SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Arbitrary-precision arithmetic in various algebraic structures

av

Gabriel Netterdag

2010 - No 1

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET, 106 91 STOCKHOLM

Arbitrary-precision arithmetic in various algebraic structures

Gabriel Netterdag

Självständigt arbete i matematik 30 högskolepoäng, grundnivå

Handledare: Torsten Ekedahl

2010

Abstract

This report will present some of the more commonly used methods for performing arbitrary-precision arithmetic. The ability to do arithmetic efficiently with numbers containing hundreds or thousands of digits and in some cases even more, is vital in many computational fields. Cryptography is a major real-world example which takes advantage of these methods. It will introduce, as a reference, the Diffie-Hellman key exchange protocol. Furthermore, it will discuss the methods of classic arithmetic, modular arithmetic and Karatsuba arithmetic. The report will also introduce arithmetic methods based on the Fourier transform and its generalization as well as special representations in finite fields. Finally some concrete computer implementations are given with comparisons between the different algorithms.

Contents

Chapter 1

Arbitrary-precision arithmetic

1.1 Introduction

From their introduction to the present time, computers have been used for arithmetic calculations of various sorts. They are capable of performing basic arithmetic like addition and multiplication efficiently. One limiting factor of a normal general-purpose computer is that it performs arithmetic using a fixed number of *registers* each having a fixed size for representing numbers. The operands of a computer instruction are expected to be given in these registers and the result is usually returned in some other register. Today, personal computers commonly have instructions that operate on register sizes and memory element sizes of 32 bits or 64 bits. We call these registers and memory elements a *computer word.* For example, a 32-bit computer can represent unsigned integers in the interval $[0, 2^{32})$ and for a 64-bit computer the interval would be $[0, 2^{64})$. Calculating with larger quantities poses a problem as we are limited to these word sizes. The general idea to overcome this limitation is to use the main memory of a computer to represent these larger integers. Usually the main memory has much more storage capacity than the set of registers available in the computer.

Arbitrary-precision arithmetic is a collection of methods and techniques, making it possible for computers to do arithmetic calculations involving arbitrary numbers of digits.[1] This enables the possibility to do computer arithmetic involving millions of digits or more. As the number of digits increase, it is of great importance to have methods that are efficient with respect to computational time. We want to spend as little time as possible performing these arithmetic calculations. Later, we will discuss the efficiency of methods

[1]As computers have finite storage capacity, the word *arbitrary* should not be taken literally. The arbitrariness is limited to the computers storage capacity.

with respect to time, given a certain amount of input.

Definition 1. *If the worst-case computational time for an algorithm is $f(x)$ where x is the problem size, we say that $f(x)$ is of complexity order $\mathcal{O}(g(x))$ when $x \to \infty$ if and only if*

$$\exists x_0, \exists M > 0, \forall x > x_0 : |f(x)| \leq M|g(x)|$$

This will be denoted as $f(x) \in \mathcal{O}(g(x))$. Furthermore, the computational time $f(x)$ is said to be polynomial *if $g(x)$ can be chosen to be a polynomial function.*

Algorithms that have polynomial time complexity are considered as "feasible" computations as opposed to *super-polynomial time*, which is anything slower.

Example If the computational time for some algorithm is $f(x) = 4x^3 - 3x + 7$ we have that

$$|4x^3-3x+7| \leq |4x^3+3x+7| \leq |4x^3+3x^3+7x^3| = 14x^3, \quad x > 1 \quad (1.1)$$

Therefore $f(x) \in \mathcal{O}(x^3)$ and it is polynomial.

One area where arbitrary-precision arithmetic is extensively used is in cryptography. With the growth of the Internet we are doing more sensitive transactions of information that need protection from any unauthorized party. Naively, one may say that the greater length of encryption keys etc. the harder it is is to attack such an encryption protocol. Such lengths of encryption keys require efficient arithmetic in order to be useful. There are of course other areas where arbitrary-precision methods are used. This includes error correcting codes, finding large primes and calculating π with billions of digits, the latter perhaps more of a recreational nature.

Even though it isn't the primary concern of this report to present cryptographic theory, it is nevertheless a good idea to have some basic understanding of some well known cryptographic method in order to get a grasp of the computations performed. The chosen method is the Diffie-Hellman key exchange which uses simple algebraic properties and is easily explained.

1.2 Diffie-Hellman key exchange

The Diffie-Hellman key exchange is a cryptographic protocol for establishing a secure key between two parties over an insecure communication medium.

This key may then be used with other encryption methods for the remainder of the communication session. The procedure was invented in 1976, jointly by Whitfield Diffie and Martin Hellman.[2]

The protocol is based on two parties, let's call them $\mathcal{A}$ and $\mathcal{B}$, agreeing on the the following two things:

- A cyclic group G of finite order.
- A generator $g \in G$.

The two items above is generally decided long before any other part of the protocol is set in action.

In order to have $\mathcal{A}$ and $\mathcal{B}$ agree on a secret key the following steps are performed.

1. $\mathcal{A}$ picks a random number $a \in \mathbb{Z}^+$ and sends g^a to $\mathcal{B}$,
2. $\mathcal{B}$ picks a random number $b \in \mathbb{Z}^+$ and sends g^b to $\mathcal{A}$,
3. $\mathcal{A}$ computes $(g^b)^a$,
4. $\mathcal{B}$ computes $(g^a)^b$.

Because of the associative axiom of a group both $\mathcal{A}$ and $\mathcal{B}$ are in the possession of the element $(g^b)^a = g^{ba} = g^{ab} = (g^a)^b$, which is the secret key.

As a simple example, let $G = \mathbb{Z}_5^*$ and let $g = 3$ as the generator for this group. Following the algorithm above, we get

1. $\mathcal{A}$ picks the random number $a = 7$ and therefore sends $3^7 \bmod 5 = 2$
2. $\mathcal{B}$ picks the random number $b = 10$ and therefore sends $3^{10} \bmod 5 = 4$
3. $\mathcal{A}$ computes $4^7 \bmod 5 = 4$
4. $\mathcal{B}$ computes $2^{10} \bmod 5 = 4$

Both $\mathcal{A}$ and $\mathcal{B}$ have computed the same secret key which is 4. The example is of course very simplified. A real-world implementation would require much larger values for a, b and the order of G to make it secure (for a given definition of secure) for any eavesdropping.

[2]M. Hellman has pointed out that their protocol is based on concepts introduced by Ralph Merkle, so Hellman suggested that if names are to be associated with this protocol it should be called: Diffie-Hellman-Merkle key exchange.

The security aspect of the algorithm is based on the fact that even if we know the generator g and see the values g^a and g^b passing between $\mathcal{A}$ and $\mathcal{B}$, it is difficult to calculate the secret key shared by $\mathcal{A}$ and $\mathcal{B}$ (given that a, b and the order of G is selected appropriately). This is due to the *Diffie-Hellman problem* which states that the following is a hard problem:

$$\text{Given } g \text{ and the values of } g^x, g^y \text{ find } g^{xy}.$$

The most efficient way known to solve this problem is to solve the related *discrete logarithm* problem,

$$\text{Given } g \text{ and } g^x, \text{ find the residue of } x \text{ modulo } |G|$$

As of this date, no publicly known way of solving these problems in polynomial time has been discovered and it remains an open question if these problems indeed are hard to solve as no formal proof of this exists. The naive approach would be to compute g^k for each k until $g^k = g^x$. There are some more sophisticated algorithms which is computationally faster than the naive approach but still not in polynomial time. They are in fact of exponential order.

It should be pointed out that this doesn't make the protocol invulnerable. A man-in-the-middle (denoted $\mathcal{C}$) attack could intercept the values passed between $\mathcal{A}$ and $\mathcal{B}$ and substitute these with its own. This would make $\mathcal{A}$ and $\mathcal{C}$ share one key and $\mathcal{B}$ and $\mathcal{C}$ share another key. Now $\mathcal{C}$ may receive messages from either one and then re-compose (potentially changing) the message and send it on to the other using that particular shared key.[3]

Using the Diffie-Hellman protocol with, for example, the multiplicative group of integers modulo m would require us to calculate powers of the generating element g. In addition to this, we could expect that the order of G is large and that the powers of the generating element g are integers with several hundred digits, in order to fulfill some notion of security. With these constraints at our hand, we see that it in order to use such a method effectively, we need ways of doing fast arithmetic with large integers. Later sections will introduce different techniques and representations for finite fields which could be used for doing fast arithmetic in the Diffie-Hellman case.

[3]This vulnerability is due to the lack of *authentication* between $\mathcal{A}$ and $\mathcal{B}$ which isn't considered in the Diffie-Hellman protocol.

Chapter 2

The algorithms and their properties

2.1 Classical arithmetic algorithms

All arithmetic methods presented here are based on the very simple fact that we may express integers in a positional system. This enables us to work with the individual digits that constitute the whole integer, simplifying the arithmetic. Seen in this light, these methods are an example of a divide and conquer technique as we break a large problem into several smaller ones.

The algorithms calculate the results much in the same way as we would do with pencil and paper. Let us consider how we usually add two integers. We write them one over the other, digit by digit. Starting from the right we add the digits in the same column, propagating any carry that may occur to the next column, until we have gone through all columns.

The multiplication algorithm is sometimes referred to as "naive" multiplication. The "naiveness" has to do with the fact that there are algorithms that use a smaller number of arithmetic operations and would therefore be executional faster when it comes to computer implementations. This becomes especially important as the number of digits increase. Before we look further into this arithmetic we should do some kind of formalization of the key tool we are going to use in this section. Furthermore, to make the presentation more straightforward, we assume that we work with non-negative integers and polynomials over these integers. A computer implementation may for example use an externally handled sign and take appropriate action whenever necessary in order to handle negative integers with the methods presented here.

Theorem 1. *Every integer $n \geq 0$ can be written uniquely on the form,*

$$a_0 + a_1 B + \cdots + a_{m-1} B^{m-1}$$

for some integers $m > 0$, $B > 1$ and $0 \leq a_k < B$ for $k = 0, 1, \ldots, m-1$.

Proof. By the division algorithm we have

$$\begin{aligned} n = q_0 &= q_1 B + r_1 \\ q_1 &= q_2 B + r_2 \\ &\vdots \\ q_{m-2} &= q_{m-1} B + r_{m-1} \\ q_{m-1} &= q_m B + r_m \\ &\vdots \end{aligned}$$

where $0 \leq r_s < B$ for $s = 0, 1, \ldots$ Whenever $q_k > 0$ we have that $0 \leq q_{k+1} < q_k$. For some $k+1$, say $k+1 = m$, we have $q_m = 0$. Substituting back into the relations above gives us

$$\begin{aligned} n &= ((\ldots(r_m B + r_{m-1})B + r_{m-2})B + r_{m-3})\ldots)B + r_1 \\ &= r_m B^{m-1} + r_{m-1} B^{m-2} + \cdots + r_2 B + r_1 \end{aligned} \tag{2.1}$$

□

The coefficients a_k in the theorem above are called *digits* and B is called the *radix*. The digit a_{m-1} is called the *most significant digit* and the digit a_0 is called the *least significant digit*. For radix $B = 2$ we may equivalently use the word *bit* as a synonym for digit. An integer written on the form $n = a_0 + \cdots + a_{m-1} B^{m-1}$ has *radix representation* B, or *radix* B for short. Furthermore, every integer n in radix representation B has m digits if $n < B^m$ for some integer m.

2.1.1 Addition

Slightly expanding on the previous theorem we could write an integer a as the polynomial $a(x)$ evaluated at point B. That is

$$a = a(B), \quad a(x) = a_0 + a_1 x + \cdots a_{n-1} x^{n-1} \tag{2.2}$$

Addition of two integers a, b would then translate to the addition of the two polynomials

$$a(x) = a_0 + \cdots + a_{n-1} x^{n-1}, \quad b(x) = b_0 + \cdots + b_{n-1} x^{n-1} \tag{2.3}$$

which would result in

$$a(x) + b(x) = (a_0 + b_0) + \cdots + (a_{n-1} + b_{n-1})x^{n-1} \tag{2.4}$$

Two things should be noted here. First, we have assumed a and b to be n-digit integers, because if a is an m-digit integer where $m < n$ we may very well write a as

$$a = a_0 + \cdots + a_{m-1}x^{m-1} + a_m x^m + \cdots a_{n-1}x^{n-1} \tag{2.5}$$

where $a_k = 0, k = m, \ldots, n-1$.

Second, when we have performed the addition in (2.4) we must conceptually make a second "pass" over the resulting coefficients in order to propagate carries to the position $k+1$ whenever $a_k + b_k \geq B$. In a computer implementation this "pass" is of course done as a step when adding the corresponding coefficients. This is known as *carry propagation*. Now as $0 \leq a_k < B$ we have

$$a_k + b_k \leq (B-1) + (B-1) = 1 \cdot B + (B-2) \tag{2.6}$$

which shows that the carry digit is either 0 or 1. To determine the carry we simply need to compare if $a_k + b_k \geq B$ and, if that is the case, add 1 to the next coefficient and reduce the current one with B, i.e. $a_k + b_k - B$.

Example Let us consider a simple example of adding 123 and 789. We assume radix $B = 10$ and associate these integers with the polynomials

$$p(x) = 1x^2 + 2x + 3, \quad q(x) = 7x^2 + 8x + 9 \tag{2.7}$$

Adding $p(x)$ and $q(x)$ results in,

$$p(x) + q(x) = (1+7)x^2 + (2+8)x + (3+9) = 8x^2 + 10x + 12 \tag{2.8}$$

Now propagating carries, from right to left, results in

$$p(x) +_B q(x) = 9x^2 + 1x + 2 \tag{2.9}$$

which evaluated, at point $B = 10$, gives us the expected result of 912 (here $+_B$ denotes addition with carry propagation in base B).

Given two n-digit numbers, we perform n additions, and therefore the complexity order of the addition algorithm is $\mathcal{O}(n)$. For a computer implementation, it is of importance to choose the radix B in such a way that it minimizes the number of digits and that it utilizes the underlying machine hardware in the best way. For example, a computer which has a word size of 2^{16} and is able to perform efficient addition with this word size, would only need one "digit" to represent the numbers 123 and 789 in radix 2^{16}. This would reduce the number of operations to one compared to three in the example above.

A final observation following from the carrying process is that we know how many digits the result from the addition will hold. Let's make this a bit more explicit

$$\begin{aligned} a + b &\leq (B-1) + \dots (B-1)B^{n-1} + (B-1) + \dots (B-1)B^{n-1} \\ &= (B^n - 1) + (B^n - 1) = 2B^n - 2 < B^{n+1} \end{aligned} \tag{2.10}$$

which states that the sum of two n-digit integers will be an $n+1$-digit integer. This is of interest when it comes to computer implementations, as it gives us the memory storage requirements when performing integer addition.

2.1.2 Subtraction

Following the same principle as for addition, with the assumption that $a \geq b$, we have

$$a - b = \sum_{k=0}^{n-1} a_k x^k - \sum_{k=0}^{n-1} b_k x^k = a_0 \ominus b_0 + \cdots + (a_{n-1} \ominus b_{n-1})x^{n-1} \tag{2.11}$$

Care has to be taken when considering the subtraction of digits $a_k \ominus b_k$ as we are exclusively working with non-negative integers. If $a_k \geq b_k$ we perform the subtraction as usual, that is $a_k \ominus b_k = a_k - b_k$. Else we let $a_k \ominus b_k = (B + a_k) - b_k$. Parentheses indicate addition before subtraction to exclude any non-negative integers. We also set b_{k+1} to $b_{k+1} + 1$ to reflect the "borrowing" that has been done. The expression b_{k+1} is well-formed as we have, by assumption, that $a \geq b$, which implies $a_{n-1} \geq b_{n-1}$. In other words, we don't have any "borrowing" procedure for the last digit.

2.1.3 Multiplication

The process of multiplication also take advantage of a radix representation of the operands. We again define our integers, a and b, to have some radix B representation with the help of the polynomials

$$a(x) = \sum_{k=0}^{n-1} a_k x^k, \quad b(x) = \sum_{k=0}^{m-1} b_k x^k, \quad a_k, b_k \in \{0, \dots, B-1\} \tag{2.12}$$

Here $a(x)$ represents an n-digit integer and $b(x)$ represents an m-digit integer. By the definition of polynomial multiplication, we have that

$$c(x) = a(x)b(x) = \sum_{k=0}^{n+m-1} c_k x^k, \quad c_k = \sum_{r=0}^{k} a_r b_{k-r}, \tag{2.13}$$

where $a_i = 0$ for $i \geq n$ and $b_j = 0$ for $j \geq m$. One small observation in (2.13) is that $c_{n+m-1} = 0$ because $a_r = 0$ when $r \geq n$ and $b_{n+m-1-r} = 0$ when $m + n - 1 - r \geq m \Leftrightarrow n - 1 \geq r$. The result $c = ab$ is an $n + m$-digit integer as

$$ab \leq (B^n - 1)(B^m - 1) = B^{n+m} - B^n - B^m + 1 < B^{n+m} \tag{2.14}$$

For the remainder we will consider integers a, b such that they have the same number of digits. Their radix representations will then satisfy

$$\deg a(x) = \deg b(x) = n \tag{2.15}$$

The runtime complexity may be estimated by looking at the number of multiplications performed, as these are assumed to be more expensive compared to the operations of addition. The number of multiplications performed may be found by writing $a(x)b(x)$ as

$$a_0(b_0+b_1x+\cdots+b_{n-1}x^{n-1})+\cdots+a_{n-1}x^{n-1}(b_0+b_1x+\cdots+b_{n-1}x^{n-1}) \tag{2.16}$$

Each $a_k (0 \leq k < n)$ is involved in n multiplications and there are n such a_k. It follows that we perform n^2 multiplications and the runtime complexity is of order $\mathcal{O}(n^2)$.

