed**(x, m);

`end.`

Remark that x in the algorithm **MonPro** is not larger than mR and, therefore, **MonRed** can be applied to it.

Now, if we want to compute $ab \ mod \ m$, for some $a, b \in \mathbf{Z}_m$, then we can do it in at least two ways:

- **MonPro**(**MonPro**$(a, b, m), R^2 \ mod \ m)$, or
- **MonRed**(**MonPro**$(\bar{a}, \bar{b}, m), m)$.

Although the cost of computing $R^2 \ mod \ m$ (in the first case) and $\bar{a}$ and $\bar{b}$ (in the second case) can be neglected, both variants are more expensive than the usual multiplication followed by a classical reduction. However, Montgomery multiplication is very effective when several modular multiplications w.r.t. the same modulus are needed. Such is the case when one needs to compute modular exponentiation. Details about this operation will be given in Section 7.

6.3 Comparisons

The execution time for classical reduction, Barrett reduction, and Montgomery reduction depends in a different way on the length of the argument. For the first two of them, the time complexity varies linearly between the maximum value (for an argument twice as long as the modulus) and almost zero (for an argument as long as the modulus). For arguments smaller than the modulus no reduction takes place (the arguments are already reduced). On the other hand, the time complexity for Montgomery reduction is independent of the length of the argument. This shows that both the classical and Barrett reduction are faster than Montgomery reduction below a certain length of the argument.

The table in Figure 7 gives some comparisons between classical reduction using recursive division, Barrett reduction, and Montgomery reduction. In this table, $K(k)$ stands for Karatsuba multiplication complexity of two k-digit integers.

	Barrett	Montgomery	Recursive Division
Restrictions on input	$a < \beta^{2k}$	$a < m\beta^k$	None
Pre-computation	$\lfloor \beta^{2k}/m \rfloor$	$-m_0^{-1} \ mod \ \beta$	Normalization
Post-computation	None	None	Unnormalization
Multiplications	$k(k+4)$	$k(k+1)$	$2K(k) + \mathcal{O}(k \log \ k)$

Figure 7: Three reduction methods

7 Exponentiation

The *exponentiation problem* in a monoid $(M, \cdot)$ is to compute a^n, for some $a \in M$ and integer $n > 0$. When M is a group, we may also raise the problem of computing a^n for negative integers n. The naive method for computing a^n requires $n-1$ multiplications, but as we shall see in this section we can do much better. This is very important because exponentiation is heavily used in many working fields such as primality testing, cryptography, security protocols etc. In such fields, practical implementations depend crucially on the efficiency of exponentiation.

Although the methods we are going to present can be applied to any monoid, we shall deal only with *modular exponentiation*, that is exponentiation in $\mathbf{Z}_m$, where the multiplication is "ordinary multiplication followed by reduction modulo m". Therefore, anything in this section refer to integers $a \in \mathbf{Z}_m$, $n \geq 1$, and $m \geq 2$.

We shall mainly discuss five types of exponentiation techniques:

- *general techniques*, where the base and the exponent are arbitrary;
- *fixed-exponent techniques*, where the exponent is fixed and arbitrary choices of the base are allowed;
- *fixed-base techniques*, where the base is fixed and arbitrary choices of the exponent are allowed;
- *techniques based on modulus particularities*, where special properties of the modulus are exploited;
- *exponent recoding techniques*, where various representations of the exponent are used.

Finally, we focus on *multi-exponentiation techniques*.

In general, there are two ways of reducing the time required by a modular exponentiation. One way is to reduce the number of modular multiplications, and the other is to decrease the time required for modular multiplication. Both of them will be considered in our exposition, but the stress lies on the first one (a preliminary version of this section can be found in [30]).

7.1 General Techniques

Most of the general exponentiation techniques, i.e., techniques which do not exploit any particularity of the exponent or of the base, thus being generally applicable, can be viewed as particular cases or slight variants of the *sliding window method*. This method is based on arbitrary block decompositions of the binary representation of the exponent, $n = [w_{l-1}, \ldots, w_0]_{2^+}$. The blocks w_i are usually called *windows*.

There are several variants of this method, depending on how the windows are built and scanned, or depending on the modular multiplication used. Usually, the windows can have prescribed values or can be built in an adaptive way. They can be scanned from left to right or vice-versa. The modular multiplication can be obtained from usual multiplication combined with modular reduction, or can be a Montgomery-like multiplication.

The left-to-right sliding window technique Any left-to-right sliding window method is based on a decomposition

$$a^n = (\cdots((a^{(w_{l-1})_2})^{2^{|w_{l-2}|}} \cdot a^{(w_{l-2})_2})^{2^{|w_{l-3}|}} \cdots a^{(w_1)_2})^{2^{|w_0|}} \cdot a^{(w_0)_2},$$

where $n = [w_{l-1}, \ldots, w_0]_{2^+}$ is an arbitrary block decomposition of the binary representation of n. We can use a certain strategy of choosing the windows such that the resulted windows values are in a fixed set W. In this case, we obtain the following algorithm.

LRSlidWindExp(a, n, m, W)
`input:` $0 < a < m$, $n \geq 1$, $m \geq 2$, and $W \subset \mathbf{N}$;
`output:` $x = a^n \ mod \ m$;
`begin`
1. `for` each $i \in W$ `do compute` $x_i = a^i \ mod \ m$;
2. `let` $n = (n_{k-1}, \ldots, n_0)_2$;
3. $x := 1$;
4. $i := k - 1$;
5. `while` $i \geq 0$ `do`
   `begin`
6. `find` an appropriate bitstring $n_i \cdots n_j$ such that $(n_i, \ldots, n_j)_2 \in W$;
7. `for` $l := 1$ `to` $i - j + 1$ `do` $x := x \cdot x \ mod \ m$;
8. $x := x \cdot x_{(n_i, \ldots, n_j)_2} \ mod \ m$;
9. $i := j - 1$;
   `end`
`end`.

The computation of $a^i \ mod \ m$, for all $i \in W$, can be efficiently performed by using the technique of addition sequences (see Section 7.2).

We shall discuss now some very important possible choices of W and of the corresponding windows:

- $W = \{0, 1, 3, \ldots, 2^w - 1\}$, for some $w \geq 1$. The parameter w is referred to as the *window size*. This variant has been proposed by Thurber [63] in terms of addition chains (see Section 7.2), using two kinds of windows: *zero windows*, formed by a single 0, and *odd windows*, which begin and end with 1 and have length at most w.

  In this case, the line 1 in algorithm **LRSlidWindExp** should be replaced by

  1.1. $x_0 := 1$;
  1.2. $x_1 := a$;
  1.3. $x_2 := x_1 \cdot x_1 \ mod \ m$;
  1.4. `for` $i := 3$ `to` $2^w - 1$ `step` 2 `do` $x_i = x_{i-2} \cdot x_2 \ mod \ m$;

  and line 6 should be replaced by

  6. `if` $n_i = 0$
     `then` $j := i$
     `else find` the longest bitstring $n_i \cdots n_j$ such that
     $i - j + 1 \leq w$ and $n_j = 1$;

The time complexity of the algorithm obtained in this way is given by:

- 2^{w-1} multiplications in line 1,
- $\lfloor \log n \rfloor$ squarings in line 7, and
- $|\{w_i | w_i \textit{ odd window}\}|$ multiplications in line 8.

(for a detailed analysis the reader is referred to [11, 12]).

- $W = \{0, 1, 3, \ldots, 2^w + f\}$, for some $w \geq 2$, and f odd, $1 \leq f \leq 2^w - 3$. This case has been considered by Möller [45] as an alternative to the sliding window method using only zero and odd windows for limited memory environments. Sometimes, the space available is more than sufficient for the mentioned sliding window method with the window size w but it is not enough for the case of window size $w + 1$. In order to take advantage of a larger window, we may try to apply the above method with the window size $w + 1$ as long as the resulted window values are in the pre-computed range. In this way, the line 1 in algorithm **LRSlidWindExp** should be replaced by

  1.1. $x_0 := 1$;
  1.2. $x_1 := a$;
  1.3. $x_2 := x_1 \cdot x_1 \ mod \ m$;
  1.4. `for` $i := 3$ `to` $2^w + f$ `step` 2 `do` $x_i = x_{i-2} \cdot x_2 \ mod \ m$;

  and line 6 should be replaced by

  6. `if` $n_i = 0$
     `then` $j := i$
     `else find` the longest bitstring $n_i \cdots n_j$ such that
     $i - j + 1 \leq w + 1$ and $n_j = 1$ and $(n_i, \ldots, n_j)_2 \leq 2^w + f$;

  The method obtained in this way is referred to as the *left-to-right sliding window method with fractional windows.*

- $W = \{0, 1, 2, \ldots, 2^w - 1\}$, for some $w \geq 1$. This variant has been considered by Brauer [8] in terms of addition chains, using only windows of length w ($n = [w_{l-1}, \ldots, w_0]_{2^w}$). In this case, the line 1 in algorithm **LRSlidWindExp** should be replaced by

  1.1. $x_0 := 1$;
  1.2. `for` $i := 1$ `to` $2^w - 1$ `do` $x_i = x_{i-1} \cdot a \ mod \ m$;

  and line 6 should be replaced by

  6. $j := i - w + 1$;

  The method obtained in this way is referred to as the *left-to-right* 2^w*-ary method*, or simply as the *left-to-right window method* for exponentiation. The time complexity of the obtained algorithm is given by:

  - $2^w - 1$ multiplications in line 1,

– $\lfloor \log n \rfloor$ squarings in line 7, and
– at most $\lfloor \log n \rfloor / w$ multiplications in line 8.

Thus, the number of multiplications is at most $2^w - 1 + (1 + 1/w)\lfloor \log n \rfloor$.

- $W = \{0, 1\}$. In this case, $n = (w_{l-1}, \ldots, w_0)_2$ and we obtain the so-called *left-to-right binary method* for exponentiation. It requires $\lfloor \log n \rfloor$ squarings and $Hw(n)$ multiplications [5]. The left-to-right binary method is also called the *square-and-multiply method* because it is based on repeated squarings and multiplications.

Adaptive left-to-right sliding window technique builds the windows during the parsing of the binary representation. This can be done in many ways. One of the most interesting way is based on the Lempel-Ziv parsing [73] (abbreviated LZ-parsing), proposed by Yacobi [68]. Recall that by LZ-parsing, a string w is decomposed into blocks $w = w_0 \cdots w_{l-1}$, where:

- w_0 is the first simbol of w;
- w_i is the shortest prefix of w' that is not in the set $\{w_0, \ldots, w_{i-1}\}$, if such prefix exists, and w' otherwise, for all $i \geq 1$, where $w = w_0 \cdots w_{i-1} w'$.

The following algorithm implements this idea.

LZSlidWindExp(a, n, m)
`input:` $0 < a < m$, $n \geq 1$, and $m \geq 2$;
`output:` $x = a^n \ mod \ m$;
`begin`
1. `let` $n = (n_{k-1}, \ldots, n_0)_2$;
2. $x := 1$;
3. $i := k - 1$;
4. $D := \emptyset$;
5. `while` $i \geq 0$ `do`
   `begin`
6. `find` the shortest bitstring $n_i \cdots n_j \notin pr_1(D)$;
7. `if` found() `then`
   `then begin`
8. $current\_window := n_i \cdots n_j$;
9. $current\_value := (x_{(n_i, \ldots, n_{j+1})_2})^2 \cdot a^{n_j} \ mod \ m$;
10. $x := x^{2^{i-j+1}} \cdot current\_value \ mod \ m$;
11. $D := D \cup \{(current\_window, current\_value)\}$;
12. $i := j - 1$;
   `end`
   `else begin`
13. $current\_value := x_{(n_i, \ldots, n_0)_2}$;
14. $x := x^{2^{i+1}} \cdot current\_value \ mod \ m$;
15. $i := -1$;
   `end`
   `end`
`end`.

[5] $Hw(n)$ is the *Hamming weigth* of n, that is the number of bits 1 in the binary representation of n.

