1991

Fast multiplication of multiple-precision integers

Sonja Benz

Rochester Institute of Technology
Department of Computer Science

Fast Multiplication of Multiple-Precision Integers

by
Sonja Benz

A thesis, submitted to
The Faculty of the Department of Computer Science,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Professor S. Radziszowski

Professor P. Anderson

Professor A. Kitchen

May 30, 1991

06/04/1991

Title of Thesis:

Multiplication of
multiple-precision integers.

I, Sonja Benz, hereby grant permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Contents

Abstract

Multiple-precision multiplication algorithms are of fundamental interest for both theoretical and practical reasons. The conventional method requires $O(n^2)$ bit operations whereas the fastest known multiplication algorithm is of order $O(n \log n \log\log n)$. The price that has to be paid for the increase in speed is a much more sophisticated theory and programming code.

This work presents an extensive study of the best known multiple-precision multiplication algorithms. Different algorithms are implemented in C, their performance is analyzed in detail and compared to each other. The break even points, which are essential for the selection of the fastest algorithm for a particular task, are determined for a given hardware software combination.

Chapter 1

Introduction

The multiplication operation, which except for addition is the most elementary arithmetic operation, is of fundamental interest for both purely theoretical and practical reasons. The well-known classical method achieves $O(n^2)$ bit complexity [1]. While the performance of this method is satisfactory for small numbers, the application for large numbers is restricted by an unfavorable asymptotic behavior.

The common and intuitive opinion that the school method can't be improved significantly was disproved in 1962 by A. KARATSUBA, who suggested a faster algorithm with complexity $O(n^{\log_2 3})$ [2, p.62] [15, p.278]. He obtained this improvement by employing simple algebraic identities and a recursive divide and conquer approach. By 1966 A. L. TOOM and S. A. COOK discovered another asymptotically fast algorithm with complexity $O(n2^{\alpha\sqrt{\log_2 n}})$ for some constant α [15, p.286]. They took advantage of the similarity of integer and polynomial arithmetic and interpreted multiplication of integers as multiplication of polynomials.

The best known upper bound for multiplication is $O(n \log n \log\log n)$ and is achieved by an algorithm of A. SCHÖNHAGE and V. STRASSEN [32] [5]. SCHÖNHAGE and STRASSEN used the connection between multiplication of natural numbers and polynomial multiplication and also employed the fast Fourier transform.

This upper bound applies to Turing machines, i.e. to a serial machine model with a bounded central high speed memory and a limited word size. This

[1] See section 2.2 for the definition of "order O" and "bit complexity".

machine is believed to be a realistic model of standard computers. Attempts to estimate complexity of integer multiplication must refer to a specific computational model. For a random access machine (RAM), whose registers are capable of holding integers of arbitrary size, the upper bound drops to $O(n \log n)$ [15, p.295]. A pointer machine (also called "storage modification machine" or "linking automata") can carry out an n-bit multiplication in $O(n)$ steps [30] [15, p.295].

It is a strange phenomenon that although asymptotically fast algorithms have been known for decades, the multiple-precision software still uses the standard procedures [28], R. P. BRENT, for example, who wrote an extensive Fortran multiple-precision floating-point arithmetic package, argued that he used the classical $O(n^2)$ method because "it seems difficult to implement faster algorithms in a machine independent manner in Fortran" [7]. There seems to be a general consensus that the SCHÖNHAGE-STRASSEN algorithm is "hard to implement" [5] [29]. BORODIN even states: "However, it is not at all clear that the method could ever be practically implemented" [4]. Actually, the SCHÖNHAGE-STRASSEN algorithm was originally interpreted as a theoretical result without any practical impact [31] [29]. Most practically oriented papers and books recommend the use of a simplified version of the algorithm [4] [19] [5].

The study of multiplication algorithms undoubtedly is interesting by itself and for purely scientific reasons. On the other hand, there is also a great demand for fast multiple-precision algorithms in a vast range of applications. In addition to typical applications like primality tests [21] [22] [24] and high-precision computations of constants [27] [8] [5] [11], the propagation of progessive cryptological methods [13] [26] has increased the interest in fast multiplication algorithms. Furthermore, integer multiplication mod $(2^n + 1)$ for values like $n = 800$, $n = 64000$ or even $n = 512000$ is of practical interest for the application of a fast method to compute approximately the zeros of complex polynomials [28] [31]. Other applications can be found in [6] [3] [5].

This work presents an extensive study of the best known multiple-precision multiplication algorithms. Different algorithms - the classical school method, the KARATSUBA algorithm and variations of the SCHÖNHAGE-STRASSEN algorithm - are implemented in the programming language C and tested on an INTEL 80386 machine. The performance of the multiplication algorithms is analyzed in detail, and the break even points are determined. The simplicity of the classical method produces only a small overhead when implemented. Therefore, this algorithm is expected to be the best qualified for

"small" multiple-precision numbers. The more sophisticated multiplication algorithms necessarily involve an increased amount of programming code. The enlarged overhead competes with the improved asymptotic behavior of the algorithms, and it is of special interest to specify the fastest algorithm for a given size of the factors.

Chapter 2

Theoretical Background

2.1 Preliminary Remarks

This chapter develops the theoretical concepts of five multiplication algorithms,

- the classical multiplication algorithm
- the KARATSUBA algorithm
- the ordinary fast convolution algorithm
- the original SCHÖNHAGE-STRASSEN algorithm (1971)
- the modified SCHÖNHAGE-STRASSEN algorithm (1982).

Thereby the ordinary fast convolution algorithm and the SCHÖNHAGE-STRASSEN algorithms from 1971 and 1982 are so-called "fast convolution algorithms".

Section 2.2 introduces fundamental terminology and notation. In section 2.3 the classical, and in section 2.4 the KARATSUBA multiplication algorithm is developed.

Section 2.5 is concerned with the much more complex fast convolution algorithms. After describing the general underlying indea in section 2.5.1 the mathematical background is unfolded in sections 2.5.2 2.5.6: The backbone of the fast convolution algorithms is the Fourier transform (section

2.5.2 and 2.5.4) and the convolution theorem (section 2.5.3). Common to all of the three fast convolution algorithms is the application of the "mod m fast Fourier transfrom" (section 2.5.5) which works in a finite field or ring Z_m of integers with arithmetic carried out modulo an integer m. In addition, the SCHÖNHAGE-STRASSEN algorithm from 1971 employs the Chinese remainder theorem which is introduced in section 2.5.6. Finally, sections 2.5.7, 2.5.8 and 2.5.9 develop the concrete individual algorithms, the fast ordinary convolution algorithm, the SCHÖNHAGE-STRASSEN algorithm (1971) and the modified SCHÖNHAGE-STRASSEN algorithm from 1982, respectively.

2.2 Terminology and Notation

Fundamental terminology and notation will be introduced here. When we analyze and compare multiplication algorithms we talk about time complexity. In particular, we will focus on the asymptotic time comlexity, since the size of the problems that can be solved is determined by the asymptotic behavior of a specific algorithm. If an algorithm processes inputs of size n in time $c \cdot n^2$ for some constant c, for example, the algorithm is said to be $O(n^2)$, read "order n^2".

Definition 2.1 (Order O) *A nonnegative function $g(n)$ is said to be $O(f(n))$ (read order $f(n)$) if there exists a constant c such that*

$$g(n) \leq cf(n)$$

for all but some finite (possibly empty) set of nonnegative values of n [2, p.2].

It is important to distinguish between complexity in bit operations and complexity in arithmetic operations. For example, multiplication of two n-bit integers can be done in $O(1)$ arithmetic operations but in $O(n \log n \log\log n)$ bit operations. The bitwise model takes into account, that the time of an arithmetic operation depends on the size of the operands and incorporates the so called *logarithmic cost* [1] [2] of operands. All variables are assumed to have the values 0 or 1, i.e. they are bits and the operations used are

[1] We are talking of *uniform* or *unit* cost, if we neglect any cost dependency on the length of the operands.

logical rather than arithmetic. This model is useful if we talk about basic operations, such as the arithmetic ones, which are primitive in other models. Unless stated otherwise, we will talk about bit complexity here.

Appropriate representations for large integers have to be chosen. By default all integers in this paper will be nonnegative. A positive integer can be written in radix B positional notation, with

$$u = \sum_{i=0}^{n_B-1} u_i B^i, \qquad 0 \leq u_i < B, \tag{2.1}$$

where n_B is the number of digits u_i. Well-known standard examples are the binary, octal, decimal and hexadecimal radix B representations of integers. We will write for a radix B number u

$$u = (u_{n_B-1}, \ldots, u_1, u_0)_B. \tag{2.2}$$

Occasionally, we will employ vector notation

$$u = [u_0, u_1, \ldots, u_{n_B-1}]^T, \tag{2.3}$$

with u being a length n_B column vector.

Note, that when we talked about n-bit integers and $O(n^2)$ bit complexity we should have used the terminology n_2-bit integers and $0(n_2^2)$ bit complexity instead. Often $B \neq 2$ and thus $n_2 \neq n_B$ and we carefully have to distinguish between n_2 and n_B. Nevertheless, often the meaning of n_i is clear from the context and we will omit the index and just write n.

In section 2.5 it will be shown that the critical problem in finding fast multiplication algorithms is to speed up the *convolution* of digits of two factors u and v.