The carry propagation is a bit different compared to addition and subtraction. It usually involves calculating $c_k \bmod B$ and $\lfloor c_k/B \rfloor$ in order to find the resulting digit and the carry to propagate. This could be done for each multiplication but as the operations of division and modulus is costly, usually at least as costly as multiplication, it will affect the overall performance. A better alternative is to delay the carry propagation until some last separate step. This will put some restrictions on the number of digits that can be handled without overflowing the chosen word size. In this case the carry propagation works by finding new coefficients c'_k for the result by using the following procedure

$$\begin{array}{lcl}
c'_0 & \leftarrow & c_0 \\
c'_1 & \leftarrow & c_1 + \lfloor c'_0/B \rfloor \\
c'_2 & \leftarrow & c_2 + \lfloor c'_1/B \rfloor \\
\vdots & & \\
c'_{2n-2} & \leftarrow & c_{2n-2} + \lfloor c'_{2n-3}/B \rfloor \\
c'_{2n-1} & \leftarrow & c_{2n-1} + \lfloor c'_{2n-2}/B \rfloor = \lfloor c'_{2n-2}/B \rfloor
\end{array}$$

and noting that the result of $a(x)b(x)$ is

$$a(x)b(x) = c'_0 \bmod B + (c'_1 \bmod B)x + \cdots + (c'_{2n-1} \bmod B)x^{2n-1} \tag{2.17}$$

From the above procedure we introduce the following recursive formula

$$\begin{aligned} c'_0 &= c_0 \\ c'_k &= c_k + \lfloor c'_{k-1}/B \rfloor, \quad 1 \leq k \leq 2n-1 \end{aligned} \tag{2.18}$$

Each c_k in (2.13) is maximized when $a_j = b_j = B - 1 (0 \leq j < n)$. In this case we write the coefficients as

$$c_k = \begin{cases} (k+1)(B-1)^2 & 0 \leq k \leq n-1 \\ (2n-(k+1))(B-1)^2 & n \leq k \leq 2n-1 \end{cases} \tag{2.19}$$

Combining (2.18) and (2.19) we find the following closed form

$$c'_k = c_k + \begin{cases} 0 & k = 0 \\ kB - (k+1) & 1 \leq k \leq n \\ (2n-k)(B-1) & n+1 \leq k \leq 2n-1 \end{cases} \tag{2.20}$$

For $k = 0$ the formula (2.20) is true. For $k = 1$ we have by (2.18)

$$\begin{aligned} c'_1 &= c_1 + \lfloor c'_0/B \rfloor = c_1 + \lfloor c_0/B \rfloor = c_1 + \lfloor (B-1)^2/B \rfloor = \\ &= c_1 + B - 2 = c_1 + B - (1+1) \end{aligned} \tag{2.21}$$

which equals (2.20). Now assume that there exists $k \in \{2, \ldots, n\}$ such that

$$c_k + \lfloor c'_{k-1}/B \rfloor \neq c_k + kB - (k+1) \tag{2.22}$$

By the well-ordering principle there is a smallest such k. But then

$$\begin{aligned} c'_k &= c_k + \lfloor c'_{k-1}/B \rfloor = c_k + \lfloor c_{k-1} + kB - (k+1) \rfloor = \\ &= c_k + \lfloor (k(B-1)^2 + (k-1)B - k)/B \rfloor = \\ &= c_k + \lfloor (kB^2 - kB - B)/B \rfloor = c_k + kB - (k+1) \end{aligned} \tag{2.23}$$

which contradicts the existence of such a k. By doing a similar argumentation for the case $k \in \{n+1, n+2, \ldots, 2n-1\}$, we may conclude that (2.20) is equivalent to (2.18). After somewhat tedious calculations, which we will omit, it is possible to show that

$$c'_k \leq c'_{k+1}, \quad k = 0, 1, \ldots, n-2 \tag{2.24}$$
$$c'_k \geq c'_{k+1}, \quad k = n, n+1, \ldots, 2n-2 \tag{2.25}$$
$$c'_n \leq c'_{n-1} \tag{2.26}$$

and thereby concluding that

$$c'_{n-1} \geq c'_k, \quad k \in \{0, 1, \ldots, 2n-1\} \tag{2.27}$$

The value c'_{n-1} may not be larger than the largest value possible for the chosen computer word. Using (2.20) and (2.19) we find that for a binary computer, the number of digits n we may handle have to satisfy the relation

$$n \leq \frac{2^w - 1 + B}{B^2 - B} \tag{2.28}$$

where w is the number of bits in the value representation of the chosen computer word.

Example With a word size of 32-bits the following table shows some possible digit lengths.

B (radix)	n (digits)
2	2147483649
10	47721858
256	65793
1024	4100
4096	256

We end this discussion about classic multiplication with an observation made by Crandall and Pomerance [6]. It suggests that we can do computationally better when it comes to squaring. The coefficients (2.13) can be written as

$$c_k = \sum_{r=0}^{k} a_r a_{k-r} \tag{2.29}$$

For k odd we have an even number of terms which can be written as

$$c_k = 2 \sum_{r=0}^{\lfloor k/2 \rfloor} a_r a_{k-r} \tag{2.30}$$

due to the symmetry. For k even we have an odd number of terms with a "middle" term for $r = k - r$, that is $r = k/2$. The coefficients in (2.13) is now

$$c_k = 2 \sum_{r=0}^{\lfloor k/2 \rfloor} a_r a_{k-r} - a_{k/2}^2 \tag{2.31}$$

Putting these together, we get

$$c_k = 2 \sum_{r=0}^{\lfloor k/2 \rfloor} a_r a_{k-r} - \left\{ \begin{array}{ll} 0 & k \text{ odd} \\ a_{k/2}^2 & k \text{ even} \end{array} \right. \tag{2.32}$$

This is roughly half the work compared to using the general multiplication algorithm with the same operand twice. The operation of exponentiation is common in cryptographic algorithms, and this optimization of squaring is important. We have already seen the need for exponentiation in the introduction where the Diffie-Hellman key exchange was given.

2.2 Modular arithmetic

This section introduces some properties of modular arithmetic, primarily based on the presentation made by Knuth [8].[1]

We may take advantage of modular arithmetic when working with large integers. The idea is to map the ring of integers $\mathbb{Z}_m$ to the direct product ring $\mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_n}, m_k \in \mathbb{Z}^+$ for $k = 1, 2, ..., n$. All the arithmetic is then carried out in this direct product ring. For example to multiply integers a and b, we would first map a and b to $\mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_n}$ which would give us $([a_1]_{m_1}, ..., [a_n]_{m_n})$ and $([b_1]_{m_1}, ..., [b_n]_{m_n})$. To multiply these, we only need to consider component-wise multiplication (modulus some m_k).

One of the advantages of this follows from the fact that if we have $m_k < m$, we may reduce large integers into several smaller ones, which are more easily dealt with. The following theorem gives us the justification for the idea of using a mapping from an integer ring to a direct product ring.

Theorem 2. *Let $m = m_1 m_2 ... m_n$ where $\gcd(m_i, m_j) = 1$ whenever $i \neq j$. Then $\mathbb{Z}_m$ is isomorphic to $\mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_n}$.*

Proof. Define $\Phi : \mathbb{Z}_m \longrightarrow \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_n}$ by $\Phi([x]_m) = ([x]_{m_1}, [x]_{m_2}, ..., [x]_{m_n})$. If $[x]_m = [y]_m$ we have $x - y = qm = qm_1 m_2 \cdots m_n$ for some $q \in \mathbb{Z}$, which implies that $[x]_{m_k} = [y]_{m_k}$ for $k = 1, 2, ..., n$, and so Φ is well-defined.

The function Φ is injective since if $\Phi([x]_m) = \Phi([y]_m)$ we have $[x]_{m_k} = [y]_{m_k}$ for $k = 1, 2, ..., n$, and so $x \equiv y \pmod{m_k}$ for $k = 1, 2, ..., n$. By assumption $\gcd(m_i, m_j) = 1$ whenever $i \neq j$, which implies that $[x]_m = [y]_m$. As Φ is an injective map between two finite sets it follows that Φ is surjective as well. Finally, we have to argue that Φ preserves multiplication

[1]Carl Friedrich Gauss is considered to be the original inventor of modular arithmetic.

and addition

$$\begin{aligned}\Phi([x]_m[y]_m) &= \Phi([xy]_m) = ([xy]_{m_1}, [xy]_{m_2}, \ldots, [xy]_{m_n})\\ &= ([x]_{m_1}[y]_{m_1}, [x]_{m_2}[y]_{m_2}, \ldots, [x]_{m_n}[y]_{m_n})\\ &= ([x]_{m_1}, [x]_{m_2}, \ldots, [x]_{m_n})([y]_{m_1}, [y]_{m_2}, \ldots, [y]_{m_n})\\ &= \Phi([x]_m)\Phi([y]_m)\end{aligned}$$

$$\begin{aligned}\Phi([x]_m + [y]_m) &= \Phi([x+y]_m) = ([x+y]_{m_1}, [x+y]_{m_2}, \ldots, [x+y]_{m_n})\\ &= ([x]_{m_1} + [y]_{m_1}, [x]_{m_2} + [y]_{m_2}, \ldots, [x]_{m_n} + [y]_{m_n})\\ &= ([x]_{m_1}, [x]_{m_2}, \ldots, [x]_{m_n}) + ([y]_{m_1}, [y]_{m_2}, \ldots, [y]_{m_n})\\ &= \Phi([x]_m) + \Phi([y]_m)\end{aligned}$$

$\square$

The range of integers that can be handled by this technique is equal to $m = m_1 m_2 \cdots m_n$. Choosing each m_k close to a computer word size, and having roughly n components in the modular representation we can handle n-digit numbers. This would imply that the time required for addition, subtraction and multiplication is proportional to n, which is of advantage when it comes to compound operations involving multiplication of integers, but not for addition and subtraction.

On the other hand, the method under discussion is very attractive when it comes to computer architectures which allows parallel computing. In such a case, it would be theoretically possible to compute the components all at once for addition, subtraction and multiplication, and this could be done in constant time.

However, this modular representation is not optimal for some properties. There is no easy way of testing which integer of $a = ([a_1]_{m_1}, \ldots, [a_n]_{m_n})$ and $b = ([b_1]_{m_1}, ..., [b_n]_{m_n})$ is the larger one. Division is also difficult to perform. In the modular representation there is no straightforward correlation between the components of different integers.

If we let $0 \leq a_k, b_k < m_k (k = 1, 2, \ldots, n)$ be the representatives (as we may add/subtract multiples of m_k to put them in the desired range) for the components of $a, b \in \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_n}$, the following operations for component-wise addition, subtraction and multiplication may be used

$$a_k + b_k \bmod m_k = a_k + b_k - m_k \wr a_k + b_k \geq m_k \wr \tag{2.33}$$

$$a_k - b_k \bmod m_k = a_k - b_k + m_k \wr a_k < b_k \wr \tag{2.34}$$

$$a_k b_k \bmod m_k = a_k b_k - \lfloor a_k b_k / m_k \rfloor m_k \tag{2.35}$$

Here, the notation $\wr expr \wr$ has the following meaning

$$\wr expr \wr = \begin{cases} 1 & \text{if } expr \text{ is true} \\ 0 & \text{otherwise} \end{cases} \tag{2.36}$$

From a computer's standpoint, we would normally like $m = m_1 \cdots m_n$ to be as large as possible. One way to achieve this is by selecting m_1 as the largest odd integer that will fit in the computers word size, and then choose m_2 as the next odd integer such that $m_2 < m_1$ and $\gcd(m_1, m_2) = 1$, proceeding with this until the desired range has been achieved.

When dealing with binary computers, it is convenient to choose

$$m_k = 2^{t_k} - 1 \tag{2.37}$$

and to require

$$0 \leq a_k < 2^{t_k}, \quad a \equiv a_k \pmod{2^{t_k} - 1} \tag{2.38}$$

In this setup, $a_k = 2^{t_k} - 1$ is another way of saying $a_k = 0$. This will enable the operations of addition, subtraction and multiplication to be defined in terms of modulo 2^{t_k}, instead of modulo $2^{t_k} - 1$. These operations now take the following forms

$$\begin{aligned} a_k \oplus b_k &= a_k + b_k \bmod 2^{t_k} + \wr a_k + b_k \geq 2^{t_k} \wr & (2.39) \\ a_k \ominus b_k &= a_k - b_k \bmod 2^{t_k} - \wr a_k < b_k \wr & (2.40) \\ a_k \otimes b_k &= a_k b_k \bmod 2^{t_k} \oplus \lfloor a_k b_k / 2^{t_k} \rfloor & (2.41) \end{aligned}$$

A verification of the above for $a_k, b_k \in \{1, 2, \cdots, 2^{t_k} - 1\}$ shows that the operations indeed give us the result we expect.

$$\begin{aligned} a_k \oplus b_k &= a_k + b_k \bmod 2^{t_k} + \wr a_k + b_k \geq 2^{t_k} \wr \\ &= a_k + b_k - 2^{t_k} \wr a_k + b_k \geq 2^{t_k} \wr + \wr a_k + b_k \geq 2^{t_k} \wr \\ &= a_k + b_k - (2^{t_k} - 1) \wr a_k + b_k \geq 2^{t_k} \wr \end{aligned} \tag{2.42}$$

As $a_k \oplus b_k = 0$ when $a_k + b_k = 2^{t_k} - 1$ with our convention introduced above it follows that

$$a_k \oplus b_k = a_k + b_k \bmod 2^{t_k} - 1 \tag{2.43}$$

The operator $\ominus$ is verified in a similar way. For the operator $\otimes$ we let $q = \lfloor a_k b_k / 2^{t_k} \rfloor$, so

$$\begin{aligned} a_k \otimes b_k &= a_k b_k \bmod 2^{t_k} \oplus q = a_k b_k - q 2^{t_k} \oplus q \\ &= a_k b_k - q 2^{t_k} + q - 2^{t_k} \wr a_k b_k - q(2^{t_k} - 1) \geq 2^{t_k} \wr + \wr a_k b_k - q(2^{t_k} - 1) \geq 2^{t_k} \wr \\ &= a_k b_k - (2^{t_k} - 1)(q + \wr a_k b_k - q(2^{t_k} - 1) \geq 2^{t_k} \wr) \end{aligned} \tag{2.44}$$

$\underline{a_k b_k - q(2^{t_k}-1) < 2^{t_k}}$
For $a_k b_k - q(2^{t_k}-1) = 2^{t_k}-1$, we have $a_k \otimes b_k = 0 = a_k b_k \bmod 2^{t_k}-1$.
For $a_k b_k - q(2^{t_k}-1) < 2^{t_k}-1$, we have $q \leq \frac{a_k b_k}{2^{t_k}-1} < q+1 \Leftrightarrow \lfloor \frac{a_k b_k}{2^{t_k}-1} \rfloor = q$ and therefore $a_k \otimes b_k = a_k b_k - q(2^{t_k}-1) = a_k b_k - \lfloor \frac{a_k b_k}{2^{t_k}-1} \rfloor (2^{t_k}-1) = a_k b_k \bmod 2^{t_k}-1$.
$\underline{a_k b_k - q(2^{t_k}-1) \geq 2^{t_k}}$
$a_k b_k - q(2^{t_k}-1) \geq 2^{t_k} \Leftrightarrow a_k b_k > 2^{t_k}-1+q(2^{t_k}-1) \Leftrightarrow \frac{a_k b_k}{2^{t_k}-1} > 1+q$.
With the bounds $q \leq 2^{t_k}-2$ and $a_k b_k - q2^{t_k} \leq 2^{t_k}-1$, we get
$a_k b_k - q2^{t_k} + q \leq 2^{t_k}-1+q \leq 2^{t_k}-1+2^{t_k}-2 < 2(2^{t_k}-1) \Leftrightarrow \frac{a_k b_k}{2^{t_k}-1} < q+2$
So $q+1 < \frac{a_k b_k}{2^{t_k}-1} < q+2 \Leftrightarrow \lfloor \frac{a_k b_k}{2^{t_k}-1} \rfloor = q+1$, and therefore we have
$a_k \otimes b_k = a_k b_k - (2^{t_k}-1)(q+1) = a_k b_k - (2^{t_k}-1)\lfloor \frac{a_k b_k}{2^{t_k}-1} \rfloor = a_k b_k \bmod 2^{t_k}-1$

These operations have the advantage of being quite efficient when it comes to binary computers. Arithmetic operations modulo 2^{t_k} is equivalent to the lower t_k digits (bits) in the performed operation and division with 2^{t_k} is equivalent to shifting the digits (bits) t_k positions to the right.

In order to use the mapping to direct product rings, as shown in theorem 2, we need a simple way do determine when 2^r-1 and 2^s-1 are relatively prime.

Lemma 1. *Let $m, n > 0$ be two integers. Then*

$$2^n - 1 \bmod 2^m - 1 = 2^{n \bmod m} - 1 \tag{2.45}$$

Proof. If $n < m$ then $2^n-1 \bmod 2^m-1 = 2^n-1$ and $2^{n \bmod m}-1 = 2^n-1$, so the equality (2.45) holds. For $n \geq m$, let $n = qm+r$ for some $q \in \mathbb{Z}^+$ and $0 \leq r < m$.

$$\begin{gathered}
(2^m-1)(1+2^m+\cdots+(2^m)^{q-1}) = 2^{mq}-1 \\
\Longleftrightarrow \\
2^r(2^m-1)(1+2^m+\cdots+(2^m)^{q-1}) = 2^r(2^{mq}-1) \\
\Longleftrightarrow \\
2^r(2^m-1)(1+2^m+\cdots+(2^m)^{q-1}) = 2^r(2^{mq}-1)+1-1 \\
\Longleftrightarrow \\
2^r(2^m-1)(1+2^m+\cdots+(2^m)^{q-1}) = 2^{mq+r}-1-(2^r-1) \\
\Longleftrightarrow \\
2^n-1 = (2^m-1)[2^r(1+2^m+\cdots+(2^m)^{q-1})] + (2^{n \bmod m}-1)
\end{gathered}$$

As $r = n \bmod m < m$ it follows that $2^{n \bmod m}-1 < 2^m-1$ which gives us,

$$2^n - 1 \bmod 2^m - 1 = 2^{n \bmod m} - 1$$

$\square$

Theorem 3. *Let $m, n > 0$ be two integers. Then*

$$\gcd(2^m - 1, 2^n - 1) = 2^{\gcd(m,n)} - 1 \tag{2.46}$$

Proof. We assume without, loss of generality, that $n \geq m$. The Euclidean algorithm may be used to determine $\gcd(2^m - 1, 2^n - 1)$ by letting $A_0 = 2^n - 1, A_1 = 2^m - 1$ and $A_k = A_{k-2} \bmod A_{k-1}$ and using the known fact from this algorithm that there exists a $t \in \{2, 3, \ldots\}$, for which $A_t = 0$, and therefore

$$\gcd(2^n - 1, 2^m - 1) = A_{t-1}$$

Let $B_0 = n, B_1 = m$ and $B_k = B_{k-2} \bmod B_{k-1}$. The claim is that

$$A_k = 2^{B_k} - 1, \quad k = 2, 3, \ldots, t$$

which is certainly true for $k \in \{2, 3\}$ by the previous lemma. If the formula wouldn't be as claimed, we would have, by the well-ordering principle, a smallest k, such that $A_k \neq 2^{B_k} - 1$. This is a contradiction as $A_k = A_{k-2} \bmod A_{k-1} = 2^{B_{k-2}} - 1 \bmod 2^{B_{k-1}} - 1 = 2^{B_{k-2} \bmod B_{k-1}} - 1 = 2^{B_k} - 1$, once again using the previous lemma. When $A_t = 0$, we have $2^{B_t} - 1 = 0$ if, and only if, $B_t = 0$, so $\gcd(n, m) = B_{t-1}$. Putting this together, we arrive at

$$A_{t-1} = 2^{B_{t-1}} - 1 \Leftrightarrow \gcd(2^n - 1, 2^m - 1) = 2^{gcd(n,m)} - 1$$

$\square$

One immediate consequence of this is that $2^r - 1$ and $2^s - 1$ are relatively prime if and only if r and s is relatively prime.