The set D in the algorithm **LZSlidWindExp** contains pairs (u, x_u), where $x_u = a^{(u)_2} \ mod \ m$. In line 9 of the algorithm **LZSlidWindExp** we may have $j = i$. In this case, we consider that $x_{(n_i,\ldots,n_{j+1})_2} = 1$. We must also remark that if $j \neq i$ then $x_{(n_i,\ldots,n_{j+1})_2} \in pr_2(D)$. In line 13, we also have that $x_{(n_i,\ldots,n_0)_2} \in pr_2(D)$.

The set D can be efficiently maintained using *LZ-parsing trees*. For l' distinct blocks $w_0, \ldots, w_{l'-1}$ in the LZ parsing, we shall have the trees $B_0, \ldots, B_{l'-1}$, obtained as follows:

- the tree B_0 consists in a root node labeled with 1 and a node labeled with $a^{(w_0)_2}$. The edge between these two nodes is labeled by w_0;
- assume we have constructed $B_0, \ldots, B_{i-1}$, for some $i > 0$. The tree B_i is obtained from B_{i-1} as follows:
  - starting with the root node of the tree B_i, follow the path driven by w_i until a leaf of B_i is reached;
  - add correspondingly a new node labeled by

    $$(reached\_leaf\_label)^2 \cdot a^s \ mod \ m,$$

    where s denotes the last symbol of the block w_i. The edge between these two nodes is labeled by s.

Bocharova and Kudryashov [4] suggested a modified LZ-parsing of the binary representation of the exponent by considering only blocks which begin with a 1 bit. The 0 bits between two consecutive blocks are skipped. In this way, the storage is minimized without affecting the performance. The performance of the above methods strongly depends on the entropy of the source of the exponent.

Lou and Chang [42] proposed an *adaptive left-to-right* 2^w*-ary method*. They suggested that, besides initial values

$$a^1 \ mod \ m, \ a^2 \ mod \ m, \ \ldots, \ a^{2^w-1} \ mod \ m,$$

a number of used partial results be stored and re-used during the computation.

The right-to-left sliding window technique is based on the decomposition

$$a^{[w_{l-1},\ldots,w_0]_{2^+}} = (a^{2^{k_0}})^{(w_0)_2} \cdots (a^{2^{k_{l-1}}})^{(w_{l-1})_2},$$

where $k_0 = 0$ and $k_i = \sum_{j=0}^{i-1} |w_j|$, for all $1 \leq i \leq l-1$, and $n = [w_{l-1}, \ldots, w_0]_{2^+}$ is an arbitrary block decomposition of the binary representation of n.

The last product can be re-arranged on the exponent basis by grouping all the terms with the same exponent:

$$\prod_{i=0}^{l-1} (a^{2^{k_i}})^{(w_i)_2} = \prod_{j \in W} \Big(\prod_{\{i|(w_i)_2=j\}} a^{2^{k_i}} \Big)^j,$$

where W is the set of the predicted windows values. Such grouping was first used by Yao in [69]. All these lead to the following algorithm.

RLSlidWindExp(a, n, m, W)
`input:` $0 < a < m$, $n \geq 1$, $m \geq 2$, and $W \subset \mathbf{N}$;
`output:` $x = a^n \ mod \ m$;
`begin`

1. `let` $n = (n_{k-1}, \ldots, n_0)_2$;
2. $y := a$;
3. `for` every $i \in W$ `do` $x_i := 1$;
4. $i := 0$;
5. `while` $i \leq k-1$ `do`
   `begin`
6. `find` an appropriate bitstring $n_j \cdots n_i$
   such that $(n_j, \ldots, n_i)_2 \in W$;
7. $x_{(n_j,\ldots,n_i)_2} := x_{(n_j,\ldots,n_i)_2} \cdot y \ mod \ m$;
8. $y := y^{2^{j-i+1}} \ mod \ m$;
9. $i := j + 1$;
   `end`
10. $x := \prod_{j \in W} x_j^j \ mod \ m$;

`end`.

The expression $\prod_{j \in W} x_j^j \ mod \ m$ in line 10 can be computed using any multi-exponentiation algoritm (see Section 7.6).

We shall discuss now some very important possible choices of W and of the corresponding windows:

- $W = \{0, 1, 3, \ldots, 2^w - 1\}$, for some $w \geq 1$. We may use only zero windows and odd windows. In this case, line 6 in **RLSlidWindExp** should be replaced by

  6. `if` $n_i = 0$
     `then` $j := i$
     `else`
     `find` the longest bitstring $n_j \cdots n_i$ such that
     $j - i + 1 \leq w$ and $n_j = 1$;

  Since the product $\prod_{j \in W} x_j^j \ mod \ m$ can be expressed as [35, answer to Exercise 4.6.3-9]

  $$(x_{2^w-1})^2 \cdot (x_{2^w-1} \cdot x_{2^w-3})^2 \cdots (x_{2^w-1} \cdots x_3)^2 \cdot (x_{2^w-1} \cdots x_1),$$

  line 10 in **RLSlidWindExp** should be replaced by

  10.1. `for` $j := 2^w - 1$ `downto` 3 `step` 2 `do`
  `begin`
  10.2. $x_1 := x_1 \cdot x_j^2 \ mod \ m$;
  10.3. $x_{j-2} := x_{j-2} \cdot x_j \ mod \ m$;
  `end`
  10.4. $x := x_1$;

  The algorithm obtained in this way requires

  – $\lfloor \log n \rfloor$ squarings in line 8,

– $|\{w_i | w_i \textit{ odd window}\}|$ multiplications in line 7, and
– $2^{w-1} - 1$ squarings and $2^w - 2$ multiplications in line 10.

- $W = \{0, 1, 3, \ldots, 2^w + f\}$, for some $w \geq 2$, and f odd, $1 \leq f \leq 2^w - 3$. In this case, line 6 in **RLSlidWindExp** should be replaced by

```
6. if n_i = 0
       then j := i
       else
           find the longest bitstring n_j ··· n_i such that
           j − i + 1 ≤ w + 1 and n_j = 1 and (n_j, ..., n_i)_2 ≤ 2^w + f;
```

and line 10 of the same algorithm should be replaced by

```
10.1. for j := 2^w + f downto 3 step 2 do
          begin
10.2.         x_1 := x_1 · x_j^2 mod m;
10.3.         x_{j-2} := x_{j-2} · x_j mod m;
          end
10.4. x := x_1;
```

The obtained method is referred to as the *right-to-left sliding window method with fractional windows*.

- $W = \{0, 1, 2, \ldots, 2^w - 1\}$, for some $w \geq 1$. Yao [69] proposed using only windows of length w ($n = [w_{l-1}, \ldots, w_0]_{2^w}$).

  In this case, the line 6 in algorithm **RLSlidWindExp** can be replaced with the following line:

  6. $j := i + w - 1$;

  and the line 10 of the same algorithm, because in this case $\prod_{j \in W} x_j^j \bmod m$ can be expressed as [36, answer to Exercise 4.6.3-9]

$$x_{2^w-1} \cdot (x_{2^w-1} \cdot x_{2^w-2}) \cdots (x_{2^w-1} \cdot x_{2^w-2} \cdots x_1),$$

  can be replaced by the following lines:

```
10.1. x := 1;
10.2. z := 1;
10.3. for j := 2^w − 1 downto 1 do
          begin
10.4.         z := z · x_j mod m;
10.5.         x := x · z mod m;
          end
```

The algorithm obtained in this way requires

– $\lfloor \log n \rfloor$ squarings in line 8,
– at most $\lfloor \log n \rfloor / w$ multiplications in line 7, and

– $2^{w+1} - 2$ multiplications in line 10.

The obtained method is referred to as the *right-to-left* 2^w*-ary method* and also as the *right-to-left window method* for exponentiation.

- $W = \{0, 1\}$. In this case, $n = (w_{l-1}, \ldots, w_0)_2$ and we obtain the *right-to-left binary method* for exponentiation. The right-to-left binary method is also called the *multiply-and-square method* because it is based on repeated multiplications and squarings. The right-to-left binary method also requires $\lfloor \log n \rfloor$ squarings and $Hw(n)$ multiplications. Thus, the binary method requires $2\lfloor \log n \rfloor$ multiplications in the worst case and $\frac{3}{2}\lfloor \log n \rfloor$ multiplications on average. Since $\lfloor \log n \rfloor$ is a lower bound for the number of multiplications required by a single exponentiation with the exponent n in the class of the exponentiation methods based on addition chains (see Section 7.2), the binary method is often good enough.

Sliding window with Montgomery multiplication The Montgomery multiplication is very efficient when several modular multiplications with respect to the same modulus are needed (see sections 6.2 and 6.3). Therefore, it is a good idea to use Montgomery multiplication for exponentiation. We shall present here only the algorithm corresponding to the left-to-right sliding window method with zero and odd windows using Montgomery multiplication.

LRMonSlidWindExp(a, n, m, w)
`input:` $0 < a < m$, $n \geq 1$, m and R like in Montgomery reduction, and $w \geq 1$;
`output:` $x = a^n \ mod \ m$;
`begin`

1. $x_1 :=$**MonPro**$(a, R^2 \ mod \ m)$;
2. $x_2 :=$**MonPro**(x_1, x_1);
3. `for` $i := 3$ `to` $2^w - 1$ `step` 2 `do` $x_i =$**MonPro**(x_{i-2}, x_2);
4. `let` $n = (n_{k-1}, \ldots, n_0)_2$;
5. $x := R \ mod \ m$;
6. $i := k - 1$;
7. `while` $i \geq 0$ `do`
8. `if` $n_i = 0$
   `then begin`
9. $x :=$**MonPro**(x, x);
10. $i := i - 1$;
    `end`
    `else begin`
11. `find` the longest bitstring $n_i \cdots n_j$ such that $i - j + 1 \leq w$ and $n_j = 1$;
12. `for` l:=1 `to` $i - j + 1$ `do` $x :=$**MonPro**(x, x);
13. $x :=$**MonPro**$(x, x_{(n_i, \ldots, n_j)_2})$;
14. $i := j - 1$;
    `end`
15. $x :=$**MonPro**$(x, 1)$;

`end`.

7.2 Fixed-Exponent Techniques

The problem of finding the smallest number of multiplications necessary to compute $x = a^n \ mod \ m$ is very similar to the problem of finding one of the shortest addition chains for n.

An *addition chain* of length t for a positive integer n is a sequence of integers

$$e_0 < \cdots < e_t$$

such that $e_0 = 1$, $e_t = n$, and for all $1 \leq i \leq t$ there are $0 \leq j, \ h \leq i-1$ such that $e_i = e_j + e_h$.

If $e_0 < \cdots < e_t$ is an addition chain for n, then $a^n \ mod \ m$ can be computed as follows:

$$\begin{aligned} x_0 &= a^{e_0} \ mod \ m = a \ mod \ m \\ x_1 &= a^{e_1} \ mod \ m \\ &\cdots \\ x_{t-1} &= a^{e_{t-1}} \ mod \ m \\ x_t &= a^{e_t} \ mod \ m \ = \ a^n \ mod \ m \end{aligned}$$

where, for all $1 \leq i \leq t$, $x_i = x_j \cdot x_h \ mod \ m$ for some $0 \leq j, \ h \leq i-1$. As we can see, the required number of multiplications is exactly the length of the addition chain for n. Thus, the shorter the addition chain is, the shorter the execution time for modular exponentiation is. A lower bound for the length of the addition chains for a given positive integer n is provided by the next theorem.

Theorem 7.2.1 Let n be a positive integer and $e_0 < \cdots < e_t$ an addition chain for n. Then $\lfloor \log n \rfloor \leq t$.

Proof Using finitary induction one can easily prove that $e_i \leq 2^i$, for all $0 \leq i \leq t$.

For $i = t$ we have that $n = e_t \leq 2^t$ and, thus, $log \, n \leq t$, which implies $\lfloor \log n \rfloor \leq t$. $\diamond$

Unfortunately, even if there is no proof of the fact that the problem of finding one of the shortest addition chains is **NP**-complete, this problem is considered as very hard. But one can remark that minimizing the exponentiation time is not exactly the same problem as minimizing the addition-chain length, for several reasons:

- the multiplication time is not generally constant. For example, the squaring can be performed faster, or, if an operand is in a given fixed set, pre-computation can be used;
- the time of finding an addition chain should be added to the time spent for multiplications, and must be minimized accordingly.