Definition 2.1 (Convolution) *Let R be an arbitrary commutative ring R. Assume $u = [u_0, ..., u_{n-1}]^T$ and $v = [v_0, ..., v_{n-1}]^T$ are two column vectors with $u_i, v_i \in R$. Then the* convolution *of u and v, denoted $u \otimes v$, is the vector $c = [c_0, ..., c_{2n-1}]^T$, where*

$$c_i = \sum_{j=0}^{n-1} u_j v_{i-j}$$

($u_i = v_i = 0$ if $i < 0$ or $i \geq n$).

Thus

$$\begin{aligned} c_0 &= u_0v_0 \\ c_1 &= u_0v_1 + u_1v_0 \\ c_2 &= u_0v_2 + u_1v_1 + u_2v_0 \quad , \qquad \cdots \\ c_{2n-1} &= 0. \end{aligned}$$

2.3 Classical n^2 Multiplication Algorithm

The crucial point in multiple-precision arithmetic operations for computers is that long numbers may be written in radix B notation, where $B = 2^{word\ size}$ for a given word size of a computer. The term *n-digit integer* then means any integer less than B^n. Multiple-precision operations can be constructed on top of fundamental operations for single digits which are supplied in form of single- or double-precision operations by standard computers.

Algorithm C *(Classical multiplication)*

Given nonnegative integers $(u_{n-1}, ..., u_0)_B$ and $(v_{m-1}, ..., v_0)_B$, this algorithm forms their radix-B product $(w_{m+n-1}, ..., w_0)_B$. (The conventional pencil-and-paper method is based on calculating the partial products $(u_{n-1}, ..., u_0)_B \times v_j$ first, for $0 \leq j < m$, and then adding these products together with appropriate scaling factors; but in a computer it is best to do the addition concurrently with the multiplication, as described in this algorithm.) According to [15, p.253] the outline is as follows:

C1 [Initialize.] Set $w_0, \ldots, w_{m+n-1}$ all to zero. Set $j \leftarrow 0$. (If $w_0, ..., w_{m+n-1}$ were not cleared to zero the steps below would set

$$(w_{m+n-1}, ..., w_0)_B \leftarrow (u_{n-1}, ..., u_0)_B \times (v_{m-1}, ..., v_0)_B + (w_{m+n-1}, ..., w_0)_B.$$

This more general operation is sometimes useful.)

C2 [Zero multiplier?] If $v_j = 0$, go to step C6. (This test saves a good deal of time if there is a reasonable chance that v_j is zero, but otherwise it may be omitted without affecting the validity of the algorithm.)

C3 [Initialize i.] Set $i \leftarrow 0, k \leftarrow 0$.

C4 [Multiply and add.] Set $t \leftarrow u_i \times v_j + w_{i+j} + k$; then set $w_{i+j} \leftarrow t \bmod B$ and $k \leftarrow \lfloor t/B \rfloor$. (Here the "carry" k will always be in the range $0 \leq k < B$; see below).

C5 [Loop on i.] Increase i by one. Now if $i < n$, go back to step C4; otherwise set $w_{i+j} \leftarrow k$.

C6 [Loop on j.] Increase j by one. Now if $j < m$, go back to step C2; otherwise the algorithm terminates.

The critical points for an efficient implementation of the algorithm are the two inequalities

$$0 \leq t < B^2, \qquad 0 \leq k < B,$$

since they determine the size of the register needed for the calculation on step C4. By induction, and for $k < B$ at the start of C4, it can be proved that

$$u_i \times v_i + w_{i+j} + k \leq (B-1) \times (B-1) + (B-1) + (B-1) = B^2 - 1 < B^2.$$

2.4 KARATSUBA's Algorithm

The KARATSUBA algorithm in its original form is described in [2, p.62]. A similar but more simplified method can be found in [15, 4.3.3] and is adopted in this paper.

Let u and v be two $2n$-bit numbers $(u_{2n-1}, u_{2n-2}, ..., u_0)_2$ and $(v_{2n-1}, v_{2n-2}, ..., v_0)_2$. If we partition u and v into two halves, we can write

$$u = 2^n U_1 + U_0, \quad v = 2^n V_1 + V_0,$$

where $U_1 = (u_{2n-1}, ..., u_n)_2$ is the "most significant half" of the number u and $U_0 = (u_{n-1}, ..., u_0)_2$ the "least significant half"; similarly $V_1 = (v_{2n-1}, ..., v_n)_2$ and $V_0 = (v_{n-1}, ..., v_0)_2$. Now the product uv can be computed as

$$uv = (2^{2n} + 2^n)U_1V_1 + 2^n(U_1 - U_0)(V_0 - V_1) + (2^n + 1)U_0V_0.$$

Thus the problem of multiplying $2n$-bit numbers is reduced to three multiplications of n-bit numbers, namely U_1V_1, $(U_1 - U_0)(V_0 - V_1)$, U_0V_0, plus some simple shifting and adding operations. By applying the multiplication formula recursively to evaluate the three products (U_1V_1, $(U_1 - U_0)(V_0 - V_1)$, U_0V_0) KARATSUBA's algorithm achieves a bit complexity of $O(n^{\log_2 3})$, with $log_2 3 = 1.58.. < 2$.

2.5 Fast Convolution Algorithms

2.5.1 General Idea

The crucial point in multiple-precision multiplication is its close relationship to polynomial multiplication. The conventional positional notation for integers is essentially a polynomial-based representation. If $u = (u_{n-1}, ..., u_0)_B$ is a radix B integer, then u represents the value of its associated polynomial

$$u(B) = \sum_{i=0}^{n-1} u_i B^i \qquad 0 \leq u_i < B.$$

Let us consider an example. Let $B = 10$ and $u = 314$, $v = 293$. Then

$$\begin{aligned} u(B) &= 3B^2 + 1B + 4 = 300 + 10 + 4 \\ v(B) &= 2B^2 + 9B + 3 = 200 + 90 + 3. \end{aligned}$$

The integer product $w = u \cdot v = 92002$ can also be obtained by multiplying the two polynomials $u(B)$ and $v(B)$, getting $w(B) = u(B)v(B)$, and evaluating $w(10)$.

$$\begin{aligned} w(B) &= 6B^4 + 29B^3 + 26B^2 + 39B + 12 \\ w(10) &= 92002. \end{aligned}$$

Thus, the critical problem is to find fast algorithms to speed up the convolution of digits [2] of the two factors u und v from which the result is reconstructed, called fast convolution agorithms. Fast convolution algorithms are always based on some form of fast Fourier transform.

2.5.2 Discrete Fourier Transform

The Fourier transform requires the existence of a so-called *primitive n-th root of unity*. The definition of a primitive n-th root of unity is given for an arbitrary ring R, although the discrete Fourier transform will be defined in a field F in this section [3] .

[2] See section 2.2 for the introductory definition of "convolution", and section 2.5.3 for more details.

[3] For simplicity both the discrete Fourier transform and the fast Fourier transform (section 2.5.4) will be defined in a field F. However, the definition may be extended to an arbitrary commutative ring [2]. In section 2.5.5 the Fourier transform in the ring Z_m of integers modulo m will be considered in some detail.

Definition 2.1 (Primitive n-th root of unity) *Given a commutative ring $(R, +, \cdot, 0, 1)$, an element ω_n of R such that*

1. $\omega_n^n = 1$
2. $\omega_n^k \neq 1, \quad$ *for* $1 \leq k < n$

is said to be a primitive n-th root of unity.

Equivalently we say that ω_n is of order n in R.

Example 2.1

1. In Z we have for $n = 2$, $\omega_2 = -1$.
2. In Z_5 we have for $n = 4$, $\omega_4 = 2$.
3. In C we have for an arbitrary n, $\omega_n = e^{2\pi i/n}$, $i = \sqrt{-1}$.

The Fourier transform will be defined in an arbitrary field F. Let $u = [u_0, u_1, ..., u_{n-1}]^T$ be a length n (column) vector with $u_i \in F$.

Definition 2.2 (Discrete Fourier transform (DFT)) *Assume that ω_n is a primitive n-th root of unity in a field F. Then the vector*

$$FT(u) = [f_0, f_1, ..., f_{n-1}]^T$$

with

$$f_i = \sum_{j=0}^{n-1} u_j \omega_n^{ij}, \qquad for \quad i = 0, ..., n-1$$

is called the discrete Fourier transform *of u.*

Lemma 2.1 (Inverse discrete Fourier transform) *Let ω_n be a primitive n-th root of unity in a field F. Then the vector*

$$FT^{-1}(u) = [\bar{f}_0, \bar{f}_1, ..., \bar{f}_{n-1}]^T$$

with

$$\bar{f}_i = n^{-1} \sum_{j=0}^{n-1} u_j \omega_n^{-ij}, \qquad for \quad i = 0, ..., n-1$$

is called the inverse Fourier transform *of n.*

The interested reader can find the proof that $FT^{-1}(FT(u)) = u$ in [25]. The *fast Fourier transform (FFT)* is just an efficient method of calculating the discrete Fourier transform and will be introduced in section 2.5.4.