Let us consider a simple example. By assuming a binary computer with a word size of $w = 2^{32}$ and selecting

$$m_1 = 2^{32} - 1, m_2 = 2^{31} - 1, m_3 = 2^{29} - 1, m_4 = 2^{27} - 1, m_5 = 2^{25} - 1$$

it would be possible to efficiently perform addition, subtraction and multiplication of integers in the range $m_1 \cdots m_5 > 2^{143}$.

The effectiveness of the method under discussion may be dependent on the cost of doing the conversions between $\mathbb{Z}_m$ and $\mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_n}, m_k \in \mathbb{Z}^+$ for $k = 1, 2, ..., n$.[2] Going from $n \in \mathbb{Z}_m$ to the modular representation may be done by dividing n with $m_1, m_2, \ldots, m_n$ and using the remainder. The other way around, going from the modular representation $a \in \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_n}$ to $a' \in \mathbb{Z}_m$, is a bit more complicated. This may be done with the help of the Chinese Remainder Theorem.

[2]Doing multiple operations between conversions is to be prefered, if possible, in order to keep this conversion overhead at a minimum.

Theorem 4. *Let $m_1, m_2, \ldots, m_n$ be positive integers, all pairwise co-prime.*

$$\gcd(m_i, m_j) = 1, \quad \textit{whenever} \quad i \neq j$$

Then the system of congruences

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ x &\equiv a_3 \pmod{m_3} \\ &\vdots \\ x &\equiv a_n \pmod{m_n} \end{aligned} \tag{2.47}$$

has a solution.

If we assume the theorem without proof, we need to find an integer a' such that $a' \equiv a_1 \pmod{m_1}, \ldots, a' \equiv a_n \pmod{m_n}$. The theorem gives us no help with this, but to assert the existence of such an integer. It is useless when it comes to actually finding it. Naively, we would have to try

$$a' = s, s+1, \ldots \quad (\text{for some given } s \in \mathbb{Z}^+)$$

until we find a value for which

$$\begin{aligned} a' &\equiv a_1 \pmod{m_1} \\ &\vdots && (2.48) \\ a' &\equiv a_n \pmod{m_n} && (2.49) \end{aligned}$$

A bit more help is given by considering a constructive proof of the Chinese Remainder Theorem

Proof of theorem 4. Let $m = m_1 m_2 \cdots m_n$. For $1 \leq j \leq n$ we have

$$\gcd(m_j, m/m_j) = 1$$

By the Euclidean algorithm we conclude that

$$r_j m_j + s_j m/m_j = 1$$

for some integers r_j and s_j. Let

$$x = \sum_{i=1}^{n} a_i m/m_i$$

Choosing q arbitrarily, such that $1 \leq q \leq n$, we find that

$$\begin{aligned} x - a_q &= a_1 s_1 m/m_1 + \cdots + a_q s_q m/m_q + \cdots + a_n s_n m/m_n - a_q = \\ &= a_1 s_1 m/m_1 + \cdots + a_q(s_q m/m_q - 1) + \cdots + a_n s_n m/m_n = \\ &= a_1 s_1 m/m_1 + \cdots + a_q(-r_q m_q) + \cdots + a_n s_n m/m_n = \\ &= m_q(a_1 s_1 m/m_1 m_q + \cdots + a_q(-r_q) + \cdots + a_n s_n m/m_n m_q) \end{aligned}$$

This shows that

$$x \equiv a_q \pmod{m_q}$$

and it follows that x is a solution to the system of congruences

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ x &\equiv a_3 \pmod{m_3} \\ &\vdots \\ x &\equiv a_n \pmod{m_n} \end{aligned} \tag{2.50}$$

□

However, calculating a solution x, now requires us to do multiplication of large numbers m/m_i. This is what we wanted to avoid by introducing modular arithmetic in the first place. A better method, using $\binom{n}{2}$ constants, c_{jk}, where

$$c_{jk} m_j \equiv 1 \pmod{m_k}, \tag{2.51}$$

where $1 \leq j < k \leq n$, was suggested by Garner in 1958. The constants c_{jk} are computed by using the extended Euclidean algorithm which for any given j and k computes the relation

$$u m_j + v m_k = \gcd(m_j, m_k) = 1 \tag{2.52}$$

and we choose $c_{jk} = u$. For these c_{jk} we may do the following

$$\begin{aligned} &\text{let } b_1 = a_1 \bmod m_1 \\ &\text{let } b_2 = (a_2 - b_1)c_{12} \bmod m_2 \\ &\text{let } b_3 = ((a_3 - b_1)c_{13} - b_2)c_{23} \bmod m_3 \\ &\quad \vdots \\ &\text{let } b_n = (\ldots((a_n - b_1)c_{1n} - b_2)c_{2_n} - \cdots - v_{n-1})c_{(n-1)n} \bmod m_n \end{aligned} \tag{2.53}$$

Then, the integer

$$a' = b_n m_{n-1} \cdots m_2 m_1 + \cdots + b_3 m_2 m_1 + b_2 m_1 + b_1 \tag{2.54}$$

satisfy the conditions

$$0 \le a' < m, \quad a' \equiv a_k \pmod{m_k}, \quad 1 \le k \le n \tag{2.55}$$

The relation in (2.54) is called *mixed-radix representation* of a'. Methods for converting this mixed-radix representation to binary or decimal representation is presented in Knuth [8], but we will not go into the details here.

One last observation regarding these conversions is that we may show that if $p \bmod q = d$ and $pc \bmod q = 1$, then the relation

$$(1 + 2^d + 2^{2d} + \cdots + 2^{(c-1)d})(2^p - 1) \equiv 1 \pmod{2^q - 1} \tag{2.56}$$

holds. As our moduli m_k have the special form as given in (2.37), we have a relatively simple relation for the calculation of the constants c_{jk} in (2.51).

2.3 Karatsuba multiplication

In 1960 A. Karatsuba [7] discovered an algorithm to multiply integers which uses fewer multiplication operations than the classical “naive” multiplication algorithm discussed previously in (2.1.3).

If we let x and y denote two n-digit integers, we may write these as

$$x = x_1 B^m + x_0 \text{ and } y = y_1 B^m + y_0 \tag{2.57}$$

for some radix B and $m < n$. B is chosen to be a value that is easily shifted. For example, a modern binary computer could have $B = 2$. The integer m is usually chosen to be near $n/2$.

From the above, it follows that

$$xy = (x_1 B^m + x_0)(y_1 B^m + y_0) = x_1 y_1 B^{2m} + (x_1 y_0 + x_0 y_1) B^m + x_0 y_0 \tag{2.58}$$

Normally, xy requires four multiplications to calculate the four subproducts x_1y_1, x_1y_0, x_0y_1 and x_0y_0, thereby requiring xy to be calculated with four multiplications and some additions and shifting. We consider the time taken for addition, subtraction and shifting to be negligible compared to the time taken for the operation of multiplication. In total, this suggests that we haven't come up with something better regarding computational time than the classical multiplication algorithm.

The key observation to make here is that if we let

$$\begin{aligned} U &= x_1 y_1 && (2.59)\\ V &= x_0 y_0 && (2.60)\\ W &= (x_1 + x_0)(y_1 + y_0) - U - V && (2.61) \end{aligned}$$

it follows that $xy = UB^{2m} + WB^m + V$ as we have

$$W = (x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0) - x_1y_1 - x_0y_0 = x_1y_0 + x_0y_1 \tag{2.62}$$

Calculating U, V, W now requires three multiplications and some additions and subtractions, saving one operation of multiplication compared to the classical multiplication algorithm. Furthermore, we may apply the method above recursively to calculate the values of U, V and W. This makes it possible to handle integers of arbitrary length. We split the integers involved into smaller units until we can effectively compute their products.

As a simple example, we may assume a decimal computer with $B = 10$, capable of multiplying 3-digit integers. In order to multiply 1234 by 5678 we choose $m = 2$ and write these integers as

$$x = 12 \cdot 10^2 + 34 \quad \text{and} \quad y = 56 \cdot 10^2 + 78$$

Now let

$$\begin{aligned} U &= 12 \cdot 56 = 672 \\ V &= 34 \cdot 78 = 2652 \\ W &= (12 + 34)(56 + 78) - 672 - 2652 = 2840 \end{aligned}$$

We then have

$$\begin{aligned} 1234 \cdot 5678 &= 672 \cdot 10^4 + 2840 \cdot 10^2 + 2652 = \\ &= 6720000 + 284000 + 2652 = 7006652 \end{aligned}$$

In our decimal computer, the evaluation of $672 \cdot 10^4$ and $2840 \cdot 10^2$ corresponds to simple shifts and is therefore considered cheap with respect to computing time. If our decimal computer had only been capable of multiplying 1-digit integers, we would have applied the method recursively to calculate the values of $12 \cdot 56, 34 \cdot 78$ and $(12 + 34)(56 + 78)$. To make the recursion effective it is usually a good idea to have the number of digits equal to some power of two, that is $n = 2^k$ for some integer $k > 0$.

The complexity analysis of the Karatsuba algorithm performed in Knuth [8] is given here in a somewhat expanded form. Let $T(n)$ denote the time it takes to multiply n-digit integers. It follows from the discussion above that we have

$$T(2n) \leq 3T(n) + cn \tag{2.63}$$

for some constant c, as we have three multiplications with some shifts, additions and subtractions. Now consider the inequality

$$T(2^k) \leq c(3^k - 2^k) \tag{2.64}$$

For $k = 1$, we have $T(2) \leq c(3-2) = c$, so we may choose c such that $T(2) \leq c$ holds. Let us now assume that

$$T(2^k) \leq c(3^k - 2^k) \tag{2.65}$$

holds for $k \geq 1$ with the above chosen c.

If we combine $T(2^{k+1}) = T(2 \cdot 2^k)$ with (2.63) it follows that

$$T(2 \cdot 2^k) \leq 3T(2^k) + c2^k \tag{2.66}$$

By assumption (2.65), we have $T(2^k) \leq c(3^k - 2^k)$, which gives us

$$3T(2^k)+c2^k \leq 3(c(3^k-2^k))+c2^k = c3^{k+1}-3c2^k+c2^k = c(3^{k+1}-2^{k+1}) \tag{2.67}$$

This shows that

$$T(2^m) \leq c(3^m - 2^m), \quad m \geq 1 \tag{2.68}$$

holds for some c.

We may now derive an upper bound for $T(n)$ as follows

$$T(n) = T(2^{\log_2 n}) \leq T(2^{\lceil \log_2 n \rceil}) \tag{2.69}$$

From (2.68), in combination with (2.69), we may conclude that $T(2^{\lceil \log_2 n \rceil}) \leq c(3^{\lceil \log_2 n \rceil} - 2^{\lceil \log_2 n \rceil})$ for some c. By the definition of the ceiling function, $\lceil x \rceil$, we have $\lceil \log_2 n \rceil < \log_2 n + 1$ which implies $3^{\lceil \log_2 n \rceil} < 3^{\log_2 n + 1}$. It follows that

$$T(n) \leq c(3^{\lceil \log_2 n \rceil} - 2^{\lceil \log_2 n \rceil}) < 3c \cdot 3^{\log_2 n} = 3cn^{\log_2 3} \tag{2.70}$$

Relation (2.70) shows that the time used for multiplication with the Karatsuba method is in the order of $n^{\log_2 3} \approx n^{1.585}$, which makes this algorithm substantially faster than the classical multiplication algorithm, when n is large.

2.4 Toom-Cook multiplication

We will briefly mention the Toom-Cook algorithm, of which the Karatsuba algorithm is a special case. The Toom-Cook algorithm depends on the idea that given two polynomials

$$p(x) = p_0 + p_1 x + \cdots + p^{n-1} \tag{2.71}$$

$$q(x) = q_0 + q_1 x + \cdots + q^{n-1} \tag{2.72}$$

the polynomial product $r(x) = p(x)q(x)$ is determined by evaluating $r(x)$ at $2n - 1$ distinct points. Given two integers, p and q, we select a radix B such that the radix representations of p and q, given by the polynomials $p(x)$ and $q(x)$, contains at most n digits. The name Toom-n is a synonym of the Toom-Cook algorithm and thus reflects the number of digits it operates on. For example, the Toom-3 variant for an integer m has the radix representation

$$a(x) = a_0 + a_1 x + a_2 x^2 \tag{2.73}$$

for some radix B such that $m = a(B)$.

Toom-n multiplication of integers p and q may be performed by using the following steps

Initialize Form the polynomials $p(x)$ and $q(x)$ each of degree $n - 1$ such that $p(x)$ and $q(x)$ is the radix representation of p and q in some radix B. That is $p = p(B)$ and $q = q(B)$.

Evaluate Evaluate $r(x) = p(x)q(x)$ at $2n - 1$ distinct points.

Interpolation Find the coefficients of $r(x)$.

Reconstruction Find the final result by evaluating $r(B)$.

When evaluating a polynomial, we make use of a special evaluation point, the point of infinity.

Definition 2. *The evaluation of a polynomial, $p(x) = a_0 + \cdots + a_{n-1}x^{n-1} \in \mathbb{R}[x]$, at the point of infinity is defined as*

$$p(\infty) := \lim_{x\to\infty} \frac{p(x)}{x^{\deg p(x)}} = a_{n-1}$$

The explanation of the Toom-Cook algorithm below is based on a simple example, where we will multiply the integers 123456 and 654321, using Toom-3. The selection of $B = 100$ as the radix gives us the following radix representations

$$p(x) = 12x^2 + 34x + 56 \tag{2.74}$$
$$q(x) = 65x^2 + 43x + 21 \tag{2.75}$$

In order to determine the polynomial $r(x) = p(x)q(x)$, we need to evaluate $r(x)$ at five distinct points

$$r(0) = p(0)q(0) = 1176 \tag{2.76}$$
$$r(1) = p(1)q(1) = 13158 \tag{2.77}$$
$$r(-1) = p(-1)q(-1) = 1462 \tag{2.78}$$
$$r(-2) = p(-2)q(-2) = 7020 \tag{2.79}$$
$$r(\infty) = p(\infty)q(\infty) = 780 \tag{2.80}$$

To find the polynomial $r(x) = r_0 + r_1x + \cdots + r_4x^4$, we need to solve the equation

$$\begin{pmatrix} r(0) \\ r(1) \\ r(-1) \\ r(-2) \\ r(\infty) \end{pmatrix} = \begin{pmatrix} 1 & 0^1 & 0^2 & 0^3 & 0^4 \\ 1 & 1^1 & 1^2 & 1^3 & 1^4 \\ 1 & (-1)^1 & (-1)^2 & (-1)^3 & (-1)^4 \\ 1 & (-2)^1 & (-2)^2 & (-2)^3 & (-2)^4 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix} \tag{2.81}$$

Solving such a system could be done using Gaussian elimination or, if the evaluation points have been chosen suitably, by multiplication of the inverse matrix.

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{3} & -1 & \frac{1}{6} & -2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ -\frac{1}{2} & \frac{1}{6} & \frac{1}{2} & -\frac{1}{6} & 2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r(0) \\ r(1) \\ r(-1) \\ r(-2) \\ r(\infty) \end{pmatrix} \tag{2.82}$$

Even though the matrix contain fractions the resulting coefficients of the polynomial $r(x)$ are integers. This means that we may perform the calculations using additions, subtractions and multiplication/division by small constants.

One way of effectively calculate the above, as suggested by Bodrato [12], is to use the following sequence of operations, which we apply to our example

$$\begin{aligned}
r_0 &\leftarrow r(0) && = 1176 \\
r_4 &\leftarrow r(\infty) && = 780 \\
r_3 &\leftarrow \tfrac{r(-2)-r(1)}{3} && = -2046 \\
r_1 &\leftarrow \tfrac{r(1)-r(-1)}{2} && = 5848 \\
r_2 &\leftarrow r(-1) - r(0) && = 286 \\
r_3 &\leftarrow \tfrac{r_2-r_3}{2} + 2r(\infty) && = 2726 \\
r_2 &\leftarrow r_2 + r_1 - r(\infty) && = 5354 \\
r_1 &\leftarrow r_1 - r_3 && = 3122
\end{aligned}$$

The resulting polynomial is then given by

$$r(x) = 1176 + 3122x + 5354x^2 + 2726x^3 + 780x^4 \tag{2.83}$$

Note that the matrix used only depends on the degree of the resulting polynomial and the evaluation points. For different inputs, the inverse of the matrix does not need to be re-calculated, as long as these parameters are fixed.

In order to reconstruct the result of the multiplication, we evaluate $r(B)$.

$$\begin{aligned} r(B) &= 1176 + 3122 \cdot 100 + 5354 \cdot 100^2 + 2726 \cdot 100^3 + 780 \cdot 100^4 = \\ &= 1176 + 312200 + 53540000 + 2726000000 + 78000000000 = \\ &= 80779853376 = 123456 \cdot 654321 \end{aligned} \tag{2.84}$$

Normally, one chooses B so that the evaluation of $r(B)$ is effective with respect to an implementation. In our example, we have used $B = 100$, meaning that we just left-shift each coefficient zero, two, four, six or eight steps.

The Toom-3 algorithm reduces the number of multiplications from 9 to 5. It can be shown that the time used for multiplication is proportional to $n^{\frac{\log_2 5}{\log_2 3}} \approx n^{1.465}$. In general, Toom-$k$ executes in a time proportional to $c(k)n^m$, where $m = \log_2(2k-1)/\log_2 k$, n^m is the time used for sub-multiplications and c is the time spent on additions and multiplications by small constants. By using larger k, one may lower the value n^m, but unfortunately the function c grows so rapidly when $k \to \infty$ that we would need extremely huge values of n in order to have any significant improvement over Karatsuba multiplication. An improvement was suggested by Toom. It is possible to obtain better results by letting k vary with n and by choosing larger and larger values of k, as n increases. For a more detailed exposition of the Toom-Cook algorithm, the reader is directed to Knuth [8] and Crandall and Pomerance [6].