In the case of a fixed exponent (that is, an exponent which is going to be used for many exponentiations), we may spend more time for finding a good addition chain.

The sliding window exponentiation methods can be viewed as exponentiations based on particular addition chains. For example, the left-to-right sliding window method using only zero and odd windows, with the window size w and the exponent $n = [w_{l-1}, \ldots, w_0]_{2^+}$, is based on the chain obtained by merging the next two sequences:

$$1,\ 2,\ 3,\ 5 \ldots,\ 2^w - 1$$

and

$$2^1 \cdot (w_{l-1})_2, \ldots, 2^{|w_{l-2}|} \cdot (w_{l-1})_2, 2^{|w_{l-2}|} \cdot (w_{l-1})_2 + (w_{l-2})_2,$$
$$2^1 \cdot (2^{|w_{l-2}|} \cdot (w_{l-1})_2 + (w_{l-2})_2), \ldots, 2^{|w_{l-3}|} \cdot (2^{|w_{l-2}|} \cdot (w_{l-1})_2 + (w_{l-2})_2), \ldots,\ n$$

and removing the duplicates.

This type of addition chain has an interesting property:

$$e_i = e_{i-1} + e_j,$$

for all $1 \leq i \leq t$, where either $j = i - 1$ or $0 \leq j \leq 2^{w-1}$.

For such addition chains, called in [35] *star chains*, modular multiplication and squaring can be done in a very efficient way [28]. Let us discuss them.

The main idea for modular multiplications is to use a table of pre-computed values because one of the operands is member of a fixed set. Assume that we have to compute $y \cdot z \ mod \ m$, where $y, z < m$, and z is a member of the set $\{x, x^3, \ldots, x^{2^w-1}\}$, for some $w \geq 1$. The case $w = 1$ was first studied by Kawamura et. al. [34].

If $y = (y_{k-1}, \ldots, y_0)_\beta$ and $z = x^{2j+1} \ mod \ m$, for some $0 \leq j \leq 2^{w-1} - 1$, then $y \cdot z \ mod \ m$ can be expressed as

$$y \cdot z \ mod \ m = \sum_{i=0}^{k-1} y_i \cdot (\beta^i \cdot x^{2j+1} \ mod \ m) \ mod \ m.$$

If we pre-compute and store the values

$$T(i, j) = \beta^i \cdot x^{2j+1} \ mod \ m,$$

for all $0 \leq i \leq k - 1$ and $0 \leq j \leq 2^{w-1} - 1$, then we obtain

$$y \cdot z \ mod \ m = \sum_{i=0}^{k-1} y_i \cdot T(i, j) \ mod \ m.$$

The pre-computed values can be obtained in the following way:

1. $T(0, 0) = x \ mod \ m$;
2. $T(0, j) = T(0, j - 1) \cdot x_2 \ mod \ m$, for all $1 \leq j \leq 2^{w-1} - 1$, where $x_2 = (T(0, 0))^2 \ mod \ m$.

   Thus, $T(0, j) = x^{2j+1} \ mod \ m$, for every $0 \leq j \leq 2^{w-1} - 1$;
3. $T(i, j) = T(i - 1, j) \cdot \beta \ mod \ m$, for all $1 \leq i \leq k - 1$ and $0 \leq j \leq 2^{w-1} - 1$.

Let us discuss now modular squaring. Assume that we have to compute $y^2 \ mod \ m$, for some $y < m$ and $\beta^{k-1} \leq m < \beta^k$.

Denote by $y^{(l)}$ the number obtained by shifting the integer y with l digits to the right. If $y = (y_{k-1}, \ldots, y_0)_\beta$, then $y^{(l)} = \sum_{i=0}^{k-l-1} y_{i+l} \beta^i$.

For $l = k/2$ we obtain that $y^{(l)}$ is a $k/2$-digit integer and thus, $(y^{(k/2)})^2 \ mod \ m$ can be efficiently computed because $(y^{(k/2)})^2$ has at most k digits. Moreover, if m is normalized, then the result of the modular reduction of $(y^{(k/2)})^2$ can be obtained using at most one subtraction, because $(y^{(k/2)})^2 < 2m$. We start with the value $(y^{(k/2)})^2 \ mod \ m$ and then we shall use the recurrence relation

$$(y^{(l-1)})^2 \ mod \ m = (((y^{(l)})^2 \ mod \ m) \cdot \beta^2 + 2 \cdot y^{(l)} \cdot \beta \cdot y_{l-1} + y_{l-1}^2) \ mod \ m$$

(because $y^{(l-1)} = y^{(l)} \cdot \beta + y_{l-1}$), for every $k/2 \geq l \geq 1$. Finally, we obtain $(y^{(0)})^2 \ mod \ m = y^2 \ mod \ m$.

In [63], Thurber suggested that the initial sequence $1 < 2 < 3 < 5 < \cdots < 2^w - 1$ can be replaced by a short *addition sequence* for $(w_0)_2, \ldots, (w_{l-1})_2$, i.e., a short addition chain for $max(\{(w_0)_2, \ldots, (w_{l-1})_2\})$ that contains the windows values $(w_0)_2, \ldots, (w_{l-1})_2$.

In general, short addition sequences for a sequence $x_0, \ldots, x_{l-1}$ can be used to efficiently compute a sequence of powers $a^{x_0}, \ldots, a^{x_{l-1}}$, for a given positive integer a. The problem of finding one of the shortest addition sequence for a given sequence was shown to be **NP**-complete in [17]. A simple algorithm for generating an addition sequence for a sequence $x_0, \ldots, x_{l-1}$ is presented next. It is from [5].

AddSeq$(x_0, \ldots, x_{l-1})$
`input:` $x_0, \ldots, x_{l-1}$;
`output:` *seq*, an addition sequence for $x_0, \ldots, x_{l-1}$;
`begin`
1. $protoseq := \{x_0, \ldots, x_{l-1}\} \cup \{1, 2\}$;
2. $seq := \{1, 2\}$;
3. `while` $max(protoseq) > 2$ `do`
   `begin`
4. $f := max(protoseq)$;
5. $seq := seq \cup \{f\}$;
6. $protoseq := protoseq \cup newnumbers(protoseq)$;
7. $protoseq := protoseq \setminus \{f\}$;
   `end`

`end.`

The simplest variant of the function *newnumbers* is the following:

Function newnumbers$(protoseq)$
`input:`a set *protoseq*;
`output:`a set *rez* derived from the set *protoseq*;
`begin`
1. $f := max(protoseq)$;
2. $f_1 := max(protoseq \setminus \{f\})$;
3. $rez := \{f - f_1\}$;

`end.`

Bos and Coster [6] suggested using larger windows in the sliding window method. They give a heuristic algorithm for finding good addition sequences for $x_0, \ldots, x_{l-1}$, by using a more elaborated variant of the function *newnumbers*, based on 4 rules (the input set is *protoseq* and $f = max(protoseq)$):

- **Approximation** - in case there are a and b in the set *protoseq*, $a < b$, such that $f - (a + b)$ is small, insert $f - b$;
- **Division** - in case that f is divisible with a small prime $p \in \{3, 5, 9, 17\}$, and $e_0 < \cdots < e_t$ is an addition chain for p, insert f/p, $2f/p$, $e_2 f/p$, ..., $e_{t-1} f/p$;
- **Halving** - in case there exists a relatively small number a in the set *protoseq*, insert $f - a$, $(f - a)/2$, $(f - a)/4$, ..., to a certain point;
- **Lucas** - in case there is $a \in protoseq$ such that $a = u_i$ and $f = u_j$, insert $u_{i+1}, \ldots, u_{j-1}$, where the elements of Lucas' sequence $(u_i)_{i \geq 0}$ satisfy the recurrence relation $u_0 = 1$, $u_1 = 3$, and $u_{i+2} = u_{i+1} + u_i$, for all $i \geq 0$.

In the case of a fixed exponent, we can spend more time choosing the windows and finding an optimal addition sequence for the resulted windows values.

7.3 Fixed-Base Techniques

In some cryptographic systems, a fixed element a is repeatedly raised to many different powers. In such cases, pre-computing some of the powers of a may be an option to speed up the exponentiation.

Assume we pre-compute and store $a^{\alpha_i} \ mod \ m$ for some positive integers $\alpha_0, \ldots, \alpha_{k-1}$. We write $x_{\alpha_i} = a^{\alpha_i} \ mod \ m$. If we can decompose the exponent n as $n = \sum_{i=0}^{k-1} n_i \alpha_i$, then

$$a^n \ mod \ m = \prod_{i=0}^{k-1} x_{\alpha_i}^{n_i} \ mod \ m,$$

which can be computed using any multi-exponentiation algorithm (see Section 7.6).

BGMW method In case $0 \leq n_i \leq h$ for some h and all $0 \leq i \leq k - 1$, Brickell, Gordon, McCurly, and Wilson [9] re-iterated Yao's idea [69] (see also Section 7.1) and expressed

$$\prod_{i=0}^{k-1} x_{\alpha_i}^{n_i} \ mod \ m = \prod_{d=1}^{h} y_d^d \ mod \ m,$$

where

$$y_d = \prod_{\{i | n_i = d\}} x_{\alpha_i} \ mod \ m.$$

They also re-used a clever method for computing such products from [36, answer to Exercise 4.6.3-9] as

$$\prod_{d=1}^{h} y_d^d = y_h \cdot (y_h y_{h-1}) \cdots (y_h y_{h-1} \cdots y_1)$$

which leads to the following algorithm.

BGMWExp(a, n, m)
`input:` $0 < a < m$, $n \geq 1$, $m \geq 2$;
`output:` $x = a^n \ mod \ m$;
`pre-computation:` $x_{\alpha_i} = a^{\alpha_i} \ mod \ m$, for all $0 \leq i \leq k-1$;
`begin`
1. `let` $n = \sum_{i=0}^{k-1} n_i \alpha_i$, where $0 \leq n_i \leq h$;
2. $x := 1$;
3. $y := 1$;
4. `for` $d := h$ `downto` 1 `do`
   `begin`
5. `for` each i such that $n_i = d$ `do` $y := y \cdot x_{\alpha_i} \ mod \ m$;
6. $x := x \cdot y \ mod \ m$;
   `end`

`end.`

A possible choice for the α_i values is $\alpha_i = \beta^i$, for all $0 \leq i \leq k-1$, where $\beta = 2^w$ and $w \geq 1$. Then $h = \beta - 1$ and $n = (n_{k-1}, \ldots, n_0)_\beta$. In this case, if we represent the exponent n in base β using k digits, we may compute $a^n \ mod \ m$ using at most $k + \beta - 3$ multiplications (we did not count the multiplications with one operand 1). Storage is required for $a^{\alpha_i} \ mod \ m$, $0 \leq i \leq k-1$.

De Rooij method De Rooij [54] found an efficient algorithm for computing the product $\prod_{i=0}^{k-1} x_{\alpha_i}^{n_i} \ mod \ m$ when n_i are relatively small, for all $0 \leq i \leq k-1$. The algorithm recursively uses the fact that

$$x^n \cdot y^m = (x \cdot y^{m \ div \ n})^n \cdot y^{m \ mod \ n}.$$

In the next algorithm, i_{max} and i_{next} denote the index of the greatest element in the current list of exponents $(r_0, \ldots, r_{l-1})$ and, respectively, the index of the greatest element in the list $(r_0, \ldots, r_{i_{max}-1}, r_{i_{max}+1}, \ldots, r_{l-1})$.

DeRooijExp(a, n, m)
`input:` $0 < a < m$, $n \geq 1$, $m \geq 2$;
`output:` $x = a^n \ mod \ m$;
`pre-computation:` $x_{\alpha_i} = a^{\alpha_i} \ mod \ m$, for all $0 \leq i \leq k-1$;
`begin`
1. `let` $n = \sum_{i=0}^{k-1} n_i \alpha_i$;
2. `for` $i := 0$ `to` $k-1$ `do`
   `begin`
3. $y_i := x_{\alpha_i}$;
4. $r_i := n_i$;
   `end`
5. `find` (i_{max}, i_{next});
6. `while` $r_{i_{next}} > 0$ `do`
   `begin`
7. $q := r_{i_{max}} \ div \ r_{i_{next}}$;
8. $r_{i_{max}} := r_{i_{max}} \ mod \ r_{i_{next}}$;
9. $y_{i_{next}} := y_{i_{next}} \cdot y_{i_{max}}^q \ mod \ m$;
10. `find` (i_{max}, i_{next});
    `end`

11. $x := y_{i_{max}}^{r_{i_{max}}} \ mod\ m$;
end.