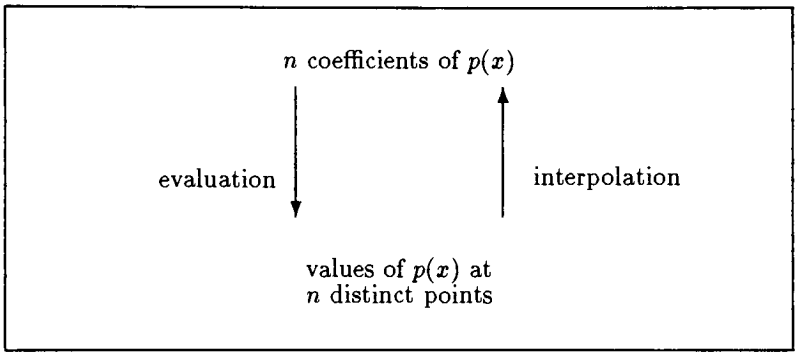
2.5.3 Application of the Fourier Transform to Convolution

In 2.5.1 we already mentioned that the conventional positional notation for integers is essentially a polynomial-based representation. Given a polynomial of degree $(n-1)$

$$p(x) = \sum_{i=0}^{n-1} u_i x^i,$$

$p(x)$ can be uniquely represented in two ways, either by a list of its coefficients $u_0, ..., u_{n-1}$ or by a list of its values at n distinct points $x_0, ..., x_{n-1}$. The process of calculating the values at different points is called *evaluation*, the inverse process of finding the coefficient representation of a polynomial given its values at points $x_0, ..., x_{n-1}$ is called *interpolation* (fig. 2.1).

Figure 2.1: Evaluation and interpolation of polynomials.



Assume $x_0, x_1, ..., x_{n-1}$ are values at $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$, with ω_n being a primitive n-th root of unity, then one can easily see that a discrete Fourier transform can be viewed as a transform from the representation of a polynomial by its coefficients to the representation by its values:

$$[1, \omega_n^{k\cdot 1}, \omega_n^{k\cdot 2}, \omega_n^{k\cdot 3}, ..., \omega_n^{k\cdot(n-1)}] \times [u_0, u_1, u_2, u_3, ..., u_{n-1}]^T =$$
$$u_0 + u_1(\omega_n^k) + u_2(\omega_n^k)^2 + u_3(\omega_n^k)^3 + ... + u_{n-1}(\omega_n^k)^{n-1}.$$

Likewise, the inverse Fourier transform is equivalent to interpolating a polynomial given its values at the n-th roots of unity.

One of the principal applications of the Fourier transform is to compute the convolution of two vectors. The definition of the convolution is given in

section 2.2. Actually, computing the convolution of two vectors is the same as computing the product of two polynomials. Let

$$p(x) = \sum_{i=0}^{n-1} u_i x^i \qquad \text{and} \qquad q(x) = \sum_{j=0}^{n-1} v_j x^j$$

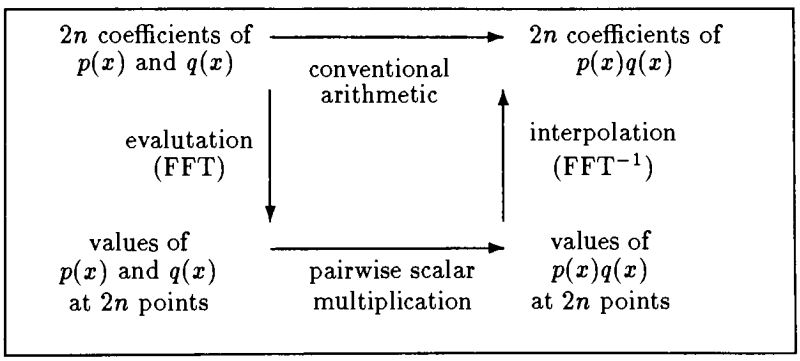
be two $(n-1)$ degree polynomials. The product is the $(2n-2)$ degree polynomial

$$p(x)q(x) = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} u_j v_{i-j} \right) x^i$$

with $u_i = v_i = 0$ if $i < 0$ or $i \geq n$. Observe that the coefficients of the product polynomial are exactly the coordinates of the convolution of the coefficient vectors $[u_0, ..., u_{n-1}]^T$ and $[v_0, ..., v_{n-1}]^T$, with $c_{2n-1} = 0$.

Since such a product polynomial can be uniquely represented by its values at $2n$ points, a new method for multiplying the n-coefficients polynomials $p(x)$ and $q(x)$ is possible. We evaluate both $p(x)$ and $q(x)$ at $2n$ selected points, then multiply together the corresponding values of the polynomials at these points, forming $2n$ products. The polynomial that uniquely fits these $2n$ values is the desired product $p(x)q(x)$ (fig. 2.2) [4].

Figure 2.2: Principle of the convolution theorem.



[4] The figure illustrates the convolution theorem (2.1) where the first n coefficients of p and q are set to zero.

Theorem 2.1 (Convolution Theorem)
Let $u = [u_0, u_1, \ldots, u_{n-1}, 0, \ldots, 0]^T$ *and* $v = [v_0, v_1, \ldots, v_{n-1}, 0, \ldots, 0]^T$ *be column vectors of length* $2n$. *Then*

$$u \otimes v = FT^{-1}(FT(u) \cdot FT(v)) \qquad \textit{[2, p.255].}$$

Since the convolution of two vectors of length n is a vector of length $2n$, a "padding" with zeros is required to achieve two $2n$-length vectors. We can avoid this "padding" with zeros if we use the so-called "wrapped" convolution.

Definition 2.2 (Negative wrapped convolution)
Let $u = [u_0, u_1, \ldots, u_{n-1}]^T$ *and* $v = [v_0, v_1, \ldots, v_{n-1}]^T$ *be column vectors of length* n. *The* negative wrapped convolution *of* u *and* v *is the vector* $d = [d_0, d_1, \ldots, d_{n-1}]^T$, *where*

$$d_i = \sum_{j=0}^{i} u_j v_{i-j} - \sum_{j=i+1}^{n-1} u_j v_{n+i-j}.$$

The negative wrapped convolution is used by the SCHÖNHAGE-STRASSEN algorithm (section 2.5.8 and 2.5.9).

Theorem 2.2 (Convolution theorem for negative wrapped convolution)
Let $u = [u_0, u_1, \ldots, u_{n-1}]^T$ *and* $v = [v_0, v_1, \ldots, v_{n-1}]^T$ *be column vectors of length* n. *Assume* $\psi^2 = \omega_n$. *Furthermore let* $d = [d_0, d_1, \ldots, d_{n-1}]^T$ *be the negative wrapped convolution of* u *and* v *and*

$$\begin{aligned} \hat{u} &= [u_0, \psi u_1, \ldots, \psi^{n-1} u_{n-1}]^T \\ \hat{v} &= [v_0, \psi v_1, \ldots, \psi^{n-1} v_{n-1}]^T. \end{aligned}$$

Then

$$\hat{d} = FT^{-1}(FT(\hat{u}) \cdot FT(\hat{v})).$$

An outline of the proof can be found in [2, p.257], the details are in [17, p.17].

2.5.4 Fast Fourier Transform

The discrete Fourier transform of a length n vector u can be computed in $O(n^2)$ time if we assume that arithmetic operations on arbitrary elements of a vector u require one step each. On the other hand, for $n = 2^k$ the fast Fourier transform achieves $O(n \log n)$ arithmetic operations, which asymptotically is the best known method.

In section 2.5.2 we called the vector $FT(u) = [f_0, f_1, \ldots, f_{n-1}]^T$ with $f_i = \sum_{j=0}^{n-1} u_j \omega_n^{ij}$ the discrete Fourier transform of u. We observed in section 2.5.3 that computing the i-th component f_i of the discrete Fourier transform is equivalent to evaluating the polynomial $p(\omega_n^i)$,

$$p(\omega_n^i) = u_0 + u_1\omega_n^i + u_2\omega_n^{2i} + \ldots + u_{n-1}\omega_n^{(n-1)i}.$$

To attain a fast algorithm for performing the Fourier transform we apply a divide and conquer method and split the problem into two subproblems, that is we replace the problem of evaluating an n-th degree polynomial by the problem of evaluation two $\frac{n}{2}$-th degree polynomials [18]. We write $p(\omega_n^i)$ as the sum of its odd- and even-power terms:

$$p(\omega_n^i) = (u_0 + u_2\omega_n^{2i} + \cdots + u_{n-2}\omega_n^{(n-2)i}) + (u_1\omega_n^i + u_3\omega_n^{3i} + \cdots + u_{n-1}\omega_n^{(n-1)i})$$

and get

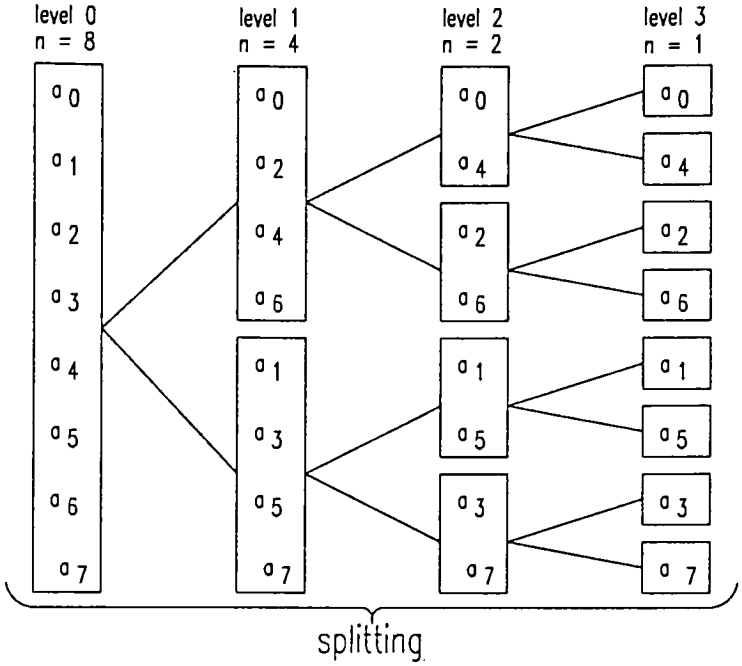
$$\begin{aligned} p(\omega_n^i) &= \textstyle\sum_{j=0}^{\frac{n}{2}-1} u_{2j}\omega_n^{2ji} + \sum_{j=0}^{\frac{n}{2}-1} u_{2j+1}\omega_n^{(2j+1)i} \\ &= s(\omega_n^{2i}) + \omega_n^i t(\omega_n^{2i}), \qquad (*) \end{aligned}$$

where s and t are polynomials of degree $\frac{n}{2}$. By using the next two lemmas this equation can be simplified significantly.