2.5 Fourier transform multiplication

The most efficient algorithms to multiply large integers are based on the Fast Fourier Transform (FFT), which in turn is just an effective way of computing the Discrete Fourier Transform (DFT). Multiplication of large integers exploit the intimate relationship between the DFT and what later is defined as the *discrete cyclic convolution.*[3]

2.5.1 The Discrete Fourier Transform and convolutions

Let us, as a starting point, define the *complex Discrete Fourier Transform.*

Definition 3. *Let $N > 1$ be an integer, and let $x = (x_0, x_1, \ldots, x_{N-1})$, where $x_k \in \mathbb{C}$ for $k = 0, \ldots, N-1$ be a vector. Then the complex Discrete Fourier*

[3]The DFT multiplication is an example of Toom-Cook multiplication.

Transform is defined as the vector

$$y = (y_0, y_1, \ldots, y_{N-1}), \quad (y)_k = y_k = \sum_{n=0}^{N-1} x_n \omega^{nk}, \quad k = 0, \ldots, N-1 \tag{2.85}$$

where $\omega = e^{\frac{2\pi i}{N}}$.

If we now let

$$A_\omega^N = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)^2} \end{pmatrix} \tag{2.86}$$

we may write the transformation in the definition above as the matrix multiplication

$$\mathcal{F}(x) = A_\omega^N x^T = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)^2} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \cdots \\ x_{N-1} \end{pmatrix} \tag{2.87}$$

Furthermore, if we consider the expression

$$\frac{1}{N} A_{\omega^{-1}}^N A_\omega^N = \frac{1}{N} \begin{pmatrix} 1 & \cdots & 1 \\ 1 & \cdots & \omega^{-(N-1)} \\ \cdots & \cdots & \cdots \\ 1 & \cdots & \omega^{-(N-1)^2} \end{pmatrix} \begin{pmatrix} 1 & \cdots & 1 \\ 1 & \cdots & \omega^{(N-1)} \\ \cdots & \cdots & \cdots \\ 1 & \cdots & \omega^{(N-1)^2} \end{pmatrix} \tag{2.88}$$

and number the rows and columns so that the element at $0, 0$ is at the top left in the resulting matrix, we have in the position r, s of this matrix

$$\frac{1}{N} \sum_{k=0}^{N-1} \omega^{rk} \omega^{-sk} = \frac{1}{N} \sum_{k=0}^{N-1} \omega^{k(r-s)} = \begin{cases} 1 & r = s \\ 0 & r \neq s \end{cases} \tag{2.89}$$

This is evidently the identity matrix. The sum in (2.89) above is zero for $r \neq s$ as it is the geometric series

$$\frac{1}{N} \sum_{k=0}^{N-1} \omega^{k(r-s)} = \frac{1}{N} \frac{(\omega^{(r-s)})^N - 1}{\omega^{r-s} - 1} = 0 \tag{2.90}$$

Note that we generally have $\omega^m = 1 (m \in \mathbb{Z})$ if, and only if, $N|m$. From this, we see that the numerator is zero in (2.90). Remembering that $r \neq s$, we have that $r - s \in \{\pm 1, \pm 2, \ldots, \pm(N-1)\}$, which implies that $N \nmid (r-s)$, and therefore the denominator in (2.90) is non-zero, so the expression is well-formed and has the value asserted. We may conclude that the inverse transformation is given by

$$\mathcal{F}^{-1}(y) = A^N_{\omega^{-1}} y^T, \quad y = (y_0, y_1, \ldots, y_{N-1}) \tag{2.91}$$

The key element of using DFTs as a means of multiplying large integers is the discrete cyclic convolution and its relationship to the DFT. We will define this term and arrive at what is known as the *convolution theorem*, which makes this relationship explicit.

Definition 4. *Let $N > 1$ be an integer, and let*

$$x = (x_0, x_1, \ldots, x_{N-1}), \quad y = (y_0, y_1, \ldots, y_{N-1})$$

be vectors with $x_k, y_k \in \mathbb{C}$ for $k = 0, \ldots, N-1$. We define the discrete cyclic convolution $x * y$ of the vectors x and y to be the vector

$$z = x * y = (z_0, z_1, \ldots, z_{N-1}) \tag{2.92}$$

where

$$(z)_k = z_k = \sum_{r=0}^{N-1} x_r y_{k-r \bmod N}, \quad k = 0, \ldots, N-1 \tag{2.93}$$

Definition 5. *Let $N > 1$ be an integer and let*

$$x = (x_0, x_1, \ldots, x_{N-1}), \quad y = (y_0, y_1, \ldots, y_{N-1})$$

be vectors with $x_k, y_k \in \mathbb{C}$ for $k = 0, \ldots, N-1$. We define the component-wise product $x \odot y$ of the vectors x and y to be the vector

$$z = x \odot y = (z_0, z_1, \ldots, z_{N-1}), \quad (z)_k = z_k = x_k y_k, \quad k = 0, \ldots, N-1 \tag{2.94}$$

Theorem 5 (Convolution theorem). *Let $N > 1$ be an integer and let*

$$x = (x_0, x_1, \ldots, x_{N-1}), \quad y = (y_0, y_1, \ldots, y_{N-1})$$

be vectors with $x_k, y_k \in \mathbb{C}$ for $k = 0, \ldots, N-1$. Then

$$\mathcal{F}(x * y) = \mathcal{F}(x) \odot \mathcal{F}(y) \tag{2.95}$$

Proof. Choose a k such that $k \in \{0, 1, \ldots, N-1\}$

$$\begin{aligned}
(\mathcal{F}(x * y))_k &= \sum_{r=0}^{N-1} \omega^{rk} (x * y)_r = \sum_{r=0}^{N-1} \omega^{rk} \sum_{s=0}^{N-1} x_s y_{r-s \bmod N} \\
&= \sum_{s=0}^{N-1} x_s \sum_{r=0}^{N-1} \omega^{rk} y_{r-s \bmod N} = \\
& x_0(y_0\omega^{0k} + y_1\omega^{k} + y_2\omega^{2k} + \cdots + y_{N-1}\omega^{(N-1)k}) + \\
& x_1(y_{N-1}\omega^{0k} + y_0\omega^{k} + y_1\omega^{2k} + \cdots + y_{N-2}\omega^{(N-1)k}) + \\
& x_2(y_{N-2}\omega^{0k} + y_{N-1}\omega^{k} + y_0\omega^{2k} + \cdots + y_{N-3}\omega^{(N-1)k}) + \\
& \vdots \\
& x_{N-1}(y_1\omega^{0k} + y_2\omega^{k} + y_3\omega^{2k} + \cdots + y_0\omega^{(N-1)k}) = \ldots
\end{aligned}$$

Rearranging and using the fact that $\omega^n = \omega^{n \bmod N}$ for $n \in \mathbb{Z}$ we have

$$\begin{aligned}
\ldots =& x_0(y_0\omega^{(0+0)k} + y_1\omega^{(0+1)}k + \cdots + y_{N-2}\omega^{(0+N-2)k} + y_{N-1}\omega^{(0+N-1)k}) \quad + \\
& x_1(y_0\omega^{(1+0)k} + y_1\omega^{(1+1)k} + \cdots + y_{N-2}\omega^{(1+N-2)k} + y_{N-1}\omega^{(1+N-1)k}) \quad + \\
& x_2(y_0\omega^{(2+0)k} + y_1\omega^{(2+1)k} + \cdots + y_{N-2}\omega^{(2+N-2)k} + y_{N-1}\omega^{(2+N-1)k}) \quad + \\
& \vdots \\
& x_{N-1}(y_0\omega^{(N-1+0)k} + y_1\omega^{(N-1+1)k} + \cdots + y_{N-1}\omega^{(N-1+N-1)k}) = \\
=& \sum_{s=0}^{N-1} x_s \sum_{r=0}^{N-1} y_r \omega^{(s+r)k} = \sum_{s=0}^{N-1} x_s \omega^{sk} \sum_{r=0}^{N-1} y_r \omega^{rk} = (\mathcal{F}(x) \odot \mathcal{F}(y))_k
\end{aligned}$$

As k was arbitrarily chosen from $\{0, 1, \ldots, N-1\}$, it follows that

$$(\mathcal{F}(x * y))_k = (\mathcal{F}(x) \odot \mathcal{F}(y))_k, \quad \forall k \in \{0, 1, \ldots, N-1\}$$

which implies

$$\mathcal{F}(x * y) = \mathcal{F}(x) \odot \mathcal{F}(y)$$

□

From (2.91) and the theorem above it immediately follows that

$$x * y = \mathcal{F}^{-1}(\mathcal{F}(x) \odot \mathcal{F}(y)) \tag{2.96}$$

The calculating of the discrete cyclic convolution of two vectors x, y may now proceed by first determining the DFT of x and y, followed by a component-wise multiplication and finally applying an inverse DFT. This will be of great importance as we now move on to multiplication of polynomials.

Let us consider two polynomials $a(x), b(x)$, where $\deg a(x) = n-1$, and $\deg b(x) = m-1$ respectively. That is

$$a(x) = \sum_{k=0}^{n-1} a_k x^k, \quad b(x) = \sum_{k=0}^{m-1} b_k x^k, \quad a_k, b_k \in \mathbb{C} \tag{2.97}$$

Multiplying $a(x)$ by $b(x)$ gives us a new polynomial $c(x)$

$$c(x) = \sum_{k=0}^{n+m-2} c_k x^k = \sum_{k=0}^{n-1} a_k x^k \sum_{k=0}^{m-1} b_k x^k \tag{2.98}$$

The coefficients c_k of the polynomial $c(x)$ may be derived by using the discrete cyclic convolution. Let $N = \deg a(x) + \deg b(x) + 1$ and represent the coefficients of the polynomials $a(x)$ and $b(x)$ by the following two vectors of length N

$$\hat{a} = (a_0, a_1, \ldots, a_{n-1}, 0, \ldots, 0), \quad \hat{b} = (b_0, b_1, \ldots, b_{m-1}, 0, \ldots, 0) \tag{2.99}$$

by "expanding" the coefficient vectors for $a(x), b(x)$ with zero's, so they have length N. Calculating the discrete cyclic convolution $\hat{c} = \hat{a} * \hat{b}$ by using definition (4) gives us

$$\begin{aligned}
(\hat{c})_0 &= a_0 b_0 \\
(\hat{c})_1 &= a_0 b_1 + a_1 b_0 \\
(\hat{c})_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
(\hat{c})_3 &= a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0 \\
&\vdots \\
(\hat{c})_{n-1} &= a_0 b_{n-1} + a_1 b_{n-2} + \cdots + a_{n-1} b_0 \\
&\vdots \\
(\hat{c})_{n+m-3} &= a_{n-2} b_{m-1} + a_{n-1} b_{m-2} \\
(\hat{c})_{n+m-2} &= a_{n-1} b_{m-1}
\end{aligned} \tag{2.100}$$

This is exactly those coefficients that we would have got from (2.98) for the polynomial $c(x)$, using the usual definition of polynomial multiplication. To be a little more explicit, we have that

$$c_k = (\hat{c})_k, \quad k = 0, 1, \ldots, n+m-2 \tag{2.101}$$

As the discrete cyclic convolution can be expressed in terms of the DFT (see 2.96), we have a method for multiplication of polynomials, and given that carry propagation is handled properly, we also have a method for multiplying integers. Assume that $a(x)$ and $b(x)$ are the polynomials introduced previously and that $c(x) = a(x)b(x)$. We find the coefficients of $c(x)$ by doing the following,

1. Let $N = \deg a(x) + \deg b(x) + 1$.
2. Let $\hat{a} = (a_0, a_1, \ldots, a_{N-1})$ and $\hat{b} = (b_0, b_1, \ldots, b_{N-1})$, where a_r are the coefficients for the polynomial $a(x)$ with $a_r = 0$ for $r > \deg a(x)$ and b_s are the coefficients for the polynomial $b(x)$ with $b_s = 0$ for $s > \deg b(x)$.
3. Calculate the DFTs $\mathcal{F}(\hat{a})$ and $\mathcal{F}(\hat{b})$.
4. Calculate the component-wise product $\mathcal{F}(\hat{a}) \odot \mathcal{F}(\hat{b})$
5. Calculate the inverse DFT $\hat{c} = \mathcal{F}^{-1}(\mathcal{F}(\hat{a}) \odot \mathcal{F}(\hat{b}))$.
6. Each coefficient c_k for the polynomial $c(x)$ is now given by $c_k = (\hat{c})_k$.
7. Handle carry propagation.

The complexity for the method above is bounded by the complexity of the DFT. Using the definition (3) naively will give a complexity of $\mathcal{O}(N^2)$, where N is the vector length given to the DFT.

If the vector $x = (x_0, \ldots, x_{N-1})$, all have real components, that is, if $Im(x_k) = 0$ for $k = 0, \ldots, N-1$, we have the following usable relation

$$(\mathcal{F}(x))_k = \overline{(\mathcal{F}(x))}_{N-k \bmod N} \tag{2.102}$$

which requires roughly half of the necessary computations.

2.5.2 The Fast Fourier Transform

We go on to the possibility of computing the DFT more efficiently, using a FFT algorithm. There are numerous algorithms for implementing a FFT. One of the more popular is an algorithm known as the Cooley-Tukey algorithm [10], named after J.W. Cooley and John Tukey who described it in a paper from 1965 [5]. This algorithm uses a divide and conquer strategy to divide a DFT transformation of size $N = N_1 N_2$ into two interleaved transformations of size N_1 and N_2, which may be further recursively transformed. This decomposition of the transformation can be traced back to work of Carl Friedrich Gauss, who used it for trajectory calculations in astronomy. Cooley and Tukey adapted the algorithm for efficient computer implementation as we will see later.

N needs to be highly composite for the algorithm to be as efficient as possible. This will ensure that the base cases in the recursion are small prime numbers. We will restrict ourselves to transformation sizes of $N = 2^n$ for some $n \in \mathbb{Z}^+$. In practice, this restriction is of no great concern as the application usually has some freedom to choose the size of the input vector for the transformation at hand, by padding the vector with zero's.

Let $N = 2^n$ for some $n \in \mathbb{Z}^+$ and let $x \in \mathbb{C}^N$. By the definition of the DFT, we have for $k = 0, \dots, N-1$

$$\begin{aligned}(\mathcal{F}(x))_k = \sum_{m=0}^{N-1} x_m \omega^{mk} &= [\text{odd and even components}] = \\ &= \sum_{m=0}^{N/2-1} x_{2m}\omega^{2mk} + \sum_{m=0}^{N/2-1} x_{2m+1}\omega^{(2m+1)k} = \\ &= \sum_{m=0}^{N/2-1} x_{2m}\omega^{2mk} + \omega^k \sum_{m=0}^{N/2-1} x_{2m+1}\omega^{2mk}\end{aligned} \tag{2.103}$$

Let the vector a contain the components from x with even indices. That is $a = (x_0, x_2, \dots\dots, x_{N-2})$. Similarly, we let $b = (x_1, x_3, \dots, x_{N-1})$ contain the components of x with odd indices. Furthermore, we have

$$\omega^k = \omega^{qN/2+r} = (-1)^q\omega^r, \quad q \in \{0,1\}, \quad r = 0, \dots, N/2-1 \tag{2.104}$$

which may be written as

$$\omega^k = (-1)^{\wr k\geq N/2\wr}\omega^{k \bmod N/2}, \quad k = 0, \dots, N-1 \tag{2.105}$$

The sums in (2.103) both have a term ω^{2mk}

$$\begin{aligned}\omega^{2mk} &= (\omega^k)^{2m} = ((-1)^{\wr k\geq N/2\wr}\omega^{k \bmod N/2})^{2m} = \\ &= \omega^{2m(k \bmod N/2)} \quad (k = 0, \dots, N-1)\end{aligned} \tag{2.106}$$

We also note that

$$\omega^2 = (e^{\frac{2\pi i}{N}})^2 = e^{\frac{2\pi i}{N/2}} \tag{2.107}$$

Putting all this together we arrive at

$$\begin{aligned}&\sum_{m=0}^{N/2-1} x_{2m}\omega^{2mk} + \omega^k \sum_{m=0}^{N/2-1} x_{2m+1}\omega^{2mk} = \\ &\sum_{m=0}^{N/2-1} a_m(\omega^2)^{m(k \bmod N/2)} + (-1)^{\wr k\geq N/2\wr}\omega^{k \bmod N/2}\sum_{m=0}^{N/2-1} b_m(\omega^2)^{m(k \bmod N/2)} = \\ &(\mathcal{F}(a))_{k \bmod N/2} + (-1)^{\wr k\geq N/2\wr}\omega^{k \bmod N/2}(\mathcal{F}(b))_{k \bmod N/2}\end{aligned} \tag{2.108}$$

Rewriting (2.108) in a somewhat different form gives us

$$\begin{aligned}(\mathcal{F}(x))_k &= (\mathcal{F}(a))_k + \omega^k(\mathcal{F}(b))_k \quad (0 \le k < N/2)\\ (\mathcal{F}(x))_{k+N/2} &= (\mathcal{F}(a))_k - \omega^k(\mathcal{F}(b))_k \quad (0 \le k < N/2)\end{aligned} \tag{2.109}$$

In order to compute $\mathcal{F}(x)$ we first need to compute $\mathcal{F}(a)$ and $\mathcal{F}(b)$, which are both DFTs of length $N/2$, and then to combine these two transformations to get $\mathcal{F}(x)$. In other words, we have re-expressed a DFT of length N in two DFTs of length $N/2$. This process can then be applied recursively for $\mathcal{F}(a)$ and $\mathcal{F}(b)$.

If we denote $T(N)$ as the runtime of this algorithm, we have

$$T(N) = 2T(N/2) + cN \quad \text{for some constant } c \tag{2.110}$$

and by the *Master theorem* [11], we conclude

$$T(N) = N \log N \tag{2.111}$$

The algorithm described so far is of recursive nature, and a typical implementation would need to allocate temporary storage for the results from the two interleaved DFTs and then combine them. What Cooley and Tukey observed in their paper was that if the components of the input vector $x = (x_0, \ldots, x_{N-1})$ is arranged in a certain way, the algorithm can be carried out non-recursive, in-order and in-place [6]. In-order means in natural order of a DFT. In-place means that there is no need for temporary storage as it replaces the values in the input vector with the result from the DFT. This is a very attractive property when it comes to computer implementations, which is one of the reasons for its popularity.