The values $y_{i_{max}}^{q} \ mod\ m$ (line 9) and $y_{i_{max}}^{r_{i_{max}}} \ mod\ m$ (line 11) can be computed using, for example, the binary method. In practice, it turns out that the variable q has almost always the value 1.

De Rooij method, often referred to as the *euclidean method* for exponentiation, is slightly slower than the BGMW method but requires much less storage.

Lim-Lee Method In [48], and later [49, 50], Pippenger studied the evaluation of sequences of products of exponentiations (see also [3] for an excellent presentation of Pippenger's work). He related the problem of efficient computing such expressions to a graph problem: finding the minimum number of edges in graphs with prescribed paths. Pippenger remarked that, if $n = (n_{h-1}, \ldots, n_0)_\beta$ and $\beta = 2^\alpha$, where $\alpha \geq 1$, then n can be expressed as

$$\begin{aligned} n &= \textstyle\sum_{i=0}^{h-1} n_i \cdot 2^{\alpha i} \\ &= \textstyle\sum_{i=0}^{h-1} (\sum_{j=0}^{\alpha-1} n_{i,j} \cdot 2^j) \cdot 2^{\alpha i} \\ &= \textstyle\sum_{j=0}^{\alpha-1} (\sum_{i=0}^{h-1} n_{i,j} \cdot 2^{\alpha i}) \cdot 2^j \end{aligned}$$

where $n_i = [n_{i,\alpha-1}, \ldots, n_{i,0}]_2$, for all $0 \leq i \leq h-1$.

Thus, a^n can be computed as

$$\begin{aligned} a^n &= \textstyle\prod_{j=0}^{\alpha-1} (\prod_{i=0}^{h-1} (a^{2^{\alpha i}})^{n_{i,j}})^{2^j} \\ &= \textstyle\prod_{j=0}^{\alpha-1} y_j^{2^j} \end{aligned}$$

where $y_j = \prod_{i=0}^{h-1} (a^{2^{\alpha i}})^{n_{i,j}}$, for all $0 \leq j \leq \alpha - 1$.

The expression $\prod_{j=0}^{\alpha-1} y_j^{2^j}$ can be efficiently evaluated using the repeated square-and-multiply technique. The evaluation of the products $\prod_{i=0}^{h-1} (a^{2^{\alpha i}})^{n_{i,j}}$, is equivalent to the problem of evaluating $\prod_{i=0}^{h-1} x_i^{n_{i,j}}$, for all $0 \leq i \leq h-1$ and $0 \leq j \leq \alpha - 1$, where $x_i = a^{2^{\alpha i}}$. Pippenger focused on the problem of evaluating *multiple-products*, i.e., sequences of products of exponentiations in case all the involved exponents are in the set $\{0, 1\}$.

As Bernstein pointed out in [3], Pippenger exponentiation algorithm was rediscovered by Lim and Lee in [41] for the case of fixed base. It is called now the *Lim-Lee method* or the *comb method*.

Lim and Lee divide the binary representation of the k-bit exponent n into h blocks w_i, for all $0 \leq i \leq h-1$, of size $\alpha = \lceil \frac{k}{h} \rceil$ and then each w_i is subdivided into v blocks $w_{i,j}$, for all $0 \leq j \leq v-1$, of size $\delta = \lceil \frac{\alpha}{v} \rceil$. This can be easily done by first padding the binary representation of the exponent on the left with $(h \cdot v \cdot \delta - k)$ zeros.

We can write:

$$n = [w_{h-1}, \ldots, w_0]_{2^\alpha} = \sum_{i=0}^{h-1} (w_i)_2 2^{i\alpha}$$

$$(w_i)_2 = [w_{i,v-1}, \ldots, w_{i,0}]_{2^\delta} = \sum_{j=0}^{v-1} (w_{i,j})_2 2^{j\delta}.$$

Let $x_0 = a$ and $x_i = x_{i-1}^{2^\alpha} = a^{2^{i\alpha}}$, for all $1 \le i \le h-1$. Then, using the relations above, we can express a^n as:

$$a^n = \prod_{i=0}^{h-1} a^{2^{i\alpha}(w_i)_2} = \prod_{i=0}^{h-1} x_i^{(w_i)_2} = \prod_{i=0}^{h-1}\prod_{j=0}^{v-1} x_i^{2^{j\delta}(w_{i,j})_2} = \prod_{j=0}^{v-1}(\prod_{i=0}^{h-1} x_i^{(w_{i,j})_2})^{2^{j\delta}}.$$

If we let $[e_{i,\alpha-1}, \ldots, e_{i,0}]_2$ be the binary representation of $(w_i)_2$, for all $0 \le i \le h-1$, then $(w_{i,j})_2$ can be binary represented as $[e_{i,j\delta+\delta-1}, \ldots, e_{i,j\delta}]_2$, for all $0 \le j \le v-1$.

Then, a^n can be rewritten as follows:

$$\begin{aligned} a^n &= \textstyle\prod_{j=0}^{v-1}(\prod_{i=0}^{h-1}\prod_{l=0}^{\delta-1} x_i^{2^l e_{i,j\delta+l}})^{2^{j\delta}} \\ &= \textstyle\prod_{j=0}^{v-1}(\prod_{l=0}^{\delta-1}(\prod_{i=0}^{h-1} x_i^{e_{i,j\delta+l}})^{2^l})^{2^{j\delta}} \\ &= \textstyle\prod_{j=0}^{v-1}\prod_{l=0}^{\delta-1}((\prod_{i=0}^{h-1} x_i^{e_{i,j\delta+l}})^{2^{j\delta}})^{2^l} \end{aligned}$$

Assume now that the following values are pre-computed and stored:

$$X[0][i] = x_{h-1}^{e_{h-1}} x_{h-2}^{e_{h-2}} \cdots x_1^{e_1} x_0^{e_0},$$

$$X[j][i] = (X[j-1][i])^{2^\delta} = (X[0][i])^{2^{j\delta}},$$

for all $1 \le i \le 2^h - 1$, $i = [e_{h-1}, \ldots, e_0]_2$, and $0 \le j \le v-1$.

Therefore, we can write:

$$a^n = \prod_{j=0}^{v-1}\prod_{l=0}^{\delta-1} X[j][I_{j,l}]^{2^l} = \prod_{l=0}^{\delta-1}(\prod_{j=0}^{v-1} X[j][I_{j,l}])^{2^l},$$

where $I_{j,l} = (e_{h-1,j\delta+l}, \ldots, e_{0,j\delta+l})_2$.

The last expression can be efficiently evaluated by the square-and-multiply method. We obtain the following algorithm.

LimLeeExp(a, n, m, h, v)
`input:` $0 < a < m$, $n \ge 1$ with $|n|_2 = k$, $m \ge 2$, $h, v \ge 1$;
`output:` $x = a^n \bmod m$;
`pre-computation:` $X[0][i] = x_{h-1}^{e_{h-1}} x_{h-2}^{e_{h-2}} \cdots x_1^{e_1} x_0^{e_0} \bmod m$
$X[j][i] = (X[j-1][i])^{2^\delta} \bmod m = (X[0][i])^{2^{j\delta}} \bmod m$,
where $0 \le j \le v-1$, $i = [e_{h-1}, \ldots, e_0]_2$,
$x_l = x^{2^{l\alpha}} \bmod m$, $0 \le l \le h-1$, $\alpha = \lceil \frac{k}{h} \rceil$, $\delta = \lceil \frac{\alpha}{v} \rceil$;
`begin`
1. `find` the binary representation of the exponent n;
2. `partition` it into $h \cdot v$ blocks of size δ;
3. `arrange` these $h \cdot v$ blocks in a $h \times v$ rectangular shape as in Figure 8;
4. $x := 1$;
5. `for` $l := \delta - 1$ `downto` 0 `do`
`begin`
6. $x := x \cdot x \bmod m$;
7. `for` $j := v - 1$ `downto` 0 `do` $x := x \cdot X[j][I_{j,l}]) \bmod m$;
`end`
`end`.

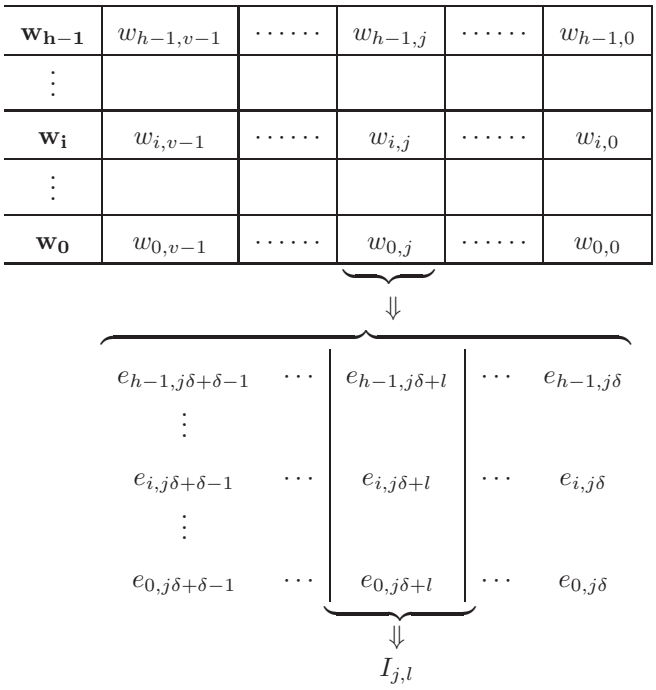


Figure 8: The exponent processing in the Lim-Lee method

In line 7, $I_{j,l}$ represents the binary value of the l^{th} bit column of the j^{th} column of the mentioned table (the numbering of the columns is from right to left and begins with 0).

The algorithm **LimLeeExp** requires

- δ squarings in line 6, and
- $\delta \cdot v$ multiplications in line 7.

Thus, it requires $\alpha + \delta$ multiplications, at the cost of $v \cdot (2^h - 1)$ pre-computed and stored values.

7.4 Techniques Using Modulus Particularities

Suppose we have to compute $a^n \ mod \ m$ and we know a factorization of m, $m = m_1 \cdots m_l$, where $m_1, \ldots, m_l$ are distinct primes of about the same size k. Such cases appear in the basic RSA decryption for $l = 2$ [53], or in *multi-prime* RSA for $l = 3$ [15].

Since

$$(a^n \ mod \ m) \ mod \ m_i = (a \ mod \ m_i)^{n \ mod \ (m_i - 1)} \ mod \ m_i$$

for all $1 \leq i \leq l$, $a^n \ mod \ m$ can be computed using the Chinese Remainder

Theorem as the unique solution modulo $m_1 \cdots m_l$ of the system:

$$\begin{cases} x & \equiv & x_1 \ mod \ m_1 \\ & \cdots & \\ x & \equiv & x_l \ mod \ m_l \end{cases}$$

where $x_i = (a \ mod \ m_i)^{n \ mod \ (m_i - 1)} \ mod \ m_i$, for all $1 \leq i \leq l$.

An efficient algorithm for finding the solution of the system above was proposed by Garner in [21]. The first who used the Chinese Remainder Theorem for exponentiation were Quisquater and Couvreur in [52].

If we take in consideration only the time required for the evaluation of the l modular exponentiations with operands of size k (we assume that the other steps are negligible), we may compute $a^n \ mod \ m$ using, on average, $\frac{3}{2}kl$ modular multiplications with operands of size k.

We shall present next an exponentiation algorithm that can be used for the RSA decryption, and, thus, we shall also consider that the exponent is fixed.

Mod2PrimeExp(a, n, m, m_1, m_2)
`input:` $0 < a < m$, $n \geq 1$, $m = m_1 \cdot m_2$, where m_1 and m_2 are distinct primes;
`output:` $x = a^n \ mod \ m$;
`pre-computation:` $n_1 = n \ mod \ (m_1 - 1)$, $n_2 = n \ mod \ (m_2 - 1)$, $m_1^{-1} \ mod \ m_2$;
`begin`
1. $x_1 = (a \ mod \ m_1)^{n_1} \ mod \ m_1$;
2. $x_2 = (a \ mod \ m_2)^{n_2} \ mod \ m_2$;
3. $x := x_1 + m_1((x_2 - x_1)(m_1^{-1} \ mod \ m_2) \ mod \ m_2)$;

`end`.