Lemma 2.1 *Let n be even and ω_n be a primitive n-th root of unity. Then ω_n^2 is a primitive $\frac{n}{2}$-th root of unity*

Proof. We will prove that corresponding to definition 2.1 ω_n^2 is a primitive $\frac{n}{2}$-th root of unity. Since $(\omega_n^2)^{\frac{n}{2}} = \omega_n^n = 1$, ω_n^2 is a $\frac{n}{2}$-th root of unity. But $(\omega_n^2)^j \neq 1$ for $0 < j < \frac{n}{2}$, otherwise we would have $\omega_n^k = 1$ with $0 < k < 2\frac{n}{2}$ in contradiction to ω_n being a primitive n-th root of unity. Thus ω_n^2 is a primitive $\frac{n}{2}$-th root of unity, as required [19, p.297]. □

Figure 2.3: Repeated splitting of a polynomial in its even and odd parts.



Lemma 2.2 *Let n be even, let ω_n be a primitive n-th root of unity. Then*

$$\omega_n^{k+\frac{n}{2}} = -\omega_n^k, \qquad \text{for } 0 \leq k < \frac{n}{2}.$$

Proof. First we note that

$$\omega_n^{k+\frac{n}{2}} = \omega_n^k \cdot \omega_n^{\frac{n}{2}}.$$

But $(\omega_n^{\frac{n}{2}})^2 = \omega_n^n = 1$, and $(\omega_n^{\frac{n}{2}})^2 = 1$ in a field implies $\omega_n^{\frac{n}{2}} = \pm 1$. Since ω_n is a primitive n-th root of unity and $1 \leq \frac{n}{2} < n$, $\omega_n^{\frac{n}{2}} \neq 1$. Hence $\omega_n^{\frac{n}{2}} = -1$ and

$$\omega_n^{k+\frac{n}{2}} = \omega_n^k \cdot \omega_n^{\frac{n}{2}} = -\omega_n^k.$$

This is the desired result. □

Applying lemma 2.2 to $(*)$ we have

$$\begin{aligned} p(\omega_n^i) &= s(\omega_n^{2i}) + \omega_n^i t(\omega_n^{2i}) \\ p(\omega_n^{i+\frac{n}{2}}) &= s(\omega_n^{2i}) - \omega_n^i t(\omega_n^{2i}), \end{aligned}$$

since $\omega_n^{i+\frac{n}{2}} = -\omega_n^i$ and $(\omega_n^{i+\frac{n}{2}})^2 = (-\omega_n^i)^2 = \omega_n^{2i}$. The problem of evaluating the polynomial $p(\omega_n^i)$ at n points reduces to the problem of evaluating two polynomials s and t at $\frac{n}{2}$ points and combining the result with the cost of $\frac{n}{2}$ scalar multiplications and n additions. For $n = 2^k$ the repeated application of the splitting algorithm reduces after $\log_2 n$ halvings to n 1-point transforms, which are trivial and of no cost at all (figure 2.3). To analyze the arithmetical complexity of the fast Fourier transform let $T(n)$ be the time required to perform an n-point Fourier transform by the above method. Then

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + n(t_{\pm} + \tfrac{t_*}{2}) \\ &= 2T(\tfrac{n}{2}) + c \cdot n \end{aligned}$$

where t_* is the time required for a multiplication, and $t_{\pm}$ is the time required for an addition. Since $T(1) = 0$, the recurrence can be solved to

$$\begin{aligned} T(n) &= c \cdot n \log_2 n \\ &= O(n \log n). \end{aligned}$$

The complete fast Fourier transform agorithm is given below [19, p.298].

Algorithm FFT (*Fast Fourier transform (FFT)*)

Input.

- Primitive n-th root of unity ω_n, for some $n = 2^k$.
- Polynomial $a(x) = \sum_{i=0}^{n-1} a_i x^i$, given by its coefficients a_i.

Output. Vector $FT(a) = [f_0, \ldots, f_{n-1}]^T$ where $f_j = \sum_{i=0}^{n-1} a_i \omega_n^{ji}$.

Method. FFT procedure (fig. 2.4).

To compute the inverse fast Fourier transform we modify the procedure slightly by replacing ω_n^k by ω_n^{-k} and dividing f_{a_i} by n.

Most scientific applications use an iterative version of the FFT procedure because of its slightly enhanced speed. Furthermore, by using the original data space to store the temporary results of the combining process the storage requirements are kept within linear bound. That way the original sequence of the input data is lost, and the output data appear in a scrambled or bit-reversed order. Figure 2.5 illustrates the principle of the in-place

Figure 2.4: Recursive FFT procedure

procedure FFT(n, $a(x)$, ω_n, $f_a[]$)

if ($n = 1$) *then do begin* /* *end of recursion* */
$f_a = a_0$
end
else do begin /* *split in even and odd terms* */

$n = \frac{n}{2}$
$b(x) = \sum_{i=0}^{n-1} a_{2i} x^i$
$c(x) = \sum_{i=0}^{n-1} a_{2i+1} x^i$

/* *recursive calls* */

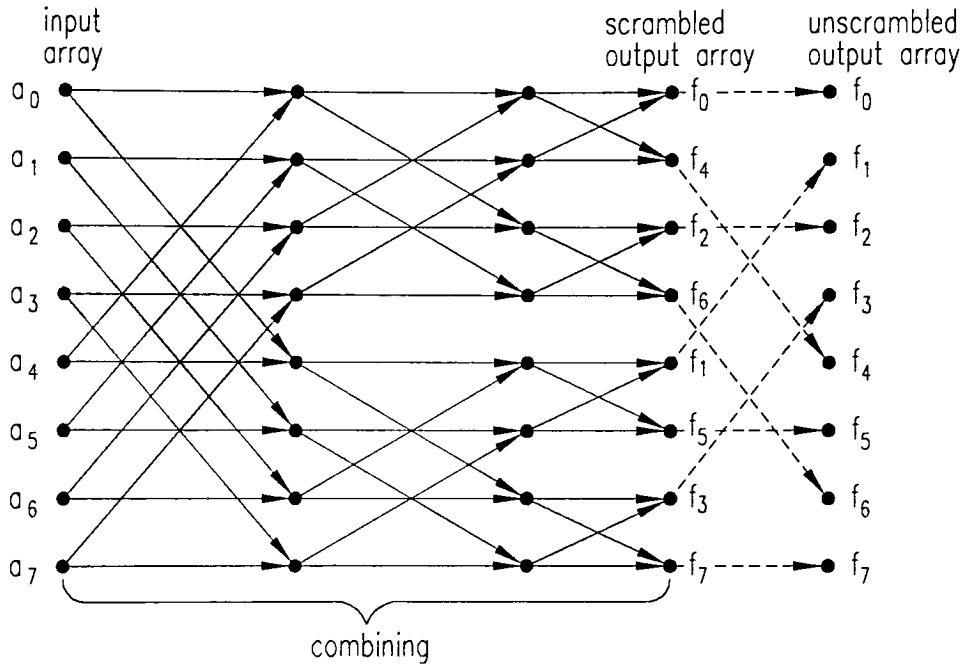
FFT(n, $b(x)$, ω_n^2, $f_b[]$)
FFT(n, $c(x)$, ω_n^2, $f_c[]$)

/* *combine* */

for ($k = 0$) *until* ($n - 1$) *do begin*
$f_{a_k} = f_{b_k} + \omega_n^k \times f_{c_k}$
$f_{a_{k+n}} = f_{b_k} - \omega_n^k \times f_{c_k}$
end
end

computation and the subsequently required unscrambling of the output data for $n = 8$. At each level of computation the array of n data is overwritten by the new results. Note, it is not necessary that the unscrambling is performed at the end of the FFT procedure. Analogously it can be applied to the input data [10]. Since in many applications the forward transform is followed by the inverse transform, the appropriate choice of FFT algorithm may effectively avoid the reordering of the data [34] [9].

Figure 2.5: In-place computation and unscrambling of the output data.