Given a vector $x = (x_0, \ldots, x_{N-1})$, where $N = 2^m$ for some $m \in \mathbb{Z}^+$, the arrangement of a new vector x' is the permutation which takes the component at index i in x to index i', where i' is the bit-reversal of i when i is considered in base 2. As an example, if $N = 8$ and $x = (x_0, \ldots, x_7)$, then $x' = (x_0, x_4, x_2, x_6, x_1, x_5, x_3, x_7)$ because the indices are permutated as

$$\begin{aligned}0 = 000_2 &\longmapsto 000_2 = 0\\ 1 = 001_2 &\longmapsto 100_2 = 4\\ 2 = 010_2 &\longmapsto 010_2 = 2\\ 3 = 011_2 &\longmapsto 110_2 = 6\\ 4 = 100_2 &\longmapsto 001_2 = 1\\ 5 = 101_2 &\longmapsto 101_2 = 5\\ 6 = 110_2 &\longmapsto 011_2 = 3\\ 7 = 111_2 &\longmapsto 111_2 = 7\end{aligned}$$

2.5.3 Generalized DFT

The DFT introduced previously may be generalized to other algebraic structures than the complex field. The proofs are more or less identical to the ones given for the complex DFT. For the sake of completeness some of them will be repeated in this more general setting.

Definition 6. *Let $\mathcal{R}$ be a ring with identity. An element $\omega \in \mathcal{R}$ is a n^{th}* primitive root of unity *if $n > 1$ is the smallest integer for which the following holds,*

1. $\omega \neq 1$
2. $\omega^n = 1$

Definition 7. *Let $\mathcal{R}$ be a ring with identity. An element $\omega \in \mathcal{R}$ is a n^{th}* principal root of unity *if ω is a n^{th} primitive root of unity and*

$$\sum_{k=0}^{n-1} \omega^{mk} = 0, \quad \forall m \in \{1, 2, \dots, n-1\} \tag{2.112}$$

Theorem 6. *Let $\mathcal{R}$ be a ring with identity. If $\omega \in \mathcal{R}$ is a n^{th} primitive root of unity and $(\omega^m - 1)^{-1} \in \mathcal{R}$ for $m \in \{1, 2, \dots, n-1\}$, then ω is a n^{th} principal root of unity.*

Proof. Let $m \in \{1, 2, \dots, n-1\}$. Then we have,

$$\begin{aligned}\sum_{k=0}^{n-1} \omega^{mk} =& 1 + \omega^m + \omega^{2m} + \dots + \omega^{(n-1)m} = \\ =& (1 + \omega^m + \omega^{2m} + \dots + \omega^{(n-1)m})(\omega^m - 1)(\omega^m - 1)^{-1} = \\ =& (\omega^{nm} - 1)(\omega^m - 1)^{-1} = ((\omega^n)^m - 1)(\omega^m - 1)^{-1} = 0 \cdot (\omega^m - 1)^{-1} = 0\end{aligned}$$

□

Corollary 1. *Let $\mathcal{K}$ be a field. If $\omega \in \mathcal{K}$ is a n^{th} primitive root of unity, then ω is a n^{th} principal root of unity.*

Theorem 7. *Let $\mathcal{R}$ be a ring with identity. If $\omega \in \mathcal{R}$ is a n^{th} principal root of unity, then $\omega^{-1} \in \mathcal{R}$ is a n^{th} principal root of unity.*

Proof. As ω is a n^{th} primitive root of unity we have,

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

which shows that $\omega^{-1} \in \mathcal{R}$. It is clear that $\omega^{-1} \neq 1$ and that $\omega^{-n} = (\omega^{-1})^n = 1$. For ω^{-1} to be a n^{th} primitive root of unity, we have to show that n is the smallest integer such that $\omega^{-n} = 1$. Suppose there is an integer $1 < m < n$ such that $\omega^{-m} = 1$. Then,

$$\omega^{-m} = 1 \Leftrightarrow \omega^m \omega^{-m} = \omega^m \cdot 1 \Leftrightarrow 1 = \omega^m$$

which is a contradiction of ω being a n^{th} primitive root of unity.

For some fixed integer $m \in \{1, \ldots, n-1\}$, we have

$$\sum_{k=0}^{n-1} \omega^{mk} = 1 + \omega^m + \omega^{2m} + \cdots + \omega^{(n-1)m} = 0$$

$$\Longleftrightarrow$$

$$(\omega^{-1})^{(n-1)m}[1 + \omega^m + \omega^{2m} + \cdots + \omega^{(n-2)m}) + \omega^{(n-1)m}] = 0$$

$$\Longleftrightarrow$$

$$(\omega^{-1})^{(n-1)m} + (\omega^{-1})^{(n-2)m} + (\omega^{-1})^{(n-3)m} + \cdots + (\omega^{-1})^m + 1 = 0$$

$$\Longleftrightarrow$$

$$\sum_{k=0}^{n-1} (\omega^{-1})^{mk} = 0$$

This establishes ω^{-1} to be a n^{th} principal root of unity. □

Definition 8. *Let $\mathcal{R}$ be a ring with identity and let $\omega \in \mathcal{R}$ be a n^{th} principal root of unity. Then the Generalized Discrete Fourier Transform, $\mathcal{F}$, is defined to be,*

$$\mathcal{F} : \mathcal{R}^n \longrightarrow \mathcal{R}^n$$

$$x \in \mathcal{R}^n \longmapsto y \in \mathcal{R}^n$$

where y is the n-tuple $(y_0, y_1, \ldots, y_{n-1})$ and

$$y_m = \sum_{k=0}^{n-1} x_k \omega^{mk}, \quad m \in \{0, \ldots, n-1\} \tag{2.113}$$

Theorem 8. *Let $\mathcal{R}$ be a ring with identity, and let $\omega \in \mathcal{R}$ be a n^{th} principal root of unity. Let $n^{-1} \in \mathcal{R}$. Then the inverse Generalized Discrete Fourier Transform, $\mathcal{F}^{-1}$, is given by*

$$\mathcal{F}^{-1}(y) = (x_0, x_1, \ldots, x_{n-1}), \quad x_m = n^{-1} \sum_{k=0}^{n-1} y_k \omega^{-mk}, \quad m \in \{0, \ldots, n-1\} \tag{2.114}$$

Proof.

$$n^{-1}\sum_{k=0}^{n-1} y_k\omega^{-mk} = n^{-1}\sum_{k=0}^{n-1}\sum_{j=0}^{n-1} x_j\omega^{kj}\omega^{-mk} = n^{-1}\sum_{j=0}^{n-1} x_j \sum_{k=0}^{n-1}\omega^{k(j-m)}$$

When $j = m$, we have

$$n^{-1}\sum_{j=0}^{n-1} x_j \sum_{k=0}^{n-1} 1 = n^{-1}\sum_{j=0}^{n-1} x_j(1+1+\cdots+1) = n^{-1}nx_j = x_m$$

When $j > m$, we have by definition (7)

$$n^{-1}\sum_{j=0}^{n-1} x_j \sum_{k=0}^{n-1}\omega^{k(j-m)} = n^{-1}\sum_{j=0}^{n-1} x_j \cdot 0 = 0$$

When $j < m$ we have by theorem 7,

$$n^{-1}\sum_{j=0}^{n-1} x_j \sum_{k=0}^{n-1}\omega^{k(j-m)} = n^{-1}\sum_{j=0}^{n-1} x_j \sum_{k=0}^{n-1}(\omega^{-1})^{k(m-j)} = n^{-1}\sum_{j=0}^{n-1} x_j \cdot 0 = 0$$

□

In the theorem above we have used that $n = n \cdot 1 = \underbrace{1+1+\cdots+1}_{n} \in \mathcal{R}$, so the notation n^{-1} makes sense. The requirement that $n^{-1} \in \mathcal{R}$ is necessary because there are rings which contain n^{th} principal roots but $n^{-1} \notin \mathcal{R}$. As an example we have $\mathbb{Z}_4$ that contains 3 as a 2nd principal root of unity but $2^{-1} \notin \mathbb{Z}_4$. Another example is $\mathbb{Z}_{10}$, which has 7 as a 4th principal root of unity, but in this case $4^{-1} \notin \mathbb{Z}_{10}$.

For a field $\mathcal{K}$, we may relax the above requirement as it is implicit in this case. We have to make sure that $n \cdot 1 = 1+1+\cdots+1 \neq 0$. The characteristic, char $\mathcal{K}$, for $\mathcal{K}$ is either 0 or p for some prime p. For char $\mathcal{K} = 0$, this is not an issue. For char $\mathcal{K} = p$, we know that $|\mathcal{K}| = p^m$ for some integer $m \geq 1$. As a principal root of unity is a generator for a subgroup of the multiplicative group in $\mathcal{K}$, we have, by the theorem of Lagrange, that $n|p^m - 1$. Suppose $n \cdot 1 = 0 \in \mathcal{K}$. Then $p|n \Leftrightarrow n = pq$, $(q \in \mathbb{Z}^+)$. But this would imply $pq|p^m - 1$, which is impossible.

The definition of the discrete cyclic convolution and the proof of the convolution theorem may also be put in this more general setting. This will enable the possibility to do efficient arithmetic in, for example, a field. As the proofs are more or less identical to the previous ones, we don't repeat them here. Note that the proof of the convolution theorem now requires a *commutative* ring with identity. We also observe that $\mathbb{R}$ and $\mathbb{Z}$ both lack primitive roots of unity. This is the reason why we introduced the complex DFT as it uses the complex field, which for $\omega = e^{\frac{2\pi i}{n}}$ has n^{th} principal roots of unity for all integers $n > 1$. This property makes it possible to use DFTs of any length. But the computations performed uses floating-point calculations that may introduce numerical errors and also, as we are only interested in integer multiplication, this may not be an optimal solution. With the Generalized DFT we have the option to perform fast multiplication in algebraic structures using only integer arithmetic.

2.5.4 Schönhage-Strassen multiplication

The Schönhage-Strassen algorithm for integer multiplication is an example of the generalized DFT in the ring $\mathbb{Z}_{2^{n/2}+1}$, where n is a power of two. In this ring one may observe that

$$2^{n/2} \equiv -1 \pmod{2^{n/2}+1} \tag{2.115}$$

which implies that

$$2^{n} \equiv 1 \pmod{2^{n/2}+1} \tag{2.116}$$

Furthermore, $n = 2^k$ for some $k \in \mathbb{Z}^+$, so $\gcd(n, 2^{n/2}+1) = 1$ shows that n is a unit in $\mathbb{Z}_{2^{n/2}+1}$. Another important property is given by the following theorem.

Theorem 9. *If $n = 2^j$ for some $j = 1, 2, \ldots$ then*

$$\sum_{k=0}^{n-1} 2^{mk} = 0, \quad m \in \{1, 2, \ldots, n-1\}$$

in the ring $\mathbb{Z}_{2^{n/2}+1}$.

Proof. By using the relation for the first n terms in a geometric series and (2.116) one may write

$$\sum_{k=0}^{n-1} 2^{mk} = (2^n)^m - 1 = 0 \qquad \pmod{2^{n/2}+1} \tag{2.117}$$

$\square$

From the discussion above, we find that $2 \in \mathbb{Z}_{2^{n/2}+1}$ is a n^{th} principal root of unity and that we may use the inverse DFT as $n^{-1} \in \mathbb{Z}_{2^{n/2}+1}$. For a binary computer, there are some advantages in using this ring when it comes to implementation

- As all principal roots are on the form 2^k for some k, multiplication and division of an integer by this principal root may be implemented as simple shifts.
- It is possible to use a modified form of the convolution theorem, known as *negacylic convolution*, which is both more efficient than the normal convolution and has the reduction modulo $2^{n/2} + 1$ as a side effect.
- The carry propagation may take advantage of the fact that reducing a value modulo $2^{n/2}+1$ can be done using only shift and add operations.

2.6 Precalculation and Zech's logarithm

For finite fields of moderate orders, it is possible to precalculate the multiplication and addition tables. The operands are then used as indices into the corresponding row and column of the table to find the result of the operation. Given a field with q elements, this would require two tables, each consuming q^2 elements of storage. It is possible to do marginally better with respect to storage requirements. By removing 0 and 1 from the multiplication table, and removing 0 from the addition table, and handle these cases separately. With multiplication, we have that if one of the operands is 0 then the result is 0, and if one of the operands is 1, the result is the other operand. With addition, we have that if one of the operands is 0, the result is the other operand. This would require $(q-1)^2 + (q-2)^2$ of total storage for the table elements. The ratio between these two table storage requirements is

$$\frac{(q-1)^2 + (q-2)^2}{2q^2} = 1 - \frac{3}{q} + \frac{5}{2q^2} \tag{2.118}$$

This expression quite rapidly tends to 1 as q get larger. For $q = 128$ we have that the saving is 763 elements and the expression in (2.118) is approximately 0.98. For $q = 1024$, the saving is 6139 elements and the ratio is approximately 0.997. This suggests that this kind of optimization may not be worth doing, especially as it slightly increases the complexity because of the special handling of 0 and 1.

There exists another approach based on this idea of this precalculation technique, which requires slightly more computations but with lesser storage requirement for the precalculated data. If we let $\mathcal{K}$ denote a finite field, with $|\mathcal{K}| = q$, we may write

$$\mathcal{K} = \mathcal{K}^* \cup \{0\} \tag{2.119}$$

where $\mathcal{K}^*$ is the multiplicative group of $\mathcal{K}$. It is known that $\mathcal{K}^*$ contains a generator α. Each element $\beta \in \mathcal{K}^*$ is on the form α^k for some $k \in \{0, 1, \ldots, q-2\}$.

If we use the symbol ∞, and denote $0 \in \mathcal{K}$ as α^{∞}, we have that each $\beta \in \mathcal{K}$ is on the form α^{γ} for some $\gamma \in \{0, 1, \dots, q-2\} \cup \{\infty\}$. For $\gamma \in \{0, 1, \dots, q-2\} \cup \{\infty\}$, we define

$$\gamma + \infty = \infty, \quad \infty + \gamma = \infty \tag{2.120}$$

$$\infty - \gamma = \infty, \quad \gamma \neq \infty \tag{2.121}$$

$$\infty \bmod n = \infty \tag{2.122}$$

$$n \bmod 0 = n \tag{2.123}$$

Multiplication of two elements $\beta, \beta' \in \mathcal{K}$ is then given by

$$\beta\beta' = \alpha^{\gamma}\alpha^{\gamma'} = \alpha^{\gamma+\gamma' \bmod q-1} \quad \gamma, \gamma' \in \{0, 1, \dots, q-2\} \cup \{\infty\} \tag{2.124}$$

and addition is given by

$$\beta + \beta' = \alpha^{\gamma} + \alpha^{\gamma'} \quad \gamma, \gamma' \in \{0, 1, \dots, q-2\} \cup \{\infty\} \tag{2.125}$$

In the case of addition, whenever $\beta \neq 0$, we may use the relation

$$\beta + \beta' = \alpha^{\gamma} + \alpha^{\gamma'} = (1 + \alpha^{\gamma'-\gamma \bmod q-1})\alpha^{\gamma} \tag{2.126}$$

The restriction $\beta \neq 0$ in (2.126) is simply a way to avoid definitions and notations which do not add anything of interest to the discussion at hand. One example is that the expression $0 + \beta'$ would lead to

$$0 + \beta' = \alpha^{\infty} + \alpha^{k} = (1 + \alpha^{k-\infty \bmod q-1})\alpha^{\infty} \tag{2.127}$$

At first, we haven't defined $\alpha^{k-\infty}$. Secondly, for some definition of this expression, we still end up with a factor of α^{∞}, which we previously defined to be 0. It would be a bit awkward to have the relation in (2.127) equal β' instead of 0, so we choose to handle this case separately.

Expression on the form $1 + \alpha^{\gamma}$ is related to the *Zech's logarithm*, which we now define.

Definition 9. *Let $\mathcal{K}$ be a finite field with $|\mathcal{K}| = q$ and $\alpha \in \mathcal{K}^*$ a generator for $\mathcal{K}^*$. Then, the* Zech's logarithm *$z(\gamma)$ is defined to be the mapping,*

$$z : \{0, 1, \dots, q-2\} \cup \{\infty\} \longrightarrow \{0, 1, \dots, q-2\} \cup \{\infty\}$$

such that,

$$\alpha^{z(\gamma)} = 1 + \alpha^{\gamma}$$

Let us now assume a finite field $\mathcal{K}$ with $|\mathcal{K}| = q$ and with a generator α. As each element is on the form

$$\alpha^\gamma, \quad \gamma \in \{0, 1, \ldots, q-2\} \cup \{\infty\} \tag{2.128}$$

we use γ as the identifier for these elements. Multiplication of two elements $\beta = \alpha^\gamma, \beta' = \alpha^{\gamma'} \in \mathcal{K}$ may then proceed by the rule given in (2.124). Addition is accomplished by combining the rule given in (2.126) and the Zech's logarithm (definition 9), whenever $\beta \neq 0$. For $\beta = 0$, we note that the result of the addition is β'. That is, (for $\beta \neq 0$)

$$\beta + \beta' = \alpha^\gamma + \alpha^{\gamma'} = (1 + \alpha^{\gamma' - \gamma \bmod q-1})\alpha^\gamma = \alpha^{z(\gamma' - \gamma \bmod q-1)}\alpha^\gamma \tag{2.129}$$

where the final product $\alpha^{z(\gamma' - \gamma \bmod q-1)}\alpha^\gamma$ is calculated by using (2.124). Note that the expression $z(\gamma' - \gamma \bmod q - 1)$ is implemented as an efficient table lookup of precomputed values for the Zech's logarithm.