The exponentiations in lines 1 and 2 can be performed using, for example, the sliding window method.

7.5 Exponent Recoding

In some cases, a different representation of the exponent may be convenient. The modification of the binary representation of the exponent into another representation is referred to as the *exponent recoding*. The number of non-zero terms in some representation is called the *weight* of the representation. Low-weighted short representations can be combined with other exponentiation methods and give good results.

In what follows we shall present two exponent recoding techniques.

Signed Binary Representation Suppose that the exponent n can be expressed as $\sum_{i=0}^{l-1} \omega_i 2^i$, where $\omega_i \in \{-1, 0, 1\}$. In this case we write

$$n = (\omega_{l-1}, \ldots, \omega_0)_{S(2)},$$

which is called a *signed binary representation* of n.

The next algorithm produces a *modified non-adjacent* [7] signed binary representation of n.

Sigbit_rec(**n**)
`input:` n;
`output:` a modified non-adjacent signed-bit representation of n;
`begin`

1. `let` $n = (n_{k-1}, \ldots, n_0)_2$;
   `repeat`
2. `scan` the current representation from right to left
3. `replace` every block of the form $0\underbrace{1\cdots1}_{i\geq 2}$
   by the block $1\underbrace{0\cdots0}_{i-1}$-1;
   `until` there is no block of the mentioned form;
4. `replace` a possible maximal prefix block of the form $\underbrace{1\cdots1}_{i\geq 3}$
   by the block $1\underbrace{0\cdots0}_{i-1}$-1;

`end`.

In line 4, we replace the prefix $\underbrace{1\cdots1}_{i}$ if and only if $i \geq 3$, because for $i = 2$, the replacement of 110 by 10-10 does not decrease the weight and it also increases the length.

A positive integer n can have more signed binary representations, but it has a unique modified non-adjacent representation. In [7] it is shown that, on average, the weight of the obtained representation is a third of the length of the binary representation of the exponent.

Such representations of the exponent can be used in an obvious manner for exponentiation. We shall present only the algorithm corresponding to the left-to-right variant of the signed binary method for exponentiation.

LRSigBinExp(a, n, m)
`input:` $0 < a < m$, $n \geq 1$, and $m \geq 2$ such that $(a, m) = 1$;
`output:` $x = a^n \ mod \ m$;
`begin`

1. `let` $n = (\omega_{l-1}, \ldots, \omega_0)_{S(2)}$;
2. $x_1 := a$;
3. $x_{-1} := a^{-1} \ mod \ m$;
4. $x := 1$;
5. `for` $i := l - 1$ `downto` 0 `do`
   `begin`
6. $x := x \cdot x \ mod \ m$;
7. `if` $\omega_i <> 0$ `then` $x := x \cdot x_{\omega_i} \ mod \ m$;
   `end`

`end`.

Signed binary representations can be also combined with the 2^w-ary method (see, for example, [19, 20]) or with the sliding window method (see, for example, [37]).

In case of the signed binary representation, unless the value $a^{-1} \ mod \ m$ can be easily computed (or pre-computed), the exponentiation methods based on such representations are worse than the ordinary ones. Nevertheless, the signed

representations can be used in groups where the elements are easily invertible, e.g., the group of the points on an elliptic curve over a finite field.

String-Replacement Representation This method has been introduced by Gollmann, Han, and Mitchell [22]. Suppose that we want to represent a positive number n as $\sum_{i=0}^{l-1} \omega_i \cdot 2^i$ with $\omega_0, \ldots, \omega_{l-1} \in \{0, 1, 3, \ldots, 2^w - 1\}$, $w \geq 2$. This can be easily done by replacing repeatedly every block of i consecutive 1 bits by the string $0 \cdots 0(2^i - 1)$ of length i, for all $2 \leq i \leq w$. In this case, we write

$$n = (\omega_{l-1}, \ldots, \omega_0)_{SR(w)}$$

and call it a *w-ary string replacement representation of n*.

The next algorithm produces such a representation, starting with the binary representation of n.

StrReplRec(n)
`input:` n and $w \geq 2$;
`output:` a w-ary string replacement representation of n;
`begin`
1. `let` $n = (n_{k-1}, \ldots, n_0)_2$;
2. `for` $i := w$ `downto` 2 `do`
3. `scan` the current representation from left to right and replace every block of i consecutive 1 bits by the string $\underbrace{0 \cdots 0}_{i-1}(2^i - 1)$;

`end`.

A positive integer n can have more w-ary string replacement representations. In [22] it is shown that, on average, the weight of the obtained representation is $\frac{1}{4}$ of the length of the binary representation of the exponent.

Such representations of the exponent can be used in an obvious manner for exponentiation.

7.6 Multi-Exponentiation

The multi-exponentiation is concerned with the evaluation of a product of several exponentiations. More exactly, we are interested in computing

$$\prod_{i=0}^{l-1} x_i^{n_i} \ mod \ m,$$

where $l \geq 2$, $x_0, \ldots, x_{l-1} \in \mathbf{Z}_m$ and $n_0, \ldots, n_{l-1}$ are positive integers. This can be done at the cost of l modular exponentiations, followed by $l - 1$ modular multiplications, but this method is completely inefficient.

Vector Addition Chains The problem of finding the smallest number of multiplications required to compute a product as the one above is strongly related to the problem of finding one of the shortest vector addition chains for $(n_0, \ldots, n_{l-1})$.

A *vector addition chain* of length t for an l-dimensional vector v is a list of l-dimensional vectors $v_0 < \cdots < v_{l+t-1}$ such that $v_i = (0, \ldots, 0, 1, 0, \cdots 0)$,

where 1 appears on the $(i+1)^{st}$ position, for $0 \leq i \leq l-1$, $v_{l+t-1} = v$, and for all $l \leq i \leq l+t-1$ there are $0 \leq j,\ h \leq i-1$ such that $v_i = v_j + v_h$, where $<$, $+$ denote, respectively, the ordinary vector order and the vector addition.

Straus [61] was the first who found a method for generating vector addition chains by generalizing the Brauer method for addition chains.

Having such a chain for $(n_0, \ldots, n_{l-1})$, in order to compute the result of the multi-exponentiation, we can follow the next steps:

$$\begin{aligned} y_0 &= x_0 \ mod\ m \\ &\ldots \\ y_{l-1} &= x_{l-1} \ mod\ m \\ &\ldots \\ y_{l+t-1} &= \textstyle\prod_{i=0}^{l-1} x_i^{n_i} \ mod\ m \end{aligned}$$

Except for the first l steps, the step i is connected with the previous steps by the following relation:

$$y_i = y_j \cdot y_h \ mod\ m$$

for some $0 \leq j,\ h \leq i-1$.

The required number of multiplications for exponentiation is exactly the length of the vector addition chain for the exponents vector involved in the multi-exponentiation.

Let $L([n_0, \ldots, n_{l-1}])$ be the length of an optimal vector addition chain for $(n_0, \ldots, n_{l-1})$, and $L(n_0, \ldots, n_{l-1})$ be the length of an optimal addition sequence for $n_0, \ldots, n_{l-1}$. Olivos [47] proved that the problem of finding an optimal vector chain for $(n_0, \ldots, n_{l-1})$ is equivalent to the problem of finding an optimal addition sequence for $n_0, \ldots, n_{l-1}$. More exactly, he proved that the following is true

$$L([n_0, \ldots, n_{l-1}]) = L(n_0, \ldots, n_{l-1}) + l - 1.$$

All the methods for multi-exponentiation which will be presented next are discussed only for the case of the left-to-right parsing of the exponents. Right-to-left variants are also possible.

The Simultaneous 2^w-ary Method This method was introduced by Straus [61] in terms of vector addition chains. We shall assume next that

$$n_i = [n_{i,k-1}, \ldots, n_{i,0}]_{2^w},$$

where $k = max_{0 \leq i \leq l-1} |n_i|_{2^w}$.

The simultaneous 2^w-ary method is based on

$$\begin{aligned} \textstyle\prod_{i=0}^{l-1} x_i^{n_i} &= \textstyle\prod_{i=0}^{l-1} x_i^{\sum_{j=0}^{k-1} 2^{wj} \cdot n_{i,j}} \\ &= \textstyle\prod_{i=0}^{l-1} \prod_{j=0}^{k-1} x_i^{2^{wj} \cdot n_{i,j}} \\ &= \textstyle\prod_{j=0}^{k-1} (\prod_{i=0}^{l-1} x_i^{n_{i,j}})^{2^{wj}} \end{aligned}$$

which leads to the following algorithm.

SimWindExp$(l, x_0, \ldots, x_{l-1}, n_0, \ldots, n_{l-1}, m, w)$
`input:` $l \geq 2$, $0 < x_0, \ldots, x_{l-1} < m$, $n_0, \ldots, n_{l-1} \geq 1$, and $m \geq 2$, $w \geq 1$;
`output:` $y = \prod_{i=0}^{l-1} x_i^{n_i} \ mod \ m$;
`begin`

1. `for` all $(e_0, \ldots, e_{l-1}) \in \{0, \ldots 2^w - 1\}^l$ `do`
   $x_{(e_0,\ldots,e_{l-1})} = \prod_{i=0}^{l-1} x_i^{e_i} \ mod \ m$;
2. `for` $i := 0$ `to` $l-1$ `do let` $n_i = [n_{i,k-1}, \ldots, n_{i,0}]_{2^w}$ as above;
3. $y := 1$;
4. `for` $j := k-1$ `downto` 0 `do`
   `begin`
5. $y := y^{2^w} \ mod \ m$;
6. $y := y \cdot x_{(n_{0,j},\ldots,n_{l-1,j})} \ mod \ m$;
   `end`

`end`

In this case, we scan w bits of each exponent per iteration, from left to right. For $w = 1$ we obtain the *simultaneous binary method*. As Bernstein pointed out in [3], the simultaneous binary method is often incorrectly referred to as the *Shamir's trick* for multi-exponentiation.

In the algorithm **SimWindExp** we have

- at most $2^{wl} - l - 1$ multiplications in line 1,
- kw squarings in line 5, and
- at most k multiplications in line 6.

Thus, the number of multiplications is at most $2^{wl} + kw + k - l - 1$.

Simultaneous Sliding Window Method The simultaneous sliding window method was introduced by Yen, Laih, and Lenstra [70] as a combination of the simultaneous 2^w-ary method and the sliding window technique.

This method is based on the fact that if $n_i = [w_{t-1}^i, \ldots, w_0^i]_{2^+}$ is an arbitrary block decomposition of the binary representation of n_i with $|w_j^i| = |w_j^{i'}| = \omega_j$ for all $0 \leq i, i' \leq l-1$ and $0 \leq j \leq t-1$, then $\prod_{i=0}^{l-1} x_i^{n_i} \ mod \ m$ can be rewritten as

$$\begin{aligned} \prod_{i=0}^{l-1} x_i^{n_i} \ mod \ m \quad = \quad & (\cdots((x_0^{(w_{t-1}^0)_2} \cdots x_{l-1}^{(w_{t-1}^{l-1})_2})^{2^{\omega_{t-2}}} \cdot x_0^{(w_{t-2}^0)_2} \cdots \\ & x_{l-1}^{(w_{t-2}^{l-1})_2})^{2^{\omega_{t-3}}} \cdots)^{2^{\omega_0}} \cdot x_0^{(w_0^0)_2} \cdots x_{l-1}^{(w_0^{l-1})_2}. \end{aligned}$$

We shall assume next that the windows are chosen such that

- $w_j^0 = \cdots = w_j^{l-1} = 0$, or
- there are $0 \leq i, i' \leq l-1$ such that $1 \preceq_{prefix} w_j^i$ and $1 \preceq_{suffix} w_j^{i'}$, and $\omega_j \leq w$,

for all $0 \leq j \leq t-1$ and some $w \geq 1$.