2.5.5 Mod m Fast Fourier Transform

In section 2.5.2 the discrete Fourier transform was defined in an arbitrary field F. If we use the field of the complex numbers some disadvantages have to be taken into consideration: Complex numbers must be approximated by finite precision numbers, thus introducing round-off errors. Both multiplication operation in general and the calculation of powers ω_n^i are time-consuming, so often the ω_n^i are calculated once and then stored in a table to save time. We may profit from working in a finite field or ring Z_m

of integers with artihmetic carried out modulo an integer m instead, and performing an FFT-like transform, called *mod m fast Fourier transform* or *number theoretic transform (NTT)* [23]. Employing a number theoretic transform makes it possible to get rid of round-off errors, to replace multiplications by shifts and additions, and to avoid the storage of the complex values ω_n^i. Naturally this requires the choice of appropiate values for the parameters n, modulus m and n-th root of unity ω_n.

Note, that Z_m is a ring in general, but it is also a field for m prime. The Fourier transform and its inverse were shown to be valid both in the field and in the ring Z_m [25]. The conditions concerning the primitive n-th root of unity ω_n (definition 2.1) in Z_m become equivalent to the congruences

$$\omega_n^n \equiv 1 (\text{mod } p_i^{r_i}), \quad \text{for all } i,\ 1 \leq i \leq l \tag{2.4}$$

$$\omega_n^k \not\equiv 1 (\text{mod } p_i), \quad \text{for all } k,\ 1 \leq k < n, \text{ and some } i, 1 \leq i \leq l \tag{2.5}$$

with

$$m = p_1^{r_1} \cdot p_2^{r_2} \cdot \ldots \cdot p_l^{r_l} \qquad [16]. \tag{2.6}$$

Next, the necessary and sufficient condition for the existence of a number theoretic transform of length n will be derived. Thereby, EULER's totient function $\varphi(m)$ is the number of integers in Z_m that are relatively prime to m.

Theorem 2.1 (EULER's theorem) *For every α and m such that the greatest common divisor of α and m is 1, denoted $gcd(\alpha, m) = 1$,*

$$\alpha^{\varphi(m)} \bmod m = 1 \qquad \textit{[12, p.42].}$$

For m a prime, $\varphi(m) = m-1$. If m is composite and has prime factorization $m = p_1^{r_1} \cdot p_2^{r_2} \cdot \ldots \cdot p_l^{r_l}$, then $\varphi(m)$ is given by

$$\varphi(m) = \prod_{i=1}^{l} p_i^{r_i - 1}(p_i - 1) \qquad [12, \text{p.41}]. \tag{2.7}$$

EULER's theorem implies that α only can be a primitive n-th root of unity if n divides $\varphi(m)$, denoted $n|\varphi(m)$, since $(\alpha^{\frac{\varphi(m)}{n}})^n \bmod m = \alpha^{\varphi(m)} \bmod m$. By applying the same line of reasoning to equation 2.4, ω_n is an n-th root of unity only if $n|\varphi(p_i^{r_i})$, with $\varphi(p_i^{r_i}) = p_i^{r_i - 1} \cdot (p_i - 1)$ (equation 2.7), i.e.

$$n | p_i^{r_i - 1}(p_i - 1).$$

Furthermore, since not only the field Z_m but also the ring Z_m is taken into condideration it must be ensured for the inverse Fourier transform that n has an inverse element in Z_m. Hence n must be relatively prime to the p_i's, and we obtain

$$n|(p_i - 1), \qquad \text{for all } i.$$

Finally, we define the necessary and sufficient condition for the existence of number theoretic transforms of length n. Theorem 2.2 states exactly what the possible transform lengths for a given modulus are.

Theorem 2.2 (Existence of n-th Degree NTT) *Let Z_m be a ring of integers modulo m, $m = p_1^{r_1} \cdot p_2^{r_2} \cdot \ldots \cdot p_l^{r_l}$. A number theoretic transform (NTT) of length n exists in Z_m if and only if*

$$n|gcd((p_1 - 1), (p_2 - 1), \ldots, (p_l - 1)) \qquad [16][1].$$

We are interested especially in NTT's which have fast implementations. There are three requirements for fast NTT's which have to be considered:

1. $n = 2^k$. This is an obvious consequence of the algorithm for the FFT.
2. Multiplication by powers of ω_n should be simple and fast, preferably shifts, which can be achieved if ω_n is a power of 2.
3. Arithmetic mod m should be fast.

The following corollary to the theorem 2.2 specifies the conditions for the existence of length n number theoretic transforms, for $n = 2^k$, thus meeting requirement 1.

Corollary 2.1 (Existence of $n = 2^k$ NTT's) *Let Z_m be a ring of integers modulo m, $m = p_1^{r_1} \cdot p_2^{r_2} \cdot \ldots \cdot p_l^{r_l}$. The length $n = 2^k$ NTT exists in Z_m if and only if it is possible to write all primes p_i $(i = 1, 2, \ldots, l)$ in the form*

$$p_i = g_i 2^{h_i} + 1,$$

where g_i is an odd number and $h_i \geq n$.

Proof. It follows from theorem 2.2 that primes p_i $(i = 1, 2, \ldots, l)$ must be greater than two for the length $n > 1$ transform to exist. Every prime number that is greater than two can always be uniquely written as

$$p_i = g_i 2^{h_i} + 1, \qquad i = 1, 2, \ldots, l$$

where $h_i \geq 1$ and g_i an odd number. The greatest common divisor of numbers $p_i - 1$ $(i = 1, 2, \cdots, l)$ can be written as

$$(p_1 - 1, p_2 - 1, \ldots, p_l - 1) = c \cdot 2^d$$

where c is the greatest common divisor of numbers g_i, and d is the smallest of the exponents h_i. It then follows from theorem 2.2 that length $n = 2^k$ NTT's exist exactly when 2^k divides 2^d, which is true if all exponents h_i conform to $h_i \geq n$ [16].□

Let us call primes of the form $p_i = g_i 2^{h_i} + 1$ *Fourier primes*. Examples of Fourier primes are some of the Fermat numbers F_t, where

$$F_t = 2^{2^t} + 1,$$

which are used by a number theoretic transform, called Fermat number transform. SCHÖNGAGE and STRASSEN adopted in their multiplication algorithm, version 1971, the Fermat number transform, whereas the version from 1982 works with moduli which are composed of Fourier primes [32] [28].

Originally Fermat conjectured that all Fermat numbers are primes, but it seems that only F_0 through F_4 are prime and all others are composite. The first seven values are:

$$\begin{aligned}
F_0 &= 3 \\
F_1 &= 5 \\
F_2 &= 17 \\
F_3 &= 257 \\
F_4 &= 65537 \\
F_5 &= 4\ 294\ 967\ 297 = 641 \times 6\ 700\ 417 \\
F_6 &= 274\ 177 \times 67\ 280\ 421\ 310\ 721 \simeq 1.84 \times 10^{19} [1].
\end{aligned}$$

Each prime factor of a composite Fermat number F_t is of the form $k2^{t+2} + 1$ [1]. To simplify matters we will concentrate on F_0 through F_4 which are prime and have a maximum length n transform of $max(n) = \varphi(F_t) = F_t - 1$. $max(n)$ can be achieved by choosing ω_n, a primitive n-th root of unity, to be 3, but requirement 2 demands that ω_n is a power of two. Now, for $\omega_n = 2$ we get $n = 2^{t+1}$, since $\omega_n^n \bmod F_t = (2)^{2^{t+1}} \bmod F_t = (2^{2^t})^2 \bmod F_t = (-1)^2 \bmod F_t = 1 \bmod F_t$. The length n for $\omega_n = 2^k$ can be found in an

analogous way. In the case of composite numbers F_t we also get $n = 2^{t+1}$ for $\omega_n = 2$ [1].

In requirement 3 we asked for simple arithmetic modular operations. The conditions to attain fast operations are not very restrictive and not only Fermat numbers fulfill them. For an arbitrary number $m = x^l + 1$, if we write a number a in radix x^l notation as a sequence of b blocks of l digits, then a mod m can be calculated by alternately adding and subtracting the b blocks of l digits.

Lemma 2.1 (Fast computing of modulo) *Let $m = x^l + 1$ and let $a = \sum_{i=0}^{b-1} a_i x^{li}$, where $0 \leq a_i < x^l$ for each i. Then*

$$a \equiv \sum_{i=0}^{b-1} a_i (-1)^i \bmod m.$$

Proof. Observe that $x^l = -1 \bmod m$ [2].□

Example 2.2
Let $x = 2, l = 2$ and $m = 2^2 + 1$. Consider $a = (101100)_2 = (230)_4 = 44$. Here $a_0 = 0$, $a_1 = 3$, and $a_2 = 2$. We compute $a_0 - a_1 + a_2 = -1$ and then add m to find that $a \equiv 4 \bmod 5$.

2.5.6 Modular Arithmetic

Modular arithmetic is used in different fast convolution algorithms. The underlying idea is to have several "moduli" $m_1, m_2, \ldots, m_r$ which are pairwise relatively prime, and instead of doing calculations directly with an integer u ($0 \leq u < m_1 \cdot m_2 \cdot \ldots \cdot m_r$) we work with "residues" $u \bmod m_i$. Let us denote

$$u_1 = u \bmod m_1, \qquad u_2 = u \bmod m_2 \ldots$$

It is trivial to compute $u_1, u_2, \ldots, u_r$ and it is important to note that there is no information lost at all by performing this computation. The *Chinese remainder theorem* states, that we can recompute u, given its residues $u_1, \ldots, u_r$.