Example The field $\mathbb{Z}_5$ may be seen as having 3 as a generator with the following properties

$$3^0 \equiv 1 \pmod 5 \tag{2.130}$$
$$3^1 \equiv 3 \pmod 5 \tag{2.131}$$
$$3^2 \equiv 4 \pmod 5 \tag{2.132}$$
$$3^3 \equiv 2 \pmod 5 \tag{2.133}$$

in addition to defining $3^\infty \equiv 0 \pmod 5$. Multiplication of 3 and 4 is computed in the normal way

$$3 \cdot 4 = 3^1 \cdot 3^2 = 3^{1+2} = 3^3 = 2 \tag{2.134}$$

Addition of 3 and 4 is done by using the Zech's logarithm which has the following mapping

$$0 \mapsto 3 \tag{2.135}$$
$$1 \mapsto 2 \tag{2.136}$$
$$2 \mapsto \infty \tag{2.137}$$
$$3 \mapsto 1 \tag{2.138}$$
$$\infty \mapsto 0 \tag{2.139}$$

This mapping is then used in the calculation

$$\begin{aligned} 3 + 4 = 3^1 + 3^2 = (1 + 3^{2-1})3^1 = (1 + 3^1)3^1 = \\ = [z(1) = 2] = 3^2 3^1 = 3^3 = 2 \end{aligned} \tag{2.140}$$

Concerning storage requirements, this method requires q elements of precalculated data for the Zech's logarithm. This is substantially more effective, with respect to storage requirement, compared to the method using complete precalculated tables which needed room for $2q^2$ elements.

2.7 Polynomial representation and normal basis

The elements of a finite field may be represented in a multitude of ways. Further, we will explore some of the most common representations and how they affect arithmetic in a finite field. Most proofs are omitted in this section. The reader is instead directed to Svensson [9] or Beachy and Blair [2] for the full exposition of these proofs. Furthermore, all occurrences of the symbol p is meant to denote a prime number, and $\mathbb{F}_{p^n}$ is meant to denote the finite field with p^n elements, for some integer n.

2.7.1 Polynomial representation

In the polynomial representation each element of the field $\mathbb{F}_{p^n}$ may be represented by a polynomial. The theorems below and the following discussion will demonstrate how to create such representations.

Theorem 10. *For every $n \in \mathbb{Z}^+$, there exists an irreducible polynomial of degree n in $\mathbb{Z}_p[x]$.*

Theorem 11. *Let $\mathcal{K}$ be a field and let $q(x) \in \mathcal{K}[x]$ be an irreducible polynomial over $\mathcal{K}$. Then the quotient ring $\mathcal{K}[x]/\langle q(x)\rangle$ is a field.*

Theorem 12. *Two finite fields are isomorphic if, and only if, they have the same number of elements.*

With these theorems, we are in the position of putting together a "recipe" for actually constructing the field $\mathbb{F}_{p^n}$ where, $n \in \mathbb{Z}^+$:

1. Choose an irreducible polynomial $q(x)$ of degree n in $\mathbb{Z}_p[x]$. This polynomial exists by theorem 10.
2. By theorem 11, we have that $\mathbb{Z}_p[x]/\langle q(x)\rangle$ is a field. It contains polynomials on the form $a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$ where $a_k \in \mathbb{Z}_p$ for $k = 0, \ldots, n-1$. Each a_k may be selected in p ways and we have n coefficients. By the multiplication principle, we therefore conclude that the field $\mathbb{Z}_p[x]/\langle q(x)\rangle$ contains p^n elements.

3. Theorem 12 states that $\mathbb{Z}_p[x]/\langle q(x)\rangle$ is isomorphic to $\mathbb{F}_{p^n}$

In order to reduce the notation slightly, we interchangeably use $\mathbb{Z}_p[x]/\langle q(x)\rangle$ and $\mathbb{F}_{p^n}$. One consequence of this is that we assume $\mathbb{F}_{p^n}$ to contain polynomials on the form

$$a_0 + a_1x + \cdots + a_{n-1}x^{n-1}, \quad a_k \in \mathbb{Z}_p \quad (k = 0, \ldots, n-1)$$

We also denote $(a_{n-1} \ldots a_1a_0)$ as an alternative to the polynomial given above.

To construct the field $\mathbb{F}_9 = \mathbb{F}_{3^2}$ we first find an irreducible polynomial of degree 2 in $\mathbb{Z}_3[x]$. Such a polynomial is given by $x^2 + 1$, and therefore $\mathbb{Z}_{3^2}/\langle x^2 + 1\rangle$ is a field containing 9 elements. The elements are:

$$\begin{array}{ccc} 0 & x & 2x \\ 1 & x+1 & 2x+1 \\ 2 & x+2 & 2x+2 \end{array}$$

Addition in $\mathbb{F}_{p^n}$ is done by adding the polynomials in the usual way, reducing each resulting coefficient modulo p. Multiplication is also done in the usual way by multiplying the polynomials, reducing the result modulo the irreducible polynomial $q(x)$.

Example Addition of $(2x + 1)$ and $2x$ is carried out as

$$(2x + 1) + 2x = 4x + 1 \text{ which reduces to } x + 1 \tag{2.141}$$

and multiplication of $(x + 1)$ and $(2x + 2)$

$$(x + 1)(2x + 2) = 2x^2 + 4x + 2 \text{ which reduces to } x \tag{2.142}$$

With this brief introduction to polynomial representation of elements in finite fields, we turn our attention to a special field. Namely the *binary field*, $\mathbb{F}_{2^n}$. This field has some appealing properties when it comes to binary computer implementations.

By the following definition, we introduce addition in $\mathbb{F}_{2^n}$.

Definition 10. *We define the function $xor(x, y)$ to be*

$$\begin{array}{rcl} xor : \mathbb{Z}_2 \times \mathbb{Z}_2 & \longrightarrow & \mathbb{Z}_2 \\ (x, y) \in \mathbb{Z}_2 \times \mathbb{Z}_2 & \longmapsto & x + y \in \mathbb{Z}_2 \end{array}$$

The result of $xor(x, y)$ is 1 if and only if exactly one of the operands x, y is 1. Addition of x and y in the field $\mathbb{F}_{2^n}$ may now be defined as

$$x \boxplus y := (xor(x_{n-1}, y_{n-1}) \dots xor(x_1, y_1) \quad xor(x_0, y_0)) \tag{2.143}$$

As $-1 \equiv 1 \pmod 2$, it follows that addition and subtraction in $\mathbb{Z}_2$ are equivalent operations.

Definition 11. *We define the left shift of an element $x = (a_{n-1} \dots a_1 a_0) \in \mathbb{F}_{2^n}$ to be*

$$\begin{aligned} lshift : \mathbb{F}_{2^n} &\longrightarrow \mathbb{F}_{2^{n+1}} \\ (a_{n-1} \dots a_1 a_0) &\longmapsto (a_{n-1} \dots a_1 a_0 0) \end{aligned}$$

and we define

$$x \blacktriangleleft k := \underbrace{lshift(lshift(\dots lshift(x) \dots))}_{k}$$

Multiplication is then performed using shifts and additions.

Example

$$\begin{aligned} &(x^4 + x^2 + x + 1)(x^6 + x^3 + x) = \\ &x^4(x^6 + x^3 + x) + x^2(x^6 + x^3 + x) + x(x^6 + x^3 + x) + (x^6 + x^3 + x) = \\ &((1001010) \blacktriangleleft 4) \boxplus ((1001010) \blacktriangleleft 2) \boxplus ((1001010) \blacktriangleleft 1) \boxplus (1001010) = \\ &(10010100000) \boxplus (100101000) \boxplus (10010100) \boxplus (1001010) = \\ &(00110111100) \end{aligned} \tag{2.144}$$

The result is then reduced by the irreducible polynomial that is associated with the field $\mathbb{F}_{2^n}$.

The reduction of a polynomial may be done using additions and shifts until the result is a member of $\mathbb{F}_{2^n}$.

Example The field $\mathbb{F}_{2^3}$ has $x^3 + x + 1 = (1011)$ as an irreducible polynomial. Reducing $x^7 + x^6 + x^4 + 1 = (11010001)$ in this field is achieved by

$$\begin{array}{cll} & 11010001 & \\ \boxplus & 10110000 & [(1011) \blacktriangleleft 4] \\ & 01100001 & \\ \boxplus & 01011000 & [(1011) \blacktriangleleft 3] \\ & 00111001 & \\ \boxplus & 00101100 & [(1011) \blacktriangleleft 2] \\ & 00010101 & \\ \boxplus & 00010110 & [(1011) \blacktriangleleft 1] \\ & 00000011 & = (x+1) \end{array}$$

If a polynomial $f(x) \in \mathbb{Z}_2[x]$ has an even number of non-zero coefficients, we note that $f(1) = 0$ for all such polynomials. By the factor theorem, we may write $f(x)$ as

$$f(x) = (x-1)q(x), \quad q(x) \in \mathbb{Z}_2[x] \tag{2.145}$$

In this case, $f(x)$ is reducible, which makes $f(x)$ impossible to use as a reduction polynomial in the field $\mathbb{F}_{2^n}$. Neither may $f(x) = x^n$, for some $n > 1$, be used. In fact, a proper reduction polynomial must contain an odd number of terms, and at least three of them must be non-zero.

Squaring of elements in binary fields is done by using the following result.

Theorem 13. *If $a = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} \in \mathbb{F}_{2^n}$, then*

$$a^2 = \sum_{k=0}^{n-1} a_k^2 x^{2k} = \sum_{k=0}^{n-1} a_k x^{2k}$$

Proof. Let $q_m = a_0 + a_1x + \cdots + a_{m-1}x^{m-1} \in \mathbb{F}_{2^n}$ $(1 \leq m < n)$. The claim above certainly holds for $(q_1)^2 = a_0^2 = a_0$ and $(q_2)^2 = (a_0 + a_1x)^2 = a_0^2 + 2a_0a_1x + a_1^2x^2 = a_0 + a_1x^2$. If the claim is false, we have, by the well-ordering principle, a smallest m such that $q_m^2 \neq \sum_{k=0}^{m-1} a_k x^{2k}$. But

$$\begin{aligned} q_m^2 &= (q_{m-1} + a_{m-1}x^{m-1})^2 = \\ &= q_{m-1}^2 + 2q_{m-1}a_{m-1}x + a_{m-1}^2 x^{2(m-1)} = \\ &= \sum_{k=0}^{m-2} a_k x^{2k} + a_{m-1}x^{2(m-1)} = \sum_{k=0}^{m-1} a_k x^{2k} \end{aligned}$$

which would be a contradiction. □

This suggests that squaring is done by inserting 0's between the binary digits in $(a_{n-1} \ldots a_1 a_0)$, and, as a result, we get $(a_{n-1}0 \ldots 0a_1 0a_0)$. A most efficient way of doing this insertion is by creating a table that maps a given value to a corresponding value containing 0's at the correct positions. A typical computer implementation could map 16-bit values to 32-bit values. This would require a table of $2^{16} = 65536$ elements, each 32-bit wide. One may also note that there are 15 places where to insert a 0 between digits in a 16-bit value. Therefore, the result of the insertions will occupy 31 bits. By letting the most significant bit be 0 in each 32-bit entry of the table, one may easily use the table for arbitrary lengths of bits in the squaring operand. It is just a simple matter of concatenating the 32-bit table results for each corresponding 16-bit value.

2.7.2 Normal basis

The discussion above shows us an effective way of computing squares in $\mathbb{F}_{2^n}$. With the use of *normal basis*, we can take this operation of squaring a bit further. Given a field $\mathbb{F}_{p^n}$ we may view this as the vector space $\mathbb{F}_{p^n}$ over the field $\mathbb{F}_p$.

Definition 12. *Let $\mathbb{F}_{p^n}$ be the vector space over the field $\mathbb{F}_p$. Then the set $\{\beta\ \beta^p\ \ldots\ \beta^{p^{n-1}}\}$, for $\beta \in \mathbb{F}_{p^n}$, forms a* normal basis *if the set $\{\beta\ \beta^p\ \ldots\ \beta^{p^{n-1}}\}$ is linearly independent.*

It is known that for every field $\mathbb{F}_{p^n}$, there exists such a normal basis. We can now represent each element $\alpha \in \mathbb{F}_{p^n}$ as

$$\alpha = a_0\beta + a_1\beta^p + \cdots + a_{n-1}\beta^{p^{n-1}}, \quad a_0, \ldots, a_{n-1} \in \mathbb{F}_p \tag{2.146}$$

We also note that the multiplicative group $\mathbb{F}^*_{p^n}$ contains $p^n - 1$ elements and recall from group theory that

$$\gamma^{p^n-1} = 1, \quad \gamma \in \mathbb{F}^*_{p^n} \tag{2.147}$$

Let us now shift our attention to the field $\mathbb{F}_{2^n}$. Assuming that $\{\beta\ \beta^2\ \ldots\ \beta^{2^{n-1}}\}$ forms a normal basis for $\mathbb{F}_{2^n}$, we have that

$$\alpha^2 = (\sum_{k=0}^{n-1} a_k\beta^{2^k})^2 = \sum_{k=0}^{n-1} a_k\beta^{2^{k+1}} = \sum_{k=0}^{n-1} a_{(n-1+k \bmod n)}\beta^{2^k} \tag{2.148}$$

This may be shown using a similar argument as in theorem 13 and by applying (2.147) to β^{2^n} $(= \beta^{2^n-1}\beta = \beta)$.

To summarize, we have achieved the following

$$\alpha^2 = (a_0\ a_1 \ldots a_{n-1})^2 = (a_{n-1}\ a_0 \ldots a_{n-2}) \tag{2.149}$$

This implies that squaring is done by rotating the vector one step to the right. Note that general multiplication using normal basis representation may be a very expensive operation.

Chapter 3

Computer implementations

In this chapter, there are some concrete implementations of the theory introduced in the former chapter. The main algorithms will be presented with a table of time measurements for different input sizes. The test data used for measurement is the first n decimals of π and the first n decimals of e, where n is the operand length. The classic algorithm implementation has also served as a reference implementation as its results where saved and later used to check the other algorithms for correctness. The source code is written in standard C++. The algorithms were executed on a HP Compaq nx 7400 portable computer with a 1.83 Ghz Intel Centrino Duo and 2 Gb of RAM, running Windows Vista 32-bit. All source code were compiled using Microsoft Visual Studio 2005 (VC++ v8.0) with highest code optimization and preferring speed over size.

3.1 Classic multiplication

The implementation of the classic multiplication algorithm is a straightforward application of the definition of polynomial multiplication. By using C++ *templates*, the function is parameterized on the type used by the coefficients. This makes the algorithm usable with all types that fulfill the requirements of addition and multiplication. It is then possible to choose a type that represents the problem domain in the best way.

Classic multiplication algorithm

```
namespace classic {


template<typename ForwardIterator1, typename ForwardIterator2>
void multiply(ForwardIterator1 op1first, ForwardIterator1 op1last,
            ForwardIterator1 op2first, ForwardIterator1 op2last,
            ForwardIterator2 result)
{
   std::fill_n(result, std::distance(op1first, op1last) + std::distance(op2first, op2last),
            typename std::iterator_traits<ForwardIterator2>::value_type());

   for (; op1first != op1last; ++op1first, ++result) {
      ForwardIterator2 res = result;
      for (ForwardIterator1 first = op2first; first != op2last; ++first, ++res) {
         *res += *op1first * *first;
      }
   }
}

} // namespace classic
```

Classic multiplication execution time

digits	time	digits	time	digits	time
2^3	0.18 μs	2^9	0.45 ms	2^{15}	1.95 s
2^4	0.51 μs	2^{10}	1.76 ms	2^{16}	7.86 s
2^5	1.83 μs	2^{11}	7.04 ms	2^{17}	31.3 s
2^6	6.94 μs	2^{12}	29.0 ms	2^{18}	137 s
2^7	28.6 μs	2^{13}	125 ms	2^{19}	1113 s
2^8	111 μs	2^{14}	489 ms	2^{20}	3850 s

The algorithm was specialized using the type *unsigned int*, which in this implementation may hold values in $[0, 2^{32} - 1]$.

3.2 Karatsuba multiplication

The algorithm source presented here is based on a simple implementation provided by Burch [4]. The source code implemented here uses parameterization, as with the classic algorithm, in order to be as general as possible. In addition, there is an analysis of how many digits it is possible to use without distorting the result.

Karatsuba is a recursive algorithm, which needs to pass partial results up the call chain in order to combine them to the final result. There are typically two ways of achieving this

a) Have each step allocate a storage area and return this up to the caller.

b) Have an overall storage area that is used for all recursive calls.

Option a) usually performs much worse than option b) as the operation of allocating storage along the way is too costly when the number of digits get large. With option b), we pre-allocate storage which uses a size of $6 \cdot n$ elements, where n is the digit length. This area holds the sub-results and the final result, which occupies $2 \cdot n$ elements. It also uses a threshold value for which the algorithm switches to use classic multiplication. This threshold is used because the classic algorithm is more effective for small n.

One implementation issue that is of concern is how to choose a computer word and a digit length such that the algorithm doesn't overflow. In the classic algorithm, the largest valued coefficient, c_{max}, is found by letting all digits in the operands have the value $B-1$. The algorithm will thus calculate c_{max} as

$$c_{max} = \sum_{k=0}^{n-1} (B-1)^2 = n(B-1)^2 \tag{3.1}$$

where $n = 2^m (m > 0)$ is the number of digits in one operand and B is the radix representation. The Karatsuba algorithm will find the value c_{max} slightly different

$$c_{max} = (\ldots((2^{2m}(B-1)^2) - 2^{2m-1}(B-1)^2) - \cdots) - 2^m(B-1)^2 \tag{3.2}$$

Note that the value of c_{max} is the same in both the classical and Karatsuba algorithm and both algorithms need to consider a computer word such that the calculation doesn't overflow. But the difference is that the Karatsuba algorithm starts out with calculating the intermediate result $2^{2m}(B-1)^2$, which is 2^m times larger in magnitude than the final result. One way to remedy the situation is to select a computer word capable of holding this intermediate result. The problem with this approach is that we must use more storage and that it will potentially slow down the computational time. A better approach is given below.

We begin with a definition of the modulus operator

$$a \bmod N := a - \lfloor a/N \rfloor N, \quad (a \geq 0, N > 0) \tag{3.3}$$

followed by the relations

$$\begin{aligned} ((a \bmod N) - (b \bmod N)) \bmod N &= (a-b) \bmod N && (3.4) \\ ((a \bmod N)(b \bmod N)) \bmod N &= (ab) \bmod N && (3.5) \\ & && (3.6) \end{aligned}$$

If we let $q_1 = \lfloor a/N \rfloor, q_2 = \lfloor b/N \rfloor$, relation (3.4) follows from

$$\begin{aligned}
&(a \bmod N) - (b \bmod N)) \bmod N = \\
&= (a - q_1 N - (b - q_2 N)) - \lfloor (a - q_1 N - (b - q_2 N))/N \rfloor = \\
&= a - b - N(q_1 - q_2) - \lfloor (a-b)/N - q_1 + q_2 \rfloor N \\
&= a - b - N(q_1 - q_2) - (\lfloor (a-b)/N \rfloor - q_1 + q_2) N \\
&= a - b - \lfloor (a-b)/N \rfloor N - N(q_1 - q_2) - N(q_2 - q_1) \\
&= a - b - \lfloor (a-b)/N \rfloor N = a - b \bmod N
\end{aligned} \tag{3.7}$$

Relation (3.5) may be shown using the same principle.