In the first case, the windows $w_j^0, \ldots, w_j^{l-1}$ form a *zero global window*, while in the other case, the windows $w_j^0, \ldots, w_j^{l-1}$ form a *odd global window*.

The corresponding algorithm is given below.

SimSlidWindExp$(l, x_0, \ldots, x_{l-1}, n_0, \ldots, n_{l-1}, m, w)$
`input:` $l \geq 2$, $0 < x_0, \ldots, x_{l-1} < m$, $n_0, \ldots, n_{l-1} \geq 1$, $m \geq 2$, and $w \geq 1$;
`output:` $y = \prod_{i=0}^{l-1} x_i^{n_i} \ mod\ m$;
`begin`

1. `for` all $(e_0, \ldots, e_{l-1}) \in \{0, \ldots 2^w - 1\}^l \setminus \{0, 2, \ldots, 2^w - 2\}^l$ `do`
   $x_{(e_0,\ldots,e_{l-1})} = \prod_{i=0}^{l-1} x_i^{e_i} \ mod\ m$;
2. `for` $i := 0$ `to` $l - 1$ `do let` $n_i = [n_{i,k-1}, \ldots, n_{i,0}]_2$ as above;
3. $y := 1$;
4. $j := k - 1$;
5. `while` $j \geq 0$ `do`
   `begin`
6. `if` $(\forall 0 \leq i \leq l-1)(n_{i,j} = 0)$ `then`
   `begin`
7. $y := y \cdot y \ mod\ m$;
8. $j := j - 1$;
   `end`
   `else`
   `begin`
9. find the smallest h such that
   $j - h + 1 \leq w$ and $(\exists 0 \leq i \leq l-1)(n_{i,h} = 1)$;
10. $y := y^{2^{j-h+1}} \ mod\ m$;
11. $y := y \cdot x_{((n_{0,j},\ldots,n_{0,h})_2,\ldots,(n_{l-1,j},\ldots,n_{l-1,h})_2)} \ mod\ m$;
12. $j := h - 1$;
    `end`
    `end`

`end`

In the algorithm **SimSlidWindExp** we have

- at most $2^{wl} - 2^{(w-1)l} - l$ multiplications in line 1,
- k squarings in lines 7 and 10, and
- $|\{(w_j^0, \ldots, w_j^{l-1}) | (w_j^0, \ldots, w_j^{l-1})$ odd global window$\}|$ multiplications in line 11.

Interleaving Sliding Window Method This technique has been proposed by Möller [44]. It is based on l window sizes, one for each exponent. The sliding window method with only zero and odd windows is used for each term $x_i^{n_i}$, but the key point is that the individual squaring steps are collected in one global squaring step. This can be done by maintaining the position of the end of each individual window and thus, the local result is multiplied with the global prospective result only when the end of the respective individual window is reached. The algorithm is presented bellow.

InterSlidWindExp$(l, x_0, \ldots, x_{l-1}, n_0, \ldots, n_{l-1}, w_0, \ldots, w_{l-1}, m)$
`input:` $l \geq 2$, $0 < x_0, \ldots, x_{l-1} < m$, $n_i, w_i \geq 1$ for $0 \leq i \leq l-1$, $m \geq 2$;
`output:` $y = \prod_{i=0}^{l-1} x_i^{n_i} \ mod\ m$;
`begin`

1. `for` all $0 \leq i \leq l-1$ and odd numbers $1 \leq e_i \leq 2^{w_i} - 1$
   `do` $x_{i,e_i} = x_i^{e_i} \ mod\ m$;
2. `for` $i := 0$ `to` $l-1$ `do let` $n_i = [n_{i,k-1}, \ldots, n_{i,0}]_2$ as above;
3. $y := 1$;
4. `for` i:=0 `to` $l-1$ `do` $position\_window_i := undefined$;
5. `for` j:=$k-1$ `downto` 0 `do`
   `begin`
6. $y := y \cdot y \ mod\ m$;
7. `for` $i := 0$ `to` $l-1$ `do`
   `begin`
8. `if` $position\_window_i = undefined$ and $n_{i,j} = 1$ `then`
   `begin`
9. `find` the smallest h such that
   $j - h + 1 \leq w_i$ and $n_{i,h} = 1$;
10. $e_i := (n_{i,j}, \ldots, n_{i,h})_2$;
11. $position\_window_i := h$;
   `end`;
12. `if` $position\_window_i = j$ `then`
   `begin`
13. $y := y \cdot x_{i,e_i} \ mod\ m$;
14. $position\_window_i := undefined$;
   `end`
   `end`
   `end`

`end`

In the algorithm **InterSlidWindExp** we have

- $\sum_{i=0}^{l-1} 2^{w_i - 1}$ multiplications in line 1,
- k squarings in line 6, and
- in line 13, the number of multiplications is exactly the total number of individual odd windows.

8 MpNT: A Multi-precision Number Theory Package

MpNT is a multi-precision number theory package, developed at the Faculty of Computer Science, "Al.I.Cuza" University of Iaşi, Romania. It is intended to be highly efficient and portable without disregarding code structure or clarity.

This C++ library was started as a base for cryptographic applications. However, it can be used in any other domain where efficient large number computations are required. Special optimizations apply for the Intel IA-32 and compatible processors.

The user interface is intended to be as intuitive, consistent, and easy-to-use as possible. This can be achieved by providing the classes that best model the mathematical concepts, hiding the actual implementation. For the time being, the library supports integer and modular arithmetic with practically unlimited precision.

MpNT is freely available for non-commercial use, according to the GNU Lesser General Public License. Any criticism and suggestions are warmly welcomed.

8.1 MpNT – Development Policies

Three main goals are to be achieved when implementing a library: *efficiency*, *portability*, and *usability*. Developing involves making a series of choices and tradeoffs that will essentially affect the characteristics of the final product. For a number theory library it is hard to completely satisfy all these requirements. Thus, many products of this kind have been developed, giving the user the possibility to choose. There is no unanimously accepted solution to this matter, and new approaches are still found every day.

A detailed presentation of the policies we adopted while designing MpNT is given below (see also [29]).

Programming language Choosing the programming language of the library essentially affects the characteristics of the final product. A well-written program using only assembly code is very fast, but lacks portability and is very hard to maintain. On the other side, developing the same program in a high-level language will make it easily portable, but some efficiency will be lost. Therefore, a compromise solution is to use both. As a high-level language, C++ makes a good choice because it retains C's ability to deal efficiently with the fundamental objects of the hardware (bits, bytes, words, addresses, etc.), and the programs are easier to understand and maintain. The assembly language should be used only for the most frequently called functions. These routines should be small, carefully optimized, and easy to rewrite for different architectures. They form the *library kernel*. It is worth mentioning that every library should offer its kernel written also in a high-level language, with the stated purpose of maintaining portability.

Therefore, we have used ISO C++ for the main part of the library for its high portability and because it is the fastest high-level programming language available. A clean and intuitive interface could be built using OOP. Assembly language is used only for the small machine-dependent kernel that is intended to

increase the performance of the library, because it is closest to what the machine architecture really provides. Special optimizations apply for the Intel I-32 and compatible processors under Windows and Linux. For portability purposes, this set of routines is also available in C++.

Number Representation Number representation highly depends on the facilities provided by the hardware architecture, including: the dimension of the microprocessor's registers, the instructions set, cache sizes, the parallelism level provided etc.

MpNT uses signed-magnitude representation for its multi-precision integers (objects of the `MpInt` class). The current implementation of the class includes four private attributes:

- `flags` – uses every bit independently to store a logical value (0 or 1). One bit stores the sign of the number: 0 for positive or null and 1 for negative. The implemented functions make sure that the number 0 never has the sign bit set. Two more bits keep special status information to avoid unnecessary copying by the overloaded operators. The other bits are yet unused.
- `msz` – the number of allocated digits.
- `sz` – the number of digits that are actually used in representation. For the number 0, the value of this field must be null.
- `rep` – a pointer to the actual number representation (an array of digits). `rep[0]` stores the least significant digit of the multi-precision integer. Normally, `rep[sz-1]` is not null if `sz` is strictly positive, thus avoiding insignificant zeroes. For best performance digits should have the size of the longest microprocessor's register that may be used for arithmetical operations. All the bits of a digit are used in representing the number, thus making it easier to write efficient assembly code.

Even though this representation uses more memory it has the advantage of providing quick access to class information. The yet unused flag bits may also store other information concerning the representation of the multi-precision integer.

Library structure Developing an easy to maintain and expand library requires some sort of modular structure. Most of the current libraries group their functions in different layers, each of them having a different level of abstraction. It is desirable that only low-level (kernel) routines have direct access to the number representation. This gives the functions in the higher layers a degree of independence.

The MpNT library is structured on two layers: the kernel and the C++ classes. Most of the kernel functions operate on unsigned arrays of digits, such as: comparisons, bit-shifts and basic arithmetical operations. However, they are risky to use because they assume that enough memory has been allocated for the results and that certain relations between the operands hold.

Because of similarities in the number representation, the capability of using the GMP [24] or even the CLN [27] kernel as an alternative may be easily added.

The user interface is intended to be as intuitive, consistent, and easy-to-use as possible. This can be achieved by providing the classes that best model the mathematical concepts, hiding the actual implementation. Backward binary compatibility throughout the library development is more than desirable. In our case, the C++ classes, such as `MpInt` and `MpMod`, provide a safe and easy to use interface.

The `MpInt` Class
: The `MpInt` class provides multi-precision integer arithmetic. An object of the `MpInt` class signifies an integer, in the amount and sign representation. It can be regarded as an array of binary digits (the least significant digit is indexed by 0). The user has access to this representation (reading and writing) by indicating either the individual bits or groups of bits.

  Functions performing arithmetical operations are provided: addition, subtraction, multiplication, division, greatest common divisor, bit operations etc. All operators which are available for the `int` type are also defined for objects of the `MpInt` class, therefore they can be regarded as normal integers, but with no length restrictions.

  The `MpInt` class also hides the functions of the kernel, providing the classes that rely upon it with a high level of independence.

The `MpMod` Class
: The `MpMod` class provides Multi-precision modular arithmetic. Only one modulus can be used at any time, and all the computations are performed using it. Also, all the interface functions of the class ensure that the number is always modularly reduced. Functions determining the multiplicative inverse, and performing modular multiplication and exponentiation are provided along with other basic operations (addition, subtraction etc.).

Algorithm selection In many cases several algorithms may be used to perform the same operation depending on the length of the operands. Of course, the ones with the best $\mathcal{O}$-complexity are preferred when dealing with huge numbers, but on smaller operands a simpler, highly optimised algorithm may perform much better. This is why careful performance testing is required to find out the limits of applicability.

Even though in MpNT we implemented more than one algorithm for some operations, the interface functions will use only the routines or the combination of routines proved to be most efficient.

Memory management The most used memory allocation policy is to allow the user to explicitly allocate memory (on demand allocation). Additional space may be transparently allocated whenever a variable does not have enough (e.g., GMP, NTL). Memory leaks may be prevented by the use of class destructors. This is easy to implement but the user has responsibilities in managing memory. The drawback may be eliminated by using a garbage collector (e.g., CLN), but the speed loss could be unacceptable. Some libraries give the user the possibility to choose the allocation technique that best suits his application or even to use his own memory management routines (e.g., LiDIA).

The memory management policy adopted in MpNT is based on explicit allocation of memory. To avoid frequent reallocation, when the exact amount of

necessary memory is known, the user may make such a request. For the same reason, whenever reallocation occurs, we provide more space than needed. Memory may be released either on demand or automatically by the class destructors. Currently, the C++ operators `new` and `delete` are used, but we intend to allow the user to choose the preferred allocation functions.

Error handling A desirable approach is to signal the occurred errors, allowing the user to choose the actual handling policy. This involves supplementary checking and is time consuming, making the code harder to read and maintain. Therefore the most frequent approach is to ignore errors, which surely involves some risks, but eliminates the overhead.

Unlike other libraries, we chose not to ignore errors. When an error is detected an exception is thrown, depending on the error type (division by zero, infinite result, allocation failure etc.), and the user may choose whether to handle it or not.

8.2 Implementation Notes

These notes apply to MpNT 0.1 only and are prone to change as the library develops.