Theorem 2.1 (Chinese remainder theorem) *Let $m_1, m_2, \ldots, m_r$ be positive integers that are relatively prime in pairs, i.e.,*

$$gcd(m_j, m_k) = 1 \qquad \text{if} \quad j \neq k.$$

Let $m = m_1 \cdot m_2 \cdot \ldots \cdot m_r$ and let $u_1, \ldots, u_r$ be integers. Then there is exactly one integer u that satisfies the conditions

$$0 \leq u < m, \text{ and } \quad u \equiv u_j \pmod{m_j} \quad \text{for} \quad 1 \leq j \leq r [15, p.270].$$

Proof. For each j, $1 \leq j \leq r$, $gcd(m_j, \frac{m}{m_j}) = 1$. Therefore there exists an inverse element y_j such that $(\frac{m}{m_j}) \cdot y_j \bmod m_j = 1$. Since m_j and $\frac{m}{m_j}$ are relatively prime, by EULER's theorem (theorem 2.1) we may take

$$y_j = (\frac{m}{m_j})^{\varphi(m_j)-1}.$$

Furthermore, $(\frac{m}{m_j})y_j \bmod m_i = 0$ for $i \neq j$ because m_i is a factor of $(\frac{m}{m_j})$. Let

$$u = \left(\sum_{j=1}^{r} (\frac{m}{m_j}) y_j u_j \right) \bmod m. \tag{2.8}$$

Then u is a solution of $u \bmod m_j = u_j$ $(1 \leq j \leq r)$ because

$$u \bmod m_j = \left(\frac{m}{m_j} \right) y_j u_j \bmod m_j = u_j$$

[15, p.270] [19, p.254][12, p.47] . □

Unfortunately, calculating u corresponding to formulae 2.8 requires both the evaluation of EULER's φ-function and the handling of quite large numbers, since $\frac{m}{m_j} y_j u_j$ could be very large. In KNUTH [15, p.274] and LIPSON [19, p.254] the description of a superior algorithm can be found. We include such an algorithm for the special case of $r = 2$, since that way it is employed by the algorithm of SCHÖNHAGE and STRASSEN [32].

Let $u_1 = u \bmod m_1$, $u_2 = u \bmod m_2$ and $\gcd(m_1, m_2) = 1$. We are looking for the solutions to

$$u = u_1 + \alpha \cdot m_1 \tag{2.9}$$

$$u = u_2 + \beta \cdot m_2 \tag{2.10}$$

Note that $u_1 + \alpha m_1 \equiv u_2 \bmod m_2$ if $\alpha m_1 \equiv u_2 - u_1 \bmod m_2$. Since (m_1, m_2) are relatively prime we have a inverse element m_1^{-1} with $m_1 m_1^{-1} \equiv 1 \bmod$

m_2. Thus, if we choose $\alpha = (u_2 - u_1) \cdot m_1^{-1} \bmod m_2$ then

$$\begin{aligned} u &= u_1 + \alpha m_1 \\ &\equiv u_1 + (u_2 - u_1) m_1^{-1} m_1 \bmod m_2 \\ &\equiv u_1 + (u_2 - u_1) 1 \bmod m_2 \\ &\equiv u_2 \bmod m_2, \end{aligned}$$

so that

$$u = u_1 + \alpha m_1 = u_1 + [((u_2 - u_1) \cdot m_1^{-1}) \bmod m_2] \cdot m_1 \qquad (2.11)$$

satisfies equations 2.9 and 2.10 [19, p.254].

Example 2.3
$m_1 = 7, m_2 = 9, u_1 = 6, u_2 = 3$, then

$$\begin{aligned} u &\equiv 6 \pmod 7 \\ u &\equiv 3 \pmod 9 \end{aligned}$$

and $m_1^{-1} \bmod 9 = 7^{-1} \bmod 9 = 4$, since $4 \cdot 7 \equiv 1 \bmod 9$. Finally $u = 6 + [(3 - 6) \cdot 4 \bmod 9] \cdot 7 = 48$ [19, p.255].

Equation 2.11 is used by the SCHÖNHAGE-STRASSEN algorithm [32]. The Chinese remainder theorem is also used in a fast convolution algorithm, called "three prime algorithm" (see also section 2.5.7). Here three small primes p_i, where the p_i's are smaller than the word size of the computer, are chosen to perform the fast Fourier transform and its inverse in Z_{p_i}. Single- or double precision operations can be used, resulting in fast Fourier transform procedures, but the resulting coefficients have to be reconstructed by the Chinese remainder theorem.

2.5.7 Ordinary Fast Convolution Algorithm

Simple fast convolution algorithms work analogously to the principles developed in section 2.5.3. There are two additional problems involved, which we will discuss now.

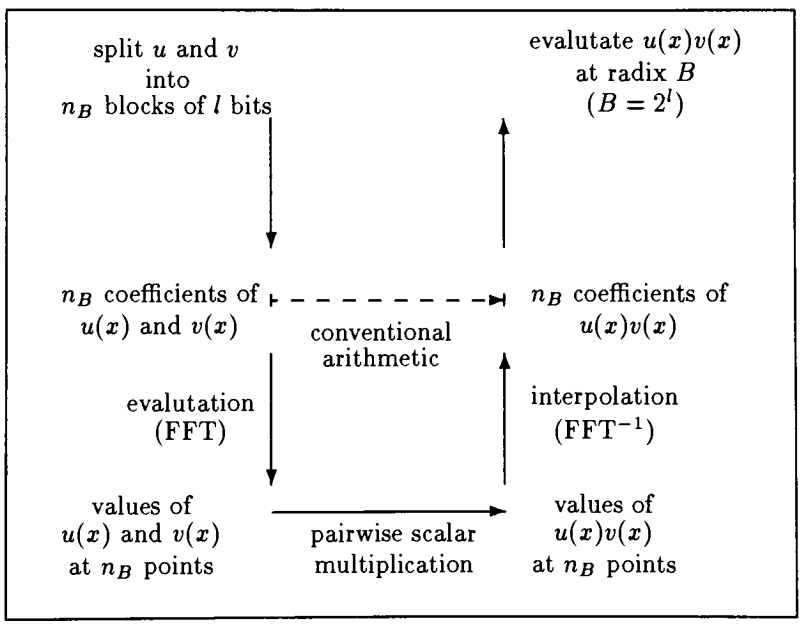
The first problem is how to find "admissible" triples (ω_{n_B}, n_B, m) [5] , such that n_B has a multiplicative inverse and ω_{n_B} is a primitive n_B-th root of

[5] Recall the terminology introduced in section 2.2. We have to distinguish between n_2 and n_B, since here an n_2-bit number will be split into n_B blocks of l bits.

unity in the ring of integers modulo m, so we can apply the fast Fourier transform and the convolution theorem [2, p.274]. We want to set aside this question for a moment.

The second problem is how to split two large n_2-bit numbers u and v into coefficients, needed in the polynomial-based representation and multiplication algorithms. Imagine a large n_2-bit integer u, where corresponding to the convolution theorem (theorem 2.1) the $\frac{n_2}{2}$ high order bits are zeros. We may think of it, for example, as a huge string of bits which has to be stored and handled in some proper form. Now, an obvious solution is to split the number into n_B blocks of bits, each block fitting the size of a byte or a computer word. Furthermore we would like to define the blocks to be our coefficients of the polynomial $u(B)$. Fig. 2.6 shows an extension of fig. 2.2 including the process of splitting two n_2-bit numbers u and v into n_B blocks of l bits, where $n_2 = n_B \cdot l$, and reconstructing the resulting long integer product from the single coefficients. Note that the dashed line indicates a logical relation which is not a part of the calculation procedure. We per-

Figure 2.6: Scheme of ordinary convolution.



form the Fourier transform in Z_m, and the bit-size of the blocks has to be at least as large as $\lceil \log_2(m+1) \rceil$. Furthermore, m should be large enough to preserve the elements c_i of the convolution. Since

$$u = \sum_{i=0}^{n_B-1} u_i 2^{li}, \qquad v = \sum_{i=0}^{n_B-1} v_i 2^{li}, \tag{2.12}$$

where $0 \leq u_i, v_i < 2^l$ for $0 \leq i < \frac{n_B}{2}$ and $u_i = v_i = 0$ for $i \geq \frac{n_B}{2}$, the elements c_i of the convolution are

$$c_i = \sum_{j=0}^{n_B-1} u_j v_{i-j}, \qquad 0 \leq i < n_B \tag{2.13}$$

($u_i = v_i = 0$ if $i < 0$ or $i \geq \frac{n_B}{2}$). From this expression it is obvious that for $0 \leq u_i, v_i < 2^l$

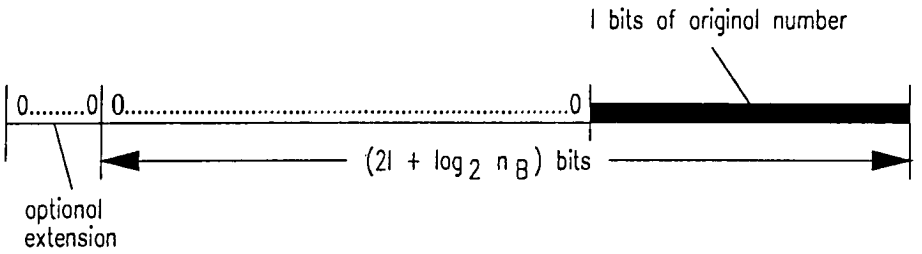
$$0 \leq c_i < \frac{n_B}{2} \cdot 2^{2l}. \tag{2.14}$$

Thus

$$m \geq \frac{n_B}{2} \cdot 2^{2l} > max(c_i) \tag{2.15}$$

is the resulting condition for m. Fig. 2.7 illustrates the principal structure of a coefficient block. It would be attractive to have coefficient blocks which

Figure 2.7: Scheme of coefficient block



are smaller or equal to the word size of a computer (or eventually half the word size), since then only single or double precision operations are needed to perform the fast Fourier transform and its inverse. But then we limit m to $m \leq$ *word size* and unfortunately, as a result, the permitted size for large numbers u and v is also limited.