When using *unsigned integer types* in C++ one is guaranteed that calculations involving such types are reduced modulo 2^w, where w is the number of bits in the value representation [1]. Assuming such a type $N = 2^w$ and in combination with (3.4) and (3.5), relation (3.2) becomes

$$\begin{aligned}
&(((\ldots(2^{2m} \bmod N \cdot (B-1)^2 \bmod N) \bmod N - \\
&(2^{2m-1} \bmod N \cdot (B-1)^2 \bmod N)) \bmod N - \\
&\cdots) \bmod N) - 2^m \bmod N(B-1)^2 \bmod N) \bmod N = \\
&= (((\ldots(2^{2m}(B-1)^2 \bmod N - 2^{2m-1}(B-1)^2 \bmod N) \bmod N - \\
&\cdots) \bmod N) - 2^m(B-1)^2) \bmod N = \\
&= (\ldots((2^{2m}(B-1)^2) - 2^{2m-1}(B-1)^2) - \cdots) - 2^m(B-1)^2 \bmod N
\end{aligned} \tag{3.8}$$

If

$$(\ldots((2^{2m}(B-1)^2) - 2^{2m-1}(B-1)^2) - \cdots) - 2^m(B-1)^2 \bmod N < N \tag{3.9}$$

the relation is equivalent to (3.2). It follows that we have the same requirement for the Karatsuba algorithm as with the classical algorithm with respect to the overflow constraint. We should select a computer word, depending on the number of digits, as

$$n(B-1)^2 < 2^w \tag{3.10}$$

where w is the number of bits in the value representation for some unsigned type. The use of delayed carry propagation will need to tighten the requirement slightly, as given in (2.28).

Karatsuba multiplication algorithm

```
namespace karatsuba {


const int threshold = 16;
template<typename RandomAccessIterator, typename T>
void multiply(RandomAccessIterator a,
              RandomAccessIterator b,
              RandomAccessIterator ret, T d)
{
   const T d_half = d / 2;
   const T d5    = d * 5;

   RandomAccessIterator alo = a, ahi = a + d_half;
   RandomAccessIterator blo = b, bhi = b + d_half;
   RandomAccessIterator asum = ret + d5;
   RandomAccessIterator bsum = asum + d_half;
   RandomAccessIterator x1 = ret;
   RandomAccessIterator x2 = ret + d;
   RandomAccessIterator x3 = x2 + d;

   if (d <= threshold) {
      classic::multiply(a, a + d, b, b + d, ret);
      return;
   }

   for (T i = 0; i < d_half; ++i) {
      asum[i] = ahi[i] + alo[i];
      bsum[i] = bhi[i] + blo[i];
   }

   multiply(alo, blo, x1, d_half);
   multiply(ahi, bhi, x2, d_half);
   multiply(asum, bsum, x3, d_half);

   for (T i = 0; i < d; ++i)
      x3[i] = x3[i] - x1[i] - x2[i];

   for (T i = 0; i < d; ++i)
      ret[i + d_half] += x3[i];
}

}  // namespace karatsuba
```

Karatsuba multiplication execution time

digits	time	digits	time	digits	time
2^3	0.20 μs	2^9	0.18 ms	2^{15}	134.6 ms
2^4	0.54 μs	2^{10}	0.54 ms	2^{16}	411.0 ms
2^5	1.76 μs	2^{11}	1.65 ms	2^{17}	1.22 s
2^6	5.77 μs	2^{12}	5.07 ms	2^{18}	3.67 s
2^7	18.6 μs	2^{13}	14.7 ms	2^{19}	11.0 s
2^8	56.8 μs	2^{14}	44.7 ms	2^{20}	33.1 s

The algorithm was specialized using the type *unsigned int*, which in this implementation may hold values in $[0, 2^{32} - 1]$.

3.3 Cooley-Tukey multiplication

The algorithm used is an implementation of the presentation given by Crandall and Pomerance [6] with the possibility to parameterize the algorithm.

Cooley-Tukey multiplication algorithm

```
namespace fourier {

template<typename RandomAccessIterator, typename Comp>
void cooley_tukey(RandomAccessIterator first, RandomAccessIterator last, Comp comp)
{
    typedef typename std::iterator_traits<RandomAccessIterator>::difference_type size_type;
    typedef typename Comp::value_type value_type;

    const size_type n = std::distance(first, last);

    fourier_detail::scramble(first, n);

    for (size_type m = 1; m < n; m *= 2) {
        for (size_type j = 0; j < m; ++j) {
            const value_type twiddle = comp.twiddle(j, m);
            for (size_type i = j; i < n; i += (m * 2)) {
                const value_type tmp = first[i];
                first[i]     = comp.add(tmp, comp.mul(twiddle, first[i + m]));
                first[i + m] = comp.sub(tmp, comp.mul(twiddle, first[i + m]));
            }
        }
    }
}

}   // namespace fourier
```

Cooley-Tukey multiplication execution time

digits	time	digits	time	digits	time
2^3	18.1 μs	2^9	2.22 ms	2^{15}	215 ms
2^4	42.0 μs	2^{10}	4.81 ms	2^{16}	494 ms
2^5	94.5 μs	2^{11}	10.3 ms	2^{17}	1.37 s
2^6	212 μs	2^{12}	22.4 ms	2^{18}	3.00 s
2^7	468 μs	2^{13}	48.3 ms	2^{19}	6.55 s
2^8	1.03 ms	2^{14}	101 ms	2^{20}	14.3 s

The algorithm was specialized using the type *std::complex<double>* where *double* follows the definition given by IEEE 754.

3.4 Cooley-Tukey with Zech's logarithm

In this section, there is an example of a combination of the Cooley-Tukey algorithm with the idea of using pre-calculated tables for certain operations. By using the generator α for the multiplicative group of a field, and by using a special symbol to denote 0, we may represent each element in the field $\mathcal{K}$ by its corresponding exponent n such that

$$\beta = \alpha^n \quad (\beta \in \mathcal{K}) \tag{3.11}$$

As previously, multiplication becomes addition of exponents, and addition become addition of exponents, using Zech's logarithm.

A good candidate for use in an implementation is the field $\mathbb{Z}_{257}$. The field contains relatively few elements, thereby making it possible to use in combination with pre-calculated tables, such that the use of storage is somewhat feasible. The multiplicative group of $\mathbb{Z}_{257}$ has 3 as a generator and therefore

$$\mathbb{Z}_{257} = \langle 3 \rangle \cup \{0\} \tag{3.12}$$

One may also note that as 3 is a 256^{th} principal root of unity, 3^2 is a 128^{th} principal root of unity, 3^4 is a 64^{th} principal root of unity and so on. This makes it possible to use the Cooley-Tukey algorithm for transformation lengths 256, 128, 64, and so on. We also define

$$0 := 3^{256} \tag{3.13}$$

in this field. Finally, in order to use these transformation lengths, we know that there exists an irreducible polynomial $q(x)$ in $\mathbb{Z}_{257}[x]$ with degree n such that

$$\mathbb{Z}_{257}[x]/\langle q(x) \rangle \tag{3.14}$$

forms a field with 257^n elements.

We may now provide the general outline of the multiplication algorithm of $a, b \in \mathbb{Z}_{257}[x]/\langle q(x)\rangle$.

1. Each element in the vector a, b is represented by its exponent, and we use the special exponent of 256 to denote the number 0.
2. Transform a, b with the use of the Cooley-Tukey FFT. The FFT takes advantage of the exponential representation (multiplication becomes addition, etc.).
3. Perform pointwise multiplication, once again taking advantage of the exponential representation.
4. Perform the inverse transformation
5. Perform, if necessary, any required reduction of the result.

As the implementation of the Cooley-Tukey algorithm is parameterized, it is easy to adapt to use this new scheme. We simply create a new composition class which implements our new rules. It also uses pre-calculated tables for the Zech's logarithm, additive inverses and the *twiddle* factor, which is the principal root of unity raised to some exponent. The example implementation also uses the classic algorithm as a reference to compare the results for correctness. Below is some timing results presented. Note however that the implementation does not perform any form of polynomial reduction and that the execution time of conversions between representations is not accounted for, so the results presented are not completely accurate. The focus of the example is the implementation part of the FFT for this particular representation. Nevertheless, it is interesting to have some kind of comparison of the algorithms. The source code is given in the appendix. One final observation concerning the Zech's logarithm. It is possible to use twice the storage for the table, using repeated values. This would avoid the necessity to do modulo calculation for the index into the table. In this case the table should be of moderate size in order to avoid any cache misses, which could introduce a high penalty for this kind of optimization.

Multiplication execution time in $\mathbb{Z}_{257}/\langle q(x)\rangle$

Algorithm	digits	time
Classic	64	$53.5\mu s$
FFT+Zech's	64	$29.7\mu s$
Classic	128	$212.4\mu s$
FFT+Zech's	128	$64.9\mu s$

The algorithm was specialized using the type *int*, which in this implementation may hold values in $[-2^{31}, 2^{31}-1]$.

3.5 Practice and theory comparisons

For some selected measurements, comparisons between the algorithms are presented below. The comparisons use the execution time spent in each algorithm, for a given input size, and computes the ratio between these two values. It also presents the theoretical ratio, which is the ratio between values for the theoretical execution time, given a fixed input size. If we assume that any constants are equal, they will cancel when forming the theoretical ratio.

From the tables below, it is seen that the actual and theoretical values can differ by several magnitudes. The assumption about equal constants is not entirely correct. In fact, these constants reflect the overhead introduced by regular bookkeeping for computer programs. For example memory management, function call, etc.

Classic / Karatsuba

n	time ratio	$n^2/n^{\log_2 3}$
2^4	0.94	3.16
2^7	1.54	7.49
2^{10}	3.26	17.8
2^{13}	8.50	42.1
2^{16}	19.1	99.8
2^{19}	101	237

Classic / Cooley-Tukey

n	time ratio	$n^2/n\log_2 n$
2^4	0.01	64
2^7	0.06	896
2^{10}	0.37	10240
2^{13}	2.59	106496
2^{16}	15.9	1048576
2^{19}	170	9961472

Karatsuba / Cooley-Tukey

n	time ratio	$n^{\log_2 3}/n \log_2 n$
2^4	0.01	1.27
2^7	0.04	2.44
2^{10}	0.11	5.77
2^{13}	0.30	15.0
2^{16}	0.83	41.1
2^{19}	1.68	117

3.6 Comments

This chapter presented concrete computer implementations of the classical multiplication algorithm, the Karatsuba multiplication algorithm and the FFT multiplication algorithm. Alongside, measurements of their different execution times have been presented. These measurements indicate, that the classical algorithm is preferred for small number of digits, but at 2^5 digits, the Karatsuba algorithm starts to be more efficient with respect to execution time. Nevertheless, with small number of digits, the difference between them might be considered negligible and one may in that case prefer the classic algorithm due to its simplicity and non-recursive nature. As for the FFT multiplication algorithm, it starts to outperform the classic algorithm for 2^{12} digits. And compared to the Karatsuba multiplication algorithm, we need to have at least 2^{18} digits in order to get performance benefits. But once again, the sheer execution time may not alone be the most important algorithm selection factor. Depending on the problem at hand, it is reasonable to choose some other algorithm that excels in some other area. As an example, the Karatsuba algorithm is 3 times faster than the FFT algorithm for 2^{13} digits. On the other hand, it uses 2 times more storage and it uses recursion, which also imposes some overhead with respect to storage. Now, the difference is $48.3\ ms - 14.7\ ms = 33.6\ ms$, which might be considered as acceptable in order to use less storage.

A word should be said about arbitrary precision software, which is used in real-world, industrial-strength, applications. They tend to use the fastest algorithms, some of them presented in this paper. They also implement them with great care, both with respect to the actual algorithm and the underlying hardware architecture. One important property to achieve is to have good *locality*. Implementations that exhibit good locality tends to run faster than implementations that lacks this property. Locality come in two forms: *temporal locality* and *spatial locality*. Implementations that have good temporal locality will, after referencing a memory location once, have a high probability of referencing the same memory location multiple times in the near future. Implementations that have good spatial locality will, after referencing a memory location once, have a high probability of referencing a nearby memory location in the near future [3]. The goal is to have the software reference memory location that will reside in *cache* memory, in contrast to fetch the location from main memory. This can have a major impact on running performance for arbitrary precision arithmetic, especially when the number of digits get large. The algorithms, or part of them, are usually implemented with "handcrafted" assembler code that will outperform the code generated by any high-level language compiler. When the underlying architecture is known, the implementation of the algorithm can be expressed with optimal instructions. These instructions may also be ordered in an optimal sequence in the CPU *execution pipeline*, for highest throughput of instructions.

Arbitrary precision implementations may also adapt themselves, depending on the size of the input. As we have seen from the tables, in this chapter, describing the execution times for the different implementations, they outperform each other in different ranges of input sizes. For small digit lengths, the classical algorithm is used. When the number of digits get larger, the Karatsuba algorithm is used. Finally, for very large digit lengths, the FFT algorithm is used.