Addition and Subtraction As the algorithms for addition and subtraction are simple, their implementation is very straightforward. No special hardware support is needed because, when adding two digits, the carry can be tested by simply checking if the sum is greater than an operand. After subtracting two digits one can test if the difference is greater than the first operand, yielding borrow. Modern processors offer however an add-with-carry (subtract-with-borrow) instruction which is used in our assembly addition (subtraction) routine. In the low-level addition (subtraction) function the digits of the operands are added (subtracted) starting with the least significant one and taking care of the carries (borrows) as they appear. It is up to the high-level function to check the signs of the numbers and chose whether to perform addition or subtraction. This function also takes care of memory allocation and treats the case when an operand is used to store the result (in-place operation), which is easy since addition (subtraction) can generally be done in-place.

Multiplication As it is more expensive than addition and subtraction, multiplication has to be taken special care of. Hardware support is needed since when two digits are multiplied a two-digit product is obtained. Since C++ does not offer the possibility of easily obtaining that product, a "Karatsuba-like" trick is being used with a great speed loss. However this shortcoming is surpassed by the use of assembly language. The high-level function only takes care of the memory allocation and sign and then calls the appropriate low-level one.

School multiplication has been carefully implemented using assembly language, and it is the best approach when dealing with relatively small numbers. Even more optimizations apply when squaring is performed. The point-wise products computed during multiplication can be arranged in a rectangular matrix that is symmetrical in the case of squaring, so almost half of the products (the ones below the diagonal) do not really have to be computed. They are

obtained by a one bit left shift and finally the products on the diagonal are added to obtain the final result.

The implementation of Karatsuba multiplication follows closely the algorithm in Section 3.2. No special optimizations apply, not even when squaring since only a $k/2$-digit subtraction and a comparison are saved. Nevertheless the algorithm is simple and has a much better time complexity than the school method. That is why, starting with several dozens of digits, it is the algorithm of choice for our library. On very large numbers however, the modular FFT algorithm performs better. The Toom-Cook algorithm [35] might just fill in the gap between Karatsuba and the FFT, but the use of such large numbers in our cryptographic applications is yet to be determined.

Division Most of the divisions performed by the algorithms presented in Section 4 are two digits divided by one producing a quotient and remainder having one digit. Since C++ does not offer this possibility, each digit is split in half and an approach similar to recursive division is followed, not without a sacrifice in speed. Nevertheless since the hardware architecture generally provides such an operation, assembly language is used to do things the fast way. The high-level division function has to take care of the memory allocation and the cases when one or both of the operands are negative. Also in-place operations have to be supported although the lower-level functions usually do not.

The **SchoolDiv2** algorithm is the simple way to divide by a one digit divisor so an optimized assembler version exists. The **SchoolDiv1** algorithm is much more complex, because it needs to normalize the operands and the quotient test used is rather complicated, so only C++ was used. The running time of this algorithm is close to the time needed to multiply the operands using school multiplication. On the other hand, the algorithm does not benefit from the asymptotical improvement brought by Karatsuba and higher algorithms, so it is used only when dealing with relatively small numbers.

Recursive division is used when the size of the divisor is over a certain limit (usually twice the Karatsuba limit) and if the length of the dividend is twice the divisors length. The second condition arises because the zero padding needed in that case would degrade the algorithm performance. Actually no zero padding is performed by algorithm implementation, thus a great efficiency gain. The **RecursiveDiv** algorithm pads a with non-significant zeroes so that its length becomes a multiple of m. Instead we simply split a into m digit pieces starting with the most significant digit, and divide them by b using $D_{2/1}$. The least significant piece has $d < n$ digits, so the last step uses School division to get the final result since x has only $d+m < 2m$ digits. It was also required that n/m was a power of two so that the divisor in $D_{2/1}$ has always an even number of digits. However the $D_{2/1}$ routine was designed so that no such restrictions apply. The approach is similar to the odd length case in the Karatsuba algorithm. If m is odd we first compute $(q', r') := D_{2/1}(a/\beta^2, b/\beta)$ and regard q' as an estimation of q/β. According to Theorem 4.1.1 (with β^m as a base) this estimation is at least two bigger than the real q/β, so at most two additional subtractions have to be performed. The final results are computed by dividing the $m+1$ digit reminder r' by b to produce q_0, the least significant digit of q, and r the final reminder. This can be done by a single step of the school division.

The greatest common divisor The Euclidean algorithm for computing $gcd(a, b)$ is not the best choice when a and b are multi-precision integers, because the divisions are quite slow. For small numbers, the binary gcd algorithm is better, as it does not use any divisions, but only shift operations and subtractions. The main disadvantage of the binary algorithm is that, on large inputs, the number of steps performed to compute the gcd is also very large, which results in a slow algorithm.

Lehmer's gcd algorithm proved to be much better than the binary or Euclidean methods. The number of step performed is the same as in Euclid's algorithm, but a lot of consecutive multi-precision division steps are replaced by simple-precision division steps, so the improvement is quite remarkable.

In practice, the best of Lehmer's gcd algorithms turned out to be **Lehmer1**. If we denote n the number of bits in a digit, our choices for β, p and h are: $\beta = 2$, $p = n - 2$, $h = |a|_2 - p$. We chose $p = n - 2$ because, for the simple-precision integers used in the algorithm, we need 1 bit for the sign and 1 bit for the possible overflow that may result. Because of the choice we made regarding h, there is a chance that b be zero. This case needs no special treatment because $b + C$ will also be 0 and the while ($b + C \neq 0$ and $b + D \neq 0$) loop will not be performed, B will be 0 and a Euclidean step will be performed.

A useful remark for implementing Lehmer's algorithm is that $Aa + Bb$ and $Ca + Db$ will always be computed by subtraction and the results will always be positive numbers. Therefore, a, b, X, and Y may be treated as positive multi-precision integers.

When the algorithm exits the main while loop, b is a simple-precision number and a may be multi-precision. If we perform a Euclidean step, then both a and b will become simple-precision numbers and the next Euclidean steps will be very fast.

Collins' and Jebelean's quotient tests can be used instead of Lehmer's quotient test, but the algorithm will not get any faster.

The extended gcd algorithms are more difficult and more expensive, as the computation of α and β such that $\alpha a + \beta b = gcd(a, b)$ involves a lot of multi-precision operations.

The Euclidean extended gcd algorithm is very slow; it can be slightly improved. We made the remark that the sequence (α_i) and (β_i) can be computed independently. Thus, if we compute only the sequence (β_i), we will determine β such that $\alpha a + \beta b = gcd(a, b)$ and than α can be computed as $(gcd(a, b) - \beta b)/a$. This trick saves a lot of multi-precision operations (additions, subtractions and multiplications). But the improvement proved not to be sufficient, and the Euclidean extended gcd algorithm is still very slow.

A better choice for the extended gcd algorithm is **Lehmer1Ext**. Like the improved version of Euclid's algorithm, **Lehmer1Ext** only computes the gcd and u, because v can be determined as $v = (gcd(a, b) - au)/b$. The values for β, p and h are as in **Lehmer1**, $Aa + Bb$ and $Ca + Db$ are actually subtractions, and $Au_0 + Bu_1$ and $Cu_0 + Du_1$ are always additions.

The algorithm can be slightly improved regarding the case in which no simple-precision step can be made ($B = 0$). If X is a simple-precision integer, and this happens often enough, the operation $Y = u - Xv$ can be performed faster, so this case may be treated separately. But the improvement is not significant, as the case $B = 0$ is very infrequent.

The binary extended gcd algorithm does not behave better than Lehmer's algorithm.

Modular arithmetic Two more classes are considered for modular arithmetic, `MpModulus` and `MpLimLee`. The first one wraps a `MpInt` number that represents the modulus. This value is a positive integer greater or equal than 2. Once a `MpInt` is assigned to a `MpModulus` instance, the necessary pre-computation is automatically performed and stored in the class members. The management of those values is made transparent to the user. The mathematical operations that are facilitated by the use of this class are the modular reduction, addition, subtraction, multiplication, squaring and exponentiation. In each case the best way is automatically chosen w.r.t. the operands' particularities (size, parity). The class improves the memory management. Each `MpModulus` object reserves some space for the intermediary operations. It is released only when the user explicitly asks for it or when the object is destroyed. By doing so the number of memory allocations is highly reduced.

The second class, `MpLimLee`, implements the Lim-Lee method for fixed base modular exponentiation algorithm. It handles the pre-computation just like the `MpModulus` class.

For modular reduction, regular division (school and recursive), Barrett and Montgomery methods, are used. Modular addition and subtraction are carried out between integers and the result is reduced. Modular multiplication is mainly used for modular exponentiation (otherwise, it is performed by multiplication and reduction).

The sliding window method using only zero and odd windows analyzes its input values and selects the most appropriate window size as in Table 1.

	3	4	5	6	7	8	9
32	**44**	47	54	69	100	164	292
64	**84**	85	91	105	136	199	327
128	164	**162**	166	179	208	271	397
256	324	316	**315**	325	352	413	538
512	644	623	**614**	618	640	697	820
1024	1285	1237	1211	**1203**	1216	1266	1383
2048	2564	2466	2406	2373	**2368**	2403	2509
4096	5124	4923	4795	4713	**4672**	4679	4762
8192	10245	9839	9573	9394	9280	**9231**	9267

Table 1: Average number of modular reductions for the sliding window exponentiation algorithm. The first row presents the window sizes. Every other line begins with the bit-length of the exponent. The marked values indicate the best choices for the corresponding bit lengths of the exponent. The correspondence is the same as in GMP.

For *fixed base exponentiation*, the Lim-Lee's algorithm is the best. The im-

plementation is supported by the `MpLimLee` class and follows the steps described in Section 7. We added routines for pre-computation and exponentiation that may use the Montgomery modular reduction method. For *multi-exponentiation*, the *interleaving sliding window* algorithm is the best.

A general note for all the exponentiation algorithms is the following. If the modulus is odd then the reduction technique that may be used is the one presented by Montgomery.

8.3 Other Libraries

A general description of the most well-known Multi-precision libraries is given below. Most of them are freely available for non-commercial purposes, according to the GNU General Public License (GMP uses Lesser GPL).

LIP [40] (Large Integer Package) is a library containing a variety of functions for arithmetic on arbitrary length signed integers that is available from Bellcore. These functions allow easy prototyping and experimentation with new number theory-based cryptographic protocols. LIP is written entirely in ANSI C, has proved to be easily portable, is intended to be easy to use, and achieves an acceptable performance on many different platforms, including 64-bit architectures. It includes functions that perform addition, subtraction, multiplication, division, exponentiation, modular reduction, primality testing, square root extraction, random number generator, and a few other tasks.

The software was originally written by Arjen K. Lenstra and later maintained by Paul Leyland.

LiDIA [25] (a C++ library for computational number theory) provides a collection of highly optimized implementations of various multi-precision data types and time-intensive algorithms. It includes routines for factoring integers, determining discrete logarithm in finite fields, counting points on elliptic curves etc. LiDIA allows the user to work with different multi-precision integer packages and memory managers like Berkeley MP, GNU MP, CLN, libI, freeLIP etc.

The library was developed at Universität des Saarlandes, Germany and organized by Thomas Papanikolaou.

CLN [27] (a Class Library for Numbers) permits computations with all kinds of numbers. It has a rich set of number classes: integers (with unlimited precision), rational numbers, floating-point numbers, complex numbers, modular integers (integers modulo a fixed integer), univariate polynomials. CLN is a C++ library that implements elementary arithmetical, logical and transcendental functions. It is memory and speed efficient aiming to be easily integrated into larger software packages.

CLN was written by Bruno Haible and is currently maintained by Richard Kreckel.

NTL [57] (a Library for doing Number Theory) is a high-performance, portable C++ library providing data structures and algorithms for arbitrary length integers; for vectors, matrices, and polynomials over the integers and over

finite fields; and for arbitrary precision floating point arithmetic. NTL is written and maintained mainly by Victor Shoup. NTL's polynomial arithmetic is one of the fastest available anywhere, and has been used to set "world records" for polynomial factorization and determining orders of elliptic curves. NTL's lattice reduction code is also one of the best available anywhere, in terms of both speed and robustness. NTL is easy to install and achieves high portability by avoiding esoteric C++ features, and by avoiding assembly code. However, it can be used in conjunction with GMP for enhanced performance.