Example 2.4
Take $m = 127 \cdot 2^{24} + 1$, where m is the largest Fourier prime less than 2^{31} [19, p.306]. Then $max(n_B) = 2^{24}$ (section 2.5.5) and since n_B is a power of two we can apply a fast Fourier transform algorithm. Using $m \geq \frac{n_B}{2} 2^{2l}$ and $n_2 = \frac{n_B}{2} l$ [6] the following examples reflect the restriction on the size of large numbers.

n_2	l	n_B	$m \geq$
208	13	2^4	2^{30}
2 816	11	2^8	2^{30}
36 864	9	2^{12}	2^{30}
458 752	7	2^{16}	2^{30}
$\approx 5 \cdot 10^6$	5	2^{20}	2^{30}

(Note, the restriction on the size of the numbers is more annoying for $m < 2^{16}$).

For larger sized numbers we either have to relax the restriction on the size of m, allowing $m >$ *word size* or we have to choose a different approach to the problem. In [25] [19, p.310] [4] the Chinese remainder theorem is used by the so-called "three prime algorithm" to achieve a fast convolution algorithm which works for still larger numbers although the m_i's are still less or equal to the word size (see also section 2.5.6).

The first problem, finding admissible triples (ω_{n_B}, n_B, m) is not trivial, there is no systematic way of determining "best" choices [1]. One must use intuition, insight and some searching. Often m is selected and the resulting possible n_B and ω_{n_B} are then examined. Examples for unrestricted sizes of m are given in [20] [28].

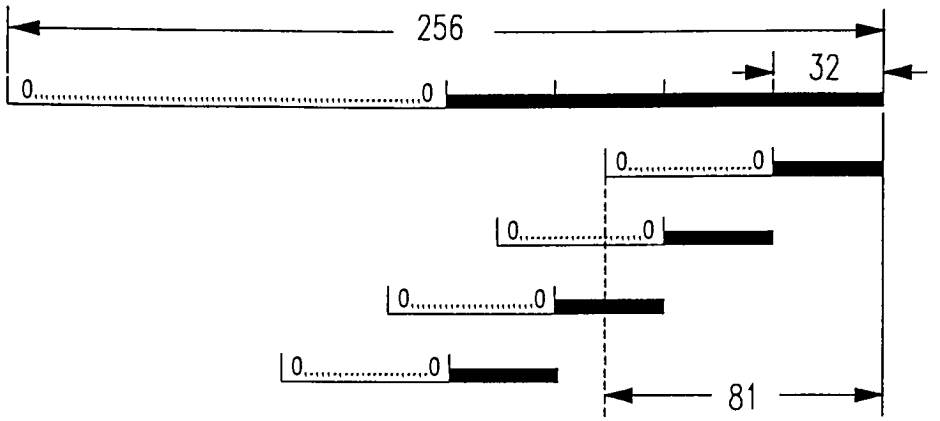
The final illustration in Fig. 2.8 uses a triple $(\omega_{n_B}, n_B, m) = (2^{20}, 8, 2^{80}+1)$ [28]. A large number u with 256 bits, the 128 leading bits filled with zeros, is subdivided into 8 blocks of 32 bits each ($\frac{256}{8} = 32$). Each block of 32 bits is embedded into a 81-bit string, as a number in $Z_{2^{80}+1}$. Then the fast convolution algorithm can be performed as indicated in Fig. 2.6.

2.5.8 SCHÖNHAGE-STRASSEN's Algorithm (1971)

Now we turn to a more sophisticated fast convolution algorithm, namely the SCHÖNHAGE-STRASSEN multiplication algorithm in its original version

[8] Here n_2 is the number of not 0-padded bits of the factors u and v.

Figure 2.8: Division of a large number into blocks of bits (ordinary fast convolution algorithm).



from 1971 [32] [2]. This algorithm achieves $O(n \log n \log\log n)$ bit complexity and is the asymptotically fastest known multiplication algorithm.

Similarly to the ordinary convolution algorithm in section 2.5.7, we split two n_2-bit numbers u and v into n_B blocks of l bits. To simplify matters, we restrict n_2 to be a power of two,

$$n_2 = 2^k. \tag{2.16}$$

For $n_2 \neq 2^k$ we add an appropriate number of leading zeros, thus increasing the constant factor of the bit complexity. Furthermore, we calculate the product of two n_2-bit integers u and v mod $(2^{n_2}+1)$. To obtain the exact product uv, we must add leading 0's and multiply the increased $2n_2$-bit integers u and v mod $(2^{2n_2}+1)$, again increasing the complexity by a constant factor. In the following let u and v be n_2-bit integers which we multiply mod $(2^{n_2}+1)$. Observe,that 2^{n_2} requires n_2+1 bits for its binary representation. For $u = 2^{n_2}$ or $v = 2^{n_2}$ the multiplication reduces to a special simple case:

1) $u = 2^{n_2}$, $v \neq 2^{n_2}$, $uv \equiv (2^{n_2}+1-v) \pmod{2^{n_2}+1}$ or

2) $u \neq 2^{n_2}$, $v = 2^{n_2}$, $uv \equiv (2^{n_2}+1-u) \pmod{2^{n_2}+1}$ or

3) $u = 2^{n_2}$, $v = 2^{n_2}$, $uv \equiv 1 \pmod{2^{n_2}+1}$.

Otherwise the n_2-bit numbers u and v are split into n_B blocks

$$\begin{aligned} n_B &= 2^{\frac{k}{2}}, && \text{if } k \text{ is even,} \\ n_B &= 2^{\frac{k-1}{2}}, && \text{if } k \text{ is odd,} \end{aligned}$$

of l bits,

$$\begin{aligned} l &= 2^{\frac{k}{2}}, && \text{if } k \text{ is even,} \\ l &= 2^{\frac{k+1}{2}}, && \text{if } k \text{ is odd,} \end{aligned}$$

with $n_2 = n_B \cdot l$. We write u and v in radix $B = 2^l$

$$\begin{aligned} u &= u_{n_B-1}2^{(n_B-1)l} + \ldots + u_1 2^l + u_0 \\ v &= v_{n_B-1}2^{(n_B-1)l} + \ldots + v_1 2^l + v_0. \end{aligned}$$

Further parameters of the admissible triple (ω_{n_B}, n_B, m) (section 2.5.7) are

$$\begin{aligned} \omega_{n_B} &= 2^{\frac{4l}{n_B}} \text{ and} \\ m &= 2^{2l} + 1. \end{aligned}$$

Table 2.1 summarizes the parameters. Theorem 2.2 states, that a number theoretic transform of length n_B exits in Z_m, for $m = p_1^{r_1} \cdot p_2^{r_2} \cdot \ldots \cdot p_l^{r_l}$, if and only if

$$n_B | \gcd((p_1 - 1), (p_2 - 1), \ldots, (p_l - 1)). \tag{2.17}$$

It can easily be shown that the parameters in table 2.1 meet this condition. We will do it for k even, k odd is analogous. First, observe that the m_i's are Fermat numbers F_t, with

$$F_t = 2^{2^t} + 1.$$

Table 2.1: Parameters for SCHÖNHAGE-STRASSEN algorithm, version 1971

	l	n_B	ω_{n_B}	m
k: even	$2^{\frac{k}{2}}$	$2^{\frac{k}{2}}$	2^4	$2^{(2^{\frac{k+2}{2}})} + 1$
k: odd	$2^{\frac{k+1}{2}}$	$2^{\frac{k-1}{2}}$	2^8	$2^{(2^{\frac{k+3}{2}})} + 1$

We must distinguish between $t \leq 4$, where F_t is prime, and $t > 4$ where F_t is composed of primes of the form $p = k2^{t+2} + 1$ (section 2.5.5). For $t \leq 4$ and F_t prime equation 2.17 reduces to the condition $n_B|(F_t - 1)$ which is true for $n_B = 2^{\frac{k}{2}}$ and $F_t = m = 2^{2^{\frac{k+2}{2}}} + 1$, since $\frac{k}{2} \leq 2^{\frac{k+2}{2}}$. For $t > 4$ we have $n_B|(2^{t+2})$ or $(2^{\frac{k}{2}})|(2^{\frac{k+2}{2}} + 2)$ since $\frac{k}{2} < \frac{k}{2} + 3$.