Appendix A

Source code

```
// compiler.hpp

#ifndef COMPILER_HPP_INCLUDED
#define COMPILER_HPP_INCLUDED

// Optimize library implementation for release version
#ifdef _DEBUG
#  define _SECURE_SCL 1
#else
#  define _SECURE_SCL 0
#endif

#endif // COMPILER_HPP_INCLUDED


// measure.hpp

#ifndef MEASURE_HPP_INCLUDED
#define MEASURE_HPP_INCLUDED

#include "compiler.hpp"

#include <iostream>
#include <ctime>

#include "misc.hpp"

namespace measure {

template<typename Op>
```

```
double duration(Op& op, int clock_mult = 1)
{
  typedef misc::uint32_t uint32_t;

  clock_t t0 = clock();
  clock_t t1 = t0;
  uint32_t c = 0;
  while (t1 < t0 + clock_mult * CLOCKS_PER_SEC) {
    op();
    ++c;
    t1 = clock();
  }
  return (t1 - t0 + 0.0) / c / CLOCKS_PER_SEC;
}

}   // namespace measure

#endif // MEASURE_HPP_INCLUDED

// misc.hpp

#ifndef MISC_HPP_INCLUDED
#define MISC_HPP_INCLUDED

#include "compiler.hpp"

#include <exception>
#include <fstream>
#include <string>

#include <boost/lexical_cast.hpp>

namespace misc {

typedef unsigned int    uint32_t;
typedef unsigned long long uint64_t;

const double pi = 3.14159265358979323846264338327950;
const char pi_digits_file[] = "pi-digits-4m.txt";
const char e_digits_file[]  = "e-digits-2m.txt";

template<typename Iter>
void propagate_carry(Iter first, Iter last, typename std::iterator_traits<Iter>::value_type base)
```

```
{
  typename std::iterator_traits<Iter>::value_type c = 0;

  while (first != last) {
    *first += c;
    c = *first / base;
    *first -= c * base;
    ++first;
  }
  if (c != 0) throw std::runtime_error("overflow");
}

template<typename Iter>
void read_digits(Iter first, Iter last, std::istream& is)
{
  char c;
  while (first != last && is >> c) {
    *first++ = c - '0';
  }
  if (first != last || !is)
    throw std::runtime_error(std::string("failed to read from stream"));
}

template<typename Iter>
void read_digits(Iter first, Iter last, const char* fname)
{
  std::ifstream is(fname);
  if (!is)
    throw std::runtime_error(std::string("failed to open file: '") + fname + '\'');

  read_digits(first, last, is);
}

template<typename Iter>
void write_digits(std::ostream& os, Iter first, Iter last)
{
  while (first != last && os) {
    char c = static_cast<char>(*first) + '0';
    os << c;
    ++first;
  }
  if (first != last)
    throw std::runtime_error("failed to write to stream");
}

template<typename T>
std::string digits_filename(T digits)
{
  return std::string("digits-") + boost::lexical_cast<std::string>(digits) + ".txt";
}

template<typename Iter>
void check_result(Iter first, Iter last)
{
  typename std::iterator_traits<Iter>::difference_type digits = std::distance(first, last);
  std::ifstream is(digits_filename(digits).c_str());

  if (!is)
    throw std::runtime_error(std::string("failed to open digits file:") +
    digits_filename(digits));

  char c;
  while (is >> c && first != last) {
    if (c - '0' != *first)
      throw std::runtime_error("check_result found a digit mismatch");
      ++first;
  }
  if (first != last)
    throw std::runtime_error("check_result got digit underflow");
}

void generate_testdigits(uint32_t digits);

}   // namespace misc

#endif // MISC_HPP_INCLUDED


// classic.hpp

#ifndef CLASSIC_HPP_INCLUDED
#define CLASSIC_HPP_INCLUDED

#include "compiler.hpp"

#include <algorithm>
```

```
#include <iterator>

namespace classic {

template<typename ForwardIterator1, typename ForwardIterator2>
void multiply(ForwardIterator1 op1first, ForwardIterator1 op1last,
              ForwardIterator1 op2first, ForwardIterator1 op2last,
              ForwardIterator2 result)
{
  std::fill_n(result, std::distance(op1first, op1last) + std::distance(op2first, op2last),
  typename std::iterator_traits<ForwardIterator2>::value_type());

  for (; op1first != op1last; ++op1first, ++result) {
    ForwardIterator2 res = result;
    for (ForwardIterator1 first = op2first; first != op2last; ++first, ++res) {
      *res += *op1first * *first;
    }
  }
}

} // namespace classic


// karatsuba.hpp

#ifndef KARATSUBA_HPP_INCLUDED
#define KARATSUBA_HPP_INCLUDED

#include "compiler.hpp"

#include "classic.hpp"

namespace karatsuba {

const int threshold = 16;

template<typename RandomAccessIterator, typename T>
void multiply(RandomAccessIterator a,
              RandomAccessIterator b,
              RandomAccessIterator ret, T d)
{
  const T d_half = d / 2;
  const T d5     = d * 5;
  RandomAccessIterator alo = a, ahi = a + d_half;
  RandomAccessIterator blo = b, bhi = b + d_half;
  RandomAccessIterator asum = ret + d5;
  RandomAccessIterator bsum = asum + d_half;
  RandomAccessIterator x1 = ret;
  RandomAccessIterator x2 = ret + d;
  RandomAccessIterator x3 = x2 + d;

  if (d <= threshold) {
    classic::multiply(a, a + d, b, b + d, ret);
    return;
  }

  for (T i = 0; i < d_half; ++i) {
    asum[i] = ahi[i] + alo[i];
    bsum[i] = bhi[i] + blo[i];
  }

  multiply(alo, blo, x1, d_half);
  multiply(ahi, bhi, x2, d_half);
  multiply(asum, bsum, x3, d_half);

  for (T i = 0; i < d; ++i)
    x3[i] = x3[i] - x1[i] - x2[i];

  for (T i = 0; i < d; ++i)
    ret[i + d_half] += x3[i];
}

} // namespace karatsuba

#endif // KARATSUBA_HPP_INCLUDED


// fourier.hpp

#ifndef FOURIER_HPP_INCLUDED
#define FOURIER_HPP_INCLUDED

#include "compiler.hpp"

#include <algorithm>
```

```
#include <iterator>

namespace fourier {
namespace fourier_detail {

template<typename RandomAccessIterator>
void scramble(RandomAccessIterator first,
              typename std::iterator_traits<RandomAccessIterator>::difference_type n)
{
  typedef typename std::iterator_traits<RandomAccessIterator>::difference_type size_type;

  const size_type n_half = n / 2;

  size_type j = 0;
  for (size_type i = 0; i < n - 1; ++i) {
    if (i < j)
      std::swap(first[i], first[j]);
    size_type k = n_half;
    while (k <= j) {
      j -= k;
      k /= 2;
    }
    j += k;
  }
}

}   // namespace fourier_detail
}   // namespace fourier

namespace fourier {

template<typename RandomAccessIterator, typename Comp>
void cooley_tukey(RandomAccessIterator first, RandomAccessIterator last, Comp comp)
{
  typedef typename std::iterator_traits<RandomAccessIterator>::difference_type size_type;
  typedef typename Comp::value_type value_type;

  const size_type n = std::distance(first, last);

  fourier_detail::scramble(first, n);

  for (size_type m = 1; m < n; m *= 2) {
    for (size_type j = 0; j < m; ++j) {
      const value_type twiddle = comp.twiddle(j, m);
      for (size_type i = j; i < n; i += (m * 2)) {
        const value_type tmp = first[i];
        first[i]     = comp.add(tmp, comp.mul(twiddle, first[i + m]));
        first[i + m] = comp.sub(tmp, comp.mul(twiddle, first[i + m]));
      }
    }
  }
}

}   // namespace fourier

#endif  // FOURIER_HPP_INCLUDED


// misc.cpp

#include "classic.hpp"

#include <iostream>
#include <vector>

#include "misc.hpp"

void misc::generate_testdigits(uint32_t digits)
{
  typedef uint32_t size_type;

  std::vector<size_type> a(digits); // Operand 1
  std::vector<size_type> b(digits); // Operand 2
  std::vector<size_type> r(2 * digits); // Result = Operand1 * Operand2

  read_digits(a.begin(), a.end(), pi_digits_file);
  read_digits(b.begin(), b.end(), e_digits_file);

  std::cout << "Calculating result using " << digits << " digits\r";
  classic::multiply(a.begin(), a.end(), b.begin(), b.end(), r.begin());
  std::cout << digits << " digits ready.                                   \n";
  misc::propagate_carry(r.begin(), r.end(), 10);

  std::ofstream os(digits_filename(2 * digits).c_str());

  if (!os)
```

```
    std::runtime_error("failed to open result file for writing");

  write_digits(os, r.begin(), r.end());
}


// test_classic.cpp

#include "compiler.hpp"

#include <iostream>
#include <vector>

#include "classic.hpp"
#include "measure.hpp"

class classic_multiply {
public:
  typedef misc::uint32_t size_type;

  explicit classic_multiply(size_type digits) :
    a(digits), b(digits), r(2 * digits) {

    misc::read_digits(a.begin(), a.end(), misc::pi_digits_file);
    misc::read_digits(b.begin(), b.end(), misc::e_digits_file);
  }

  void operator()() {
    classic::multiply(a.begin(), a.end(), b.begin(), b.end(), r.begin());
  }

  void check_result() {
    misc::propagate_carry(r.begin(), r.end(), 10);
    misc::check_result(r.begin(), r.end());
  }

private:
  std::vector<size_type> a, b, r;
};

void test_classic()
{
  for (misc::uint32_t i = 3; i <= 20; ++i) {
    classic_multiply classic(1 << i);

    double d = measure::duration(classic);
    std::cout << "Classic: " << d << " s.\n";

    classic.check_result();
  }
}


// test_karatsuba.cpp

#include "compiler.hpp"

#include <iostream>
#include <vector>

#include "karatsuba.hpp"
#include "measure.hpp"

class karatsuba_multiply {
public:
  typedef misc::uint32_t size_type;

  explicit karatsuba_multiply(size_type digits) :
    a(digits), b(digits), r(6 * digits) {

    misc::read_digits(a.begin(), a.end(), misc::pi_digits_file);
    misc::read_digits(b.begin(), b.end(), misc::e_digits_file);
  }

  void operator()() {
    karatsuba::multiply(a.begin(), b.begin(), r.begin(), a.size());
  }

  void check_result() {
    r.resize(2 * a.size());
    misc::propagate_carry(r.begin(), r.end(), 10);
    misc::check_result(r.begin(), r.end());
  }

private:
  std::vector<size_type> a, b, r;
```

```
};

void test_karatsuba()
{
  for (misc::uint32_t i = 3; i <= 20; ++i) {
    karatsuba_multiply karatsuba(1 << i);

    double d = measure::duration(karatsuba);
    std::cout << "Karatsuba: " << d << " s.\n";

    karatsuba.check_result();
  }
}


// test_cooley_tukey.cpp

#include "compiler.hpp"

#include <iostream>
#include <complex>
#include <vector>

#include "fourier.hpp"
#include "measure.hpp"

template<int S>
struct compositions {
  typedef std::complex<double> value_type;

  static std::complex<double> add(const std::complex<double>& a,
                                  const std::complex<double>& b) {
    return a + b;
  }

  static std::complex<double> sub(const std::complex<double>& a,
                                  const std::complex<double>& b) {
    return a - b;
  }

  static std::complex<double> mul(const std::complex<double>& a,
                                  const std::complex<double>& b) {
    return a * b;
  }

  template<typename T>
  std::complex<double> twiddle(T j, T m) const {
    const std::complex<double>::value_type arg = S * misc::pi * j / m;
    return std::complex<double>(std::cos(arg), std::sin(arg));
  }
};

class cooley_tukey_multiplication {
public:
  typedef misc::uint32_t size_type;

  explicit cooley_tukey_multiplication(size_type digits) : d(digits),
    ao(2 * digits), bo(2 * digits), a(2 * digits), b(2 * digits),
    c(2 * digits), r(2 * digits) {

    misc::read_digits(ao.begin(), ao.begin() + digits, misc::pi_digits_file);
    misc::read_digits(bo.begin(), bo.begin() + digits, misc::e_digits_file);
  }

  void operator()() {
    // Setup for multiplication (needed as the algorithm is in-place
    // and we will call this function several times)

    std::fill(a.begin(), a.end(), std::complex<double>());
    std::fill(b.begin(), b.end(), std::complex<double>());
    std::copy(ao.begin(), ao.begin() + d, a.begin());
    std::copy(bo.begin(), bo.begin() + d, b.begin());

    compositions<1>  forward_transformation;
    compositions<-1> inverse_transformation;

    // Forward FFT
    fourier::cooley_tukey(a.begin(), a.end(), forward_transformation);
    fourier::cooley_tukey(b.begin(), b.end(), forward_transformation);

    // Pointwise multiplication
    for (size_type i = 0; i < a.size(); ++i)
      c[i] = forward_transformation.mul(a[i], b[i]);

    // Inverse FFT
    fourier::cooley_tukey(c.begin(), c.end(), inverse_transformation);
    for (size_type i = 0; i < c.size(); ++i)
      r[i] = static_cast<size_type>((1.0 / r.size() * c[i].real()) + 0.5);
  }

  void check_result() {
    misc::propagate_carry(r.begin(), r.end(), 10);
    misc::check_result(r.begin(), r.end());
  }

private:
  size_type d;
  std::vector<std::complex<double> > ao, bo;
  std::vector<std::complex<double> > a, b, c;
  std::vector<size_type>             r;
};

void test_cooley_tukey()
{
  for (misc::uint32_t i = 3; i <= 20; ++i) {
    cooley_tukey_multiplication cooley_tukey(1 << i);

    double d = measure::duration(cooley_tukey);
    std::cout << "Cooley-Tukey: " << d << " s.\n";

    cooley_tukey.check_result();
  }
}


// test_cooley_tukey_z257.cpp

#include "compiler.hpp"

#include <iostream>
#include <cassert>
#include <sstream>
#include <cstdlib>
#include <ctime>

#include "classic.hpp"
#include "fourier.hpp"
```

```
// Calculate base^exponent mod modulus
int exp(int base, int exponent, int modulus)
{
  if (base == 0 || exponent == modulus - 1)
    return 0;

  int result = 1;
  for (int i = 0; i < exponent; ++i)
    result = (result * base) % modulus;
  return result;
}

// Find n such that element = base^n
int log(int element, int base, int modulus)
{
  assert(element >= 0 && element < modulus);

  if (element == 0)
    return modulus - 1;

  for (int i = 0; i < modulus; ++i) {
    if (element == exp(base, i, modulus))
      return i;
  }
  // Shouldn't get here
  assert(false);
  std::ostringstream os;
  os << "log failed. element=" << element;
  throw std::runtime_error(os.str());
}

// Find a such that element * a mod modulus = 1
int multiplicative_inv(int element, int modulus)
{
  assert(element >= 0 && element < modulus);

  for (int i = 1; i < modulus; ++i) {
    if ((element * i) % modulus == 1)
      return i;
  }
  // Shouldn't get here
  assert(false);
  std::ostringstream os;
  os << "multiplicative inverse failed. element=" << element;
  throw std::runtime_error(os.str());
}

// Find a such that element + a mod modulus = 0
int additive_inv(int element, int modulus)
{
  assert(element >= 0 && element < modulus);

  for (int i = 0; i < modulus; ++i) {
    if ((element + i) % modulus == 0)
      return i;
  }
  // Shouldn't get here
  assert(false);
  std::ostringstream os;
  os << "additive inverse failed. element=" << element;
  throw std::runtime_error(os.str());
}

// Find n such that generator^n = (1 + generator^exponent)
int zech(int exponent, int generator, int modulus)
{
  if (exponent == modulus - 1)
    return 0;

  int a = exp(generator, exponent, modulus);
  ++a;
  if (a == modulus)
    return modulus - 1;

  int b = log(a, generator, modulus);
  return b;
}

// Calculate Zech's logarithm
template<typename T, T Generator, T Modulus>
const T* precalc_zech()
{
  static T table[Modulus] = {};

  for (T i = 0; i < Modulus; ++i) {
    const T e = log(i, Generator, Modulus);
    table[e] = zech(e, Generator, Modulus);
  }
  return table;
}

// Calculate "twiddle" factor for use by Cooley-Tukey FFT
template<typename T, T Size, T Principal, T Modulus>
const T (*precalc_twiddle())[Size / 2]
{
  static T table[Size / 2 + 1][Size / 2] = {};

  for (T m = 1; m < Size; m *= 2) {
    for (T j = 0; j < m; ++j) {
      table[m][j] = (Size / (2 * m) * j * Principal) & (Modulus - 2);
    }
  }
  return table;
}

// Calculate additive inverse
template<typename T, T Generator, T Modulus>
const T* precalc_addinv_exp()
{
  static T table[Modulus] = {};

  for (T i = 0; i < Modulus; ++i) {
    table[log(i, Generator, Modulus)] = log(additive_inv(i, Modulus), Generator, Modulus);
  }
  return table;
}

// Encapsulate modulo arithmetic. To be used by classic multiplication
template<typename T, T M>
class zmod {
public:

  zmod(T value = T()) : v(value) {}

  zmod& operator+=(const zmod& other) {
    assert(is_member(v) && is_member(other.v));

    v = (v + other.v) % M;
    assert(is_member(v));
    return *this;
  }

  zmod operator*(const zmod& other) const {
    assert(is_member(v) && is_member(other.v));
    return zmod((v * other.v) % M);
  }

  T value() const {
    return v;
  }

private:
  bool is_member(T m) const {
    return m >= 0 && m < M;
  }

  T v;
};

// Encapsulate arithmetic based on exponents for a given Generator
template<typename T, T Modulus, T Generator, T Size, T Principal, bool Forward>
class zmod_exp {
public:

  typedef T value_type;

  zmod_exp() {
    addinv_table  = precalc_addinv_exp<T, Generator, Modulus>();
    zech_table    = precalc_zech<T, Generator, Modulus>();
    twiddle_table = precalc_twiddle<T, Size, Principal, Modulus>();
  }

  T add(T a, T b) const {
    assert(a >= 0 && a <= Modulus - 1);
    assert(b >= 0 && b <= Modulus - 1);

    if (a == Modulus - 1)
      return b;
    if (b == Modulus - 1)
      return a;
    const T d = b >= a ? b - a : b + Modulus - 1 - a;
    return mul(a, zech_table[d]);
  }

  T sub(T a, T b) const {
    assert(a >= 0 && a <= Modulus - 1);
    assert(b >= 0 && b <= Modulus - 1);

    return add(a, addinv_table[b]);
  }

  T mul(T a, T b) const {
    assert(a >= 0 && a <= Modulus - 1);
    assert(b >= 0 && b <= Modulus - 1);

    if (a == Modulus - 1 || b == Modulus - 1)
      return Modulus - 1;
    a += b;
    return a < Modulus - 1 ? a : a - (Modulus - 1);
  }

  T twiddle(T j, T m) const {
    return exp_sign<Forward>::value(twiddle_table[m][j]);
  }

private:
  template<bool B>
  struct exp_sign {
    static T value(T t) {
      return t;
    }
  };

  template<>
  struct exp_sign<false> {
    static T value(T t) {
      return t == 0 ? 0 : Modulus - 1 - t;
    }
  };

  const T* addinv_table;
  const T* zech_table;
  const T (*twiddle_table)[Size / 2];
```

```
};

void test_cooley_tukey_z257()
{
  const int size      = 128;   // 2, 4, ..., 64, 128
  const int modulus   = 257;
  const int generator = 3;
  const int princroot = (modulus - 1) / (2 * size);
  const int size_inv  = log(multiplicative_inv(2 * size, modulus), generator, modulus);


  // Multiplication using classic algorithm

  typedef zmod<int, modulus> zm;

  zm a1[size] = {};
  zm b1[size] = {};
  zm r1[2 * size] = {};

  // Create operands with "random" values
  srand(time(0));
  for (int i = 0; i < size; ++i) {
    a1[i] = rand() % modulus;
    b1[size - i - 1] = rand() % modulus;
  }

  // Do the classic multiplication
  classic::multiply(a1, a1 + size, b1, b1 + size, r1);

  // Multiplication using Cooley-Tukey algorithm

  int a2[2 * size] = {};
  int b2[2 * size] = {};

  // Use the same values as for classic multiplication
  for (int i = 0; i < size; ++i) {
    a2[i] = a1[i].value();
    b2[i] = b1[i].value();
  }

  // Convert to exponential representation
  for (int i = 0; i < 2 * size; ++i) {
    a2[i] = log(a2[i], generator, modulus);
    b2[i] = log(b2[i], generator, modulus);
  }

  // Create our transformation objects (forward and inverse)
  zmod_exp<int, modulus, generator, 2 * size, princroot, true>  forward_transform;
  zmod_exp<int, modulus, generator, 2 * size, princroot, false> inverse_transform;

  // Forward transformations
  fourier::cooley_tukey(a2, a2 + 2 * size, forward_transform);
  fourier::cooley_tukey(b2, b2 + 2 * size, forward_transform);

  // Pointwise multiplication
  for (int i = 0; i < 2 * size; ++i) {
    a2[i] = forward_transform.mul(a2[i], b2[i]);
  }

  // Inverse transformation
  fourier::cooley_tukey(a2, a2 + 2 * size, inverse_transform);

  // Multiply each element with inverse of transformation length (2 * size)
  for (int i = 0; i < 2 * size; ++i) {
    a2[i] = inverse_transform.mul(size_inv, a2[i]);
  }

  // Convert back to non-exponential representation
  for (int i = 0; i < 2 * size; ++i) {
    a2[i] = exp(generator, a2[i], modulus);
  }

  // Check if the result matches classical multiplication
  for (int i = 0; i < 2 * size; ++i) {
    assert(r1[i].value() == a2[i]);
    if (r1[i].value() != a2[i]) {
      throw std::runtime_error("multiplication results mismatch");
    }
  }
}


// main.cpp

#include <exception>
#include <iostream>
```

```
extern void test_classic();
extern void test_karatsuba();
extern void test_cooley_tukey();
extern void test_cooley_tukey_z257();

int main()
{
  try {
    test_classic();
    test_karatsuba();
    test_cooley_tukey();
    test_cooley_tukey_z257();
  }
  catch (const std::exception& ex) {
    std::cout << "Exception: " << ex.what() << '\n';
  }
}
```

Bibliography

[1] *International Standard ISO/IEC 14882:2003 Programming languages – C++*. American National Standards Institute, 25 West 43rd Street, New York, New York 10036, second edition, 2003.

[2] John A. Beachy and William D. Blair. *Abstract Algebra.* Waveland Press, Inc., third edition, 2006.

[3] Randal E. Bryant and David O'Hallaron. *Computer Systems - A Programmer's Perspective.* Pearson Education Internation, Prentice Hall, 2006.

[4] Carl Burch. Karatsuba multiplication – `http://ozark.hendrix.edu/~burch/proj/karat/index.html`, 1999.

[5] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[6] R. Crandall and C. Pomerance. *Prime Numbers–a Computational Approach.* Springer, New York, second edition, 2005.

[7] A. Karatsuba and Yu Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962.

[8] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, third edition, 1997.

[9] Per-Anders Svensson. *Abstrakt Algebra.* Studentlitteratur, 2005.

[10] Wikipedia. Cooley-tukey fft algorithm — wikipedia, the free encyclopedia, 2008. [Online; accessed 5-March-2008].

[11] Wikipedia. Master theorem — wikipedia, the free encyclopedia, 2008. [Online; accessed 5-March-2008].

[12] Wikipedia. Toom-cook multiplication — wikipedia, the free encyclopedia, 2009. [Online; accessed 18-January-2009].