The **PARI/GP** [26] system is a package which is capable of doing formal computations on recursive types at high speed; it is primarily aimed at number theorists, but can be used by anybody whose primary need is speed. It was originally developed at Bordeaux, France, by a team led by Henri Cohen. The three main advantages of the system are its speed, the possibility of using directly data types, which are familiar to mathematicians, and its extensive algebraic number theory module.

PARI can be used as a library, which can be called from an upper-level language application (for instance written in C, C++, Pascal or Fortran) or as a sophisticated programmable calculator, named GP, which contains most of the control instructions of a standard language like C.

GMP [24] (GNU Multiple Precision arithmetic library) is a C library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. The main target applications for GMP are cryptography applications and research, Internet security applications, algebra systems, computational algebra research, etc.

GMP is carefully designed to be as fast as possible, both for small operands and for huge operands. The speed is achieved by using full words as the basic arithmetic type, by using fast algorithms, with highly optimized assembly code for the most common inner loops for a lot of CPUs, and by a general emphasis on speed.

GMP is faster than any other multi-precision library. The advantage for GMP increases with the operand sizes for many operations, since GMP uses asymptotically faster algorithms. The original GMP library was written by Törbjorn Granlund, who is still developing and maintaining it. Several other individuals and organizations have contributed to GMP.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, third edition, 1976.

[2] D.P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer-Verlag, 1987.

[3] D. J. Bernstein. Pippenger's exponentiation algorithm. Preprint, University of Illinois at Chicago, 1991. http://cr.yp.to/papers.html.

[4] I. E. Bocharova and B. D. Kudryashov. Fast exponentiation in cryptography. In G. Cohen, M. Giusti, and T. Mora, editors, *Applied algebra, algebraic algorithms and error-correcting codes*, volume 948 of *Lecture Notes in Computer Science*, pages 146–157. Springer-Verlag, 1995.

[5] J. Bos. *Practical Privacy*. PhD thesis, Technische Universiteit Eindhoven, 1992.

[6] J. Bos and M. Coster. Addition chain heuristics. In G. Brassard, editor, *Advances in Cryptology—CRYPTO '89*, Lecture Notes in Computer Science, pages 400–407. Springer-Verlag, 1990.

[7] W. Bosma. Signed bits and fast exponentiation. Technical Report 9935, Department of Mathematics, University of Nijmegen, 1999.

[8] A. Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45:736–739, 1939.

[9] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation: Algorithms and lower bounds, 1995. http://research.microsoft.com/ dbwilson/bgmw/.

[10] Ch. Burnikel and J. Ziegler. Fast recursive division. Technical Report MPI-I-98-1-022, Max-Planck-Institut für Informatik, Saarbrücken (Germany), October 1998.

[11] Ç. K. Koç. Analysis of sliding window techniques for exponentiation. *Computers and Mathematics with Applications*, 30(10):17–24, 1995.

[12] H. Cohen. Analysis of the flexible window powering algorithm. Preprint, University of Bordeaux, 1991. http://www.math.u-bordeaux.fr/ cohen/.

[13] P.M. Cohn. *Algebra*, volume I. John Wiley & Sons, 1982.

[14] G.E. Collins. Lecture notes on arithmetic algorithms. University of Wisconsin, 1980.

[15] T. Collins, D. Hopkins, S. Langford, and M. Sabin. Public key cryptographic apparatus and method. United States Patent 5, 848, 159, USA, 1997.

[16] J.M. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[17] P. Downey, B. Leong, and R. Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 10(3):638–646, 1981.

[18] S.R. Dussé and B.S. Kaliski. A cryptographic library for the Motorola DSP56000. In *Advances in Cryptology EUROCRYPT'90*, volume 473 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, 1991.

[19] Ö. Eğecioğlu and Ç. K. Koç. Fast modular exponentiation. *Communication, Control, and Signal Processing*, pages 188–194, 1990.

[20] Ö. Eğecioğlu and Ç. K. Koç. Exponentiation using canonical recoding. *Theoretical Computer Science*, 129(2):407–417, 1994.

[21] H. Garner. The residue number system. *IRE Transactions on Electronic Computers*, EC-8:140–147, 1959.

[22] D. Gollmann, Y. Han, and C.J. Mitchell. Redundant integer representations and fast exponentiation. *Designs, Codes and Cryptography*, 7:135–151, 1996.

[23] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley Publishing Company, second edition, 1994.

[24] GMP Group. GMP. http://www.swox.com/gmp/.

[25] LiDIA Group. *LiDIA. A C++ Library for Computational Number Theory*. Darmstadt University of Technology. http://www.informatik.tu-darmstadt.de/TI/LiDIA/.

[26] PARI-GP Group. *PARI-GP*. Université Bordeaux. http://www.parigp-home.de/.

[27] B. Haible and R. Kreckel. CLN – Class Library for Numbers. http://www.ginac.de/CLN/.

[28] S.-M. Hong, S.-Y. Oh, and H. Yoon. New modular multiplication algorithms for fast modular exponentiation. In U. M. Maurer, editor, *Advances in Cryptology-EUROCRYPT'96*, volume 1070 of *Lecture Notes in Computer Science*, pages 166–177. Springer-Verlag, 1996.

[29] C. Hriţcu, I. Goriac, R. M. Gordân, and E. Erbiceanu. Designing a multiprecision number theory library. In *Concurrent Information Processing and Computing*. IOS Press, 2003. (to appear).

[30] S. Iftene. Modular exponentiation. In *Concurrent Information Processing and Computing*. IOS Press, 2003. (to appear).

[31] T. Jebelean. Improving the multiprecision Euclidian algorithm. In *International Symposium on Design and Implementation of Symbolic Computation Systems DISCO'93*, volume 722 of *Lecture Notes in Computer Science*, pages 45–58. Springer-Verlag, 1993.

[32] T. Jebelean. Practical integer division with Karatsuba complexity. In Wolfgang Küchlin, editor, *International Symposium on Symbolic and Algebraic Computation ISSAC'97*, pages 339–341. ACM Press, July 1997.

[33] A. Karatsuba and Y. Ofman. Multiplication of multiplace numbers on automata. *Dokl. Acad. Nauk SSSR*, 145(2):293–294, 1962.

[34] S. Kawamura, K. Takabayashi, and A. Shimbo. A fast modular exponentiation algorithm. *IEICE Trans.Fundam*, E74(8):2136–2142, 1991.

[35] D. E. Knuth. *The Art of Computer Programming. Seminumerical Algorithms*. Addison-Wesley, third edition, 1997.

[36] D.E. Knuth. *The Art of Computer Programming. Seminumerical Algorithms*. Addison Wesley, second edition, 1981.

[37] K. Koyama and Y. Tsuruoka. Speeding up elliptic cryptosystems by using a signed binary window method. In E. F. Brickell, editor, *Advances in Cryptology - CRYPTO '92*, volume 740 of *Lecture Notes in Computer Science*, pages 345–357. Springer-Verlag, 1992.

[38] G. Lamé. Note sur la limite du nombre des divisions dans la recherche du plus grand commun diviseur entre deux nombres entiers. *Comptes rendus de l'academie des sciences*, tome 19:867–870, 1844.

[39] D.H. Lehmer. Euclid's algorithm for large numbers. *American Mathematical Monthly*, 45:227–233, 1938.

[40] A. Lenstra. *Long Integer Programming (LIP) Package*. Bellcore. http://usr/spool/ftp/pub/lenstra/LIP.

[41] C. H. Lim and P. J. Lee. More flexible exponentiation with precomputation. In Y. Desmedt, editor, *Advances in Cryptology—CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107. Springer-Verlag, 1994.

[42] D.-C. Lou and C.-C. Chang. An adaptive exponentiation method. *Journal of Systems and Software*, 42:59–69, 1998.

[43] R. Moenck and A. Borodin. Fast modular transforms via division. In *The 13th Annual IEEE Symposium on Switching and Automata Theory*, pages 90–96, 1972.

[44] B. Möller. Algorithms for multi-exponentiation. In S. Vaudenay and A.M. Youssef, editors, *Selected Areas in Cryptography–SAC 2001*, volume 2259 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 2001.

[45] B. Möller. Improved techniques for fast exponentiation. In P.J. Lee and C.H. Lim, editors, *Information Security and Cryptology - ICISC 2002*, volume 2587 of *Lecture Notes in Computer Science*, pages 298–312. Springer-Verlag, 2002.

[46] P.L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[47] J. Olivos. On vectorial addition chains. *Journal of Algorithms*, 2:13–21, 1981.

[48] N. Pippenger. On the evaluation of powers and related problems. In *17th Annual Symposium on Foundations of Computer Science*, pages 258–263. IEEE Computer Society, 1976.

[49] N. Pippenger. The minimum number of edges in graphs with prescribed paths. *Mathematical Systems Theory*, 12:325–346, 1979.

[50] N. Pippenger. On the evaluation of powers and monomials. *SIAM Journal on Computing*, 9:230–250, 1980.

[51] M. Quercia. Calcul multiprécision. Technical report, INRIA, April 2000. http://pauillac.inria.fr/quercia/divrap.ps.gz.

[52] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for the RSA public-key cryptosystem. *IEE Electronics Letters*, 18(21):905–907, 1982.

[53] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[54] P. De Rooij. Efficient exponentiation using precomputation and vector addition chains. In Alfredo De Santis, editor, *Advances in Cryptology-Eurocrypt'94*, volume 950 of *Lecture Notes in Computer Science*, pages 389–399. Springer-Verlag, 1994.

[55] A. Schönhage. Schnelle berechnung von kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.

[56] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7:281–292, 1971.

[57] V. Schoup. *NTL: A Library for Doing Number Theory.* Courant Institute. http://www.shoup.net/ntl/.

[58] J. Sorenson. Two fast GCD algorithms. *Journal of Algorithms*, 16:110–144, 1994.

[59] J. Sorenson. An analysis of Lehmer's Euclidian GCD algorithm. In *International Symposium on Symbolic and Algebraic Computation ISSAC'95*, pages 254–258. ACM Press, 1995.

[60] J. Stein. Computational problems associated with Racah algebra. *Computational Physics*, 1:397–405, 1967.

[61] E.G. Straus. Addition chains of vectors. *American Mathematical Monthly*, 71:806–808, 1964.

[62] S.R. Tate. Stable computation of the complex roots of unity. *IEEE Transactions on Signal Processing*, 43(7):1709–1711, 1995.

[63] E.G. Thurber. On addition chains $l(mn) \leq l(n) - b$ and lower bounds for $c(r)$. *Duke Mathematical Journal*, 40:907–913, 1973.

[64] F.L. Ţiplea. Algebraic Foundations of Computer Science. Course Notes, Faculty of Computer Science, "Al.I.Cuza" University of Iaşi, Romania, `http://thor.info.uaic.ro/ fltiplea/`.

[65] F.L. Ţiplea. Coding Theory and Cryptography. Course Notes, Faculty of Computer Science, "Al.I.Cuza" University of Iaşi, Romania, `http://thor.info.uaic.ro/ fltiplea/`.

[66] F.L. Ţiplea. Security Protocols. Course Notes, Faculty of Computer Science, "Al.I.Cuza" University of Iaşi, Romania, `http://thor.info.uaic.ro/ fltiplea/`.

[67] F.L. Ţiplea. *Introduction to Set Theory*. "Al.I.Cuza" University Press, Iaşi, 1998.

[68] Y. Yacobi. Fast exponentiation with addition chains. In Ivan B. Damgård, editor, *Advances in Cryptology: EUROCRYPT '90*, volume 473 of *Lecture Notes in Computer Science*, pages 222–229. Springer-Verlag, 1990.

[69] A. C. Yao. On the evaluation of powers. *SIAM Journal on Computing*, 5:100–103, 1976.

[70] S.-M. Yen, C.-S. Laih, and A.K. Lenstra. Multi-exponentiation. In *IEE Proceedings-Computers and Digital Techniques*, volume 141, pages 325–326, 1994.

[71] P. Zimmermann. A proof of GMP fast division and square root implementations, September 2000. http://www.loria.fr/zimmerma/papers.

[72] P. Zimmermann. Arithmétique en précision arbitraire. Technical Report 4272, INRIA, September 2001.

[73] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, 1978.