So far, given the parameters of table 2.1, the convolution theorem can be applied. Furthermore, the three requirements for a fast number theoretic transform (section 2.5.5) are realized: Since n_B is a power of two a fast FFT like algorithm can be used. The multiplications by powers of ω_{n_B} are replaced by simple shifts since ω_{n_B} is a power of two, and we can build a fast arithmetic package for $m = 2^{2l} + 1$ by doing the calculations for blocks of radix 2^{2l}.

The product uv is equal to the convolution of the coefficients of u and v and is given by

$$uv = y_{2n_B-2}2^{(2n_B-2)l} + \ldots + y_1 2^l + y_0 \tag{2.18}$$

where

$$y_i = \sum_{j=0}^{n_B-1} u_j v_{i-j}, \qquad 0 \leq i < 2n_B. \tag{2.19}$$

(For $j < 0$ or $j > n_B - 1$, $u_j = v_j = 0$. $y_{2n_B-1} = 0$ is included for symmetry only.) Applying the convolution theorem to compute the product uv (fig. 2.2), the pairwise multiplication of the Fourier transforms of u and v would require $2n_B$ multiplications. We can reduce the number of multiplications to n_B by using the negative wrapped convolution (definition 2.2). Rewriting equation 2.18 gives

$$\begin{aligned} uv &= y_{2n_B-2}2^{(2n_B-2)l} + \ldots y_{n_B}2^{(n_B)l} + y_{n_B-1}2^{(n_B-1)l} + y_{n_B-2}2^{(n_B-2)l} + \ldots + y_0 \\ &= (y_{2n_B-2}2^{(n_B-2)l} + \ldots y_{n_B}) \cdot 2^{n_B l} + (y_{n_B-1}2^{(n_B-1)l} + y_{n_B-2}2^{(n_B-2)l} + \ldots + y_0). \end{aligned}$$

Since $n_B \cdot l = n_2$ we have $2^{n_B l} \equiv -1 \bmod (2^{n_2} + 1)$, and by lemma 2.1 (fast computing of modulo) we get

$$\begin{aligned} uv &\equiv -(y_{2n_B-2}2^{(n_B-2)l} + \ldots y_{n_B}) + (y_{n_B-1}2^{(n_B-1)l} + \ldots + y_0) \pmod{2^{n_2}+1}. \\ &\equiv (w_{n_B-1}2^{(n_B-1)l} + \ldots + w_1 2^l + w_0) \pmod{2^{n_2}+1}, \\ &\text{with} \quad w_i = y_i - y_{n_B+i}, \qquad 0 \leq i < n_B, \end{aligned}$$

where the w_i's are the coordinates of the negative wrapped convolution.

There is one nuisance more which adds some clumsiness to this multiplication algorithm. Recall that m has to be at least as large as the elements of the convolution, w_i,

$$m \geq max(w_i) \qquad \text{(section 2.5.7)}.$$

The product of two l-bit numbers is less than 2^{2l} and y_i and y_{n_B+1} are sums of $i+1$ and $n_B-(i+1)$ such products, respectively. Therefore $w_i = y_i - y_{n_B+1}$ is in the range of

$$-(n_B - 1 - i)2^{2l} < w_i < (i+1)2^{2l}, \tag{2.20}$$

and w_i can have at most $n_B \cdot 2^{2l}$ different values. On the other hand, $m < n_B \cdot 2^{2l}$, since $m = 2^{2l} + 1$, and we can't recover the precise integer product of u and v just by applying the convolution theorem once. If we use the Chinese remainder theorem (theorem 2.1) and compute the w_i's twice, once modulo $2^{2l} + 1$ and once modulo n_B, we can recover all possible $n_B 2^{2l}$ values of w_i. Take

$$\begin{aligned} w_i' &= w_i \bmod n_B \\ w_i'' &= w_i \bmod (2^{2l} + 1). \end{aligned}$$

Because n_B is a power of two and $(2^{2l} + 1)$ is odd, n_B and $(2^{2l} + 1)$ are relatively prime, and we can make use of the Chinese remainder theorem, namely by equation 2.11 we have

$$w_i = w_i'' + [((w_i' - w_i'') \cdot (2^{2l} + 1)^{-1}) \bmod n_B] \cdot (2^{2l} + 1). \tag{2.21}$$

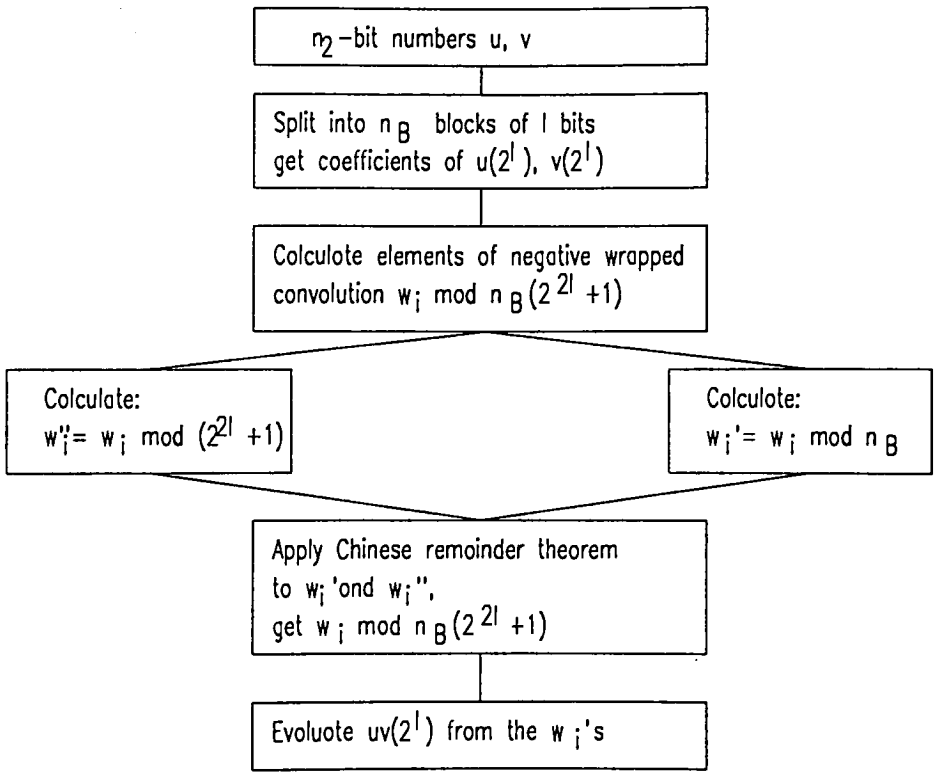
Since n_B is a power of 2 and $n_B < 2^{2l}$, $n_B | (2^{2l})$ and $(2^{2l}+1)^{-1} \bmod n_B = 1$. Thus

$$w_i = w_i'' + [(w_i' - w_i'') \bmod n_B] \cdot (2^{2l} + 1). \tag{2.22}$$

Fig. 2.9 illustrates the basic structure of the algorithm. For simplicity, the recursive character of the algorithm is not illustrated here.

The product uv can be computed from the w_i's by adding the n_B w_i's together with appropriate shifts. To calculate the w_i's mod $n_B(2^{2l} + 1)$ we compute the w_i's twice, once mod n_B and once mod $2^{2l} + 1$. Then we compute the w_i mod $n_B(2^{2l} + 1)$ by employing the Chinese remainder theorem for the partial components w_i' and w_i''.

Figure 2.9: Structure of SCHÖNHAGE-STRASSEN algorithm



We already developed the method for calculating w_i'', namely the negative wrapped convolution by means of the fast Fourier transform for the parameters of table 2.1. With $\psi = w^{\frac{2l}{n_B}}$, ψ is a $2n_B$-th root of unity, and so the negative wrapped convolution of

$$[u_0, \psi u_1, \ldots, \psi^{n_B-1} u_{n_B-1}] \quad \text{and} \quad [v_0, \psi v_i, \ldots, \psi^{n_B-1} v_{n_B-1}]$$

is

$$[w_0, \psi u_1, \ldots, \psi^{n_B-1} w_{n_B-1}] \bmod (2^{2l}+1).$$

The $w_i'' = w_i \bmod (2^{2l}+1)$ can be obtained by dividing by ψ^i which are simple shift operations since ψ is a power of two.

The method to calculate $w_i \bmod n_B$ is different. Take

$$\begin{aligned} u_i' &= u_i \bmod n_B \\ v_i' &= v_i \bmod n_B. \end{aligned}$$

The convolution elements y_i', where

$$y_i' = \sum_{i=0}^{n_B-1} u_i' v_{i-j}', \qquad 0 \le i < 2n_B - 1, \quad 0 \le u_i, v_i < n_B$$

and

$$y_i' < n_B \cdot 2^{2\log_2 n_B}$$

require at most $3\log_2 n_B$ bits for their representation. We form two longer numbers $\hat{u}$ and $\hat{v}$ as shown in fig. 2.10 with

$$\hat{u} = \sum_{i=0}^{n_B-1} u_i' 2^{3\log_2 n_B}, \quad \hat{v} = \sum_{i=0}^{n_B-1} v_i' 2^{3\log_2 n_B}$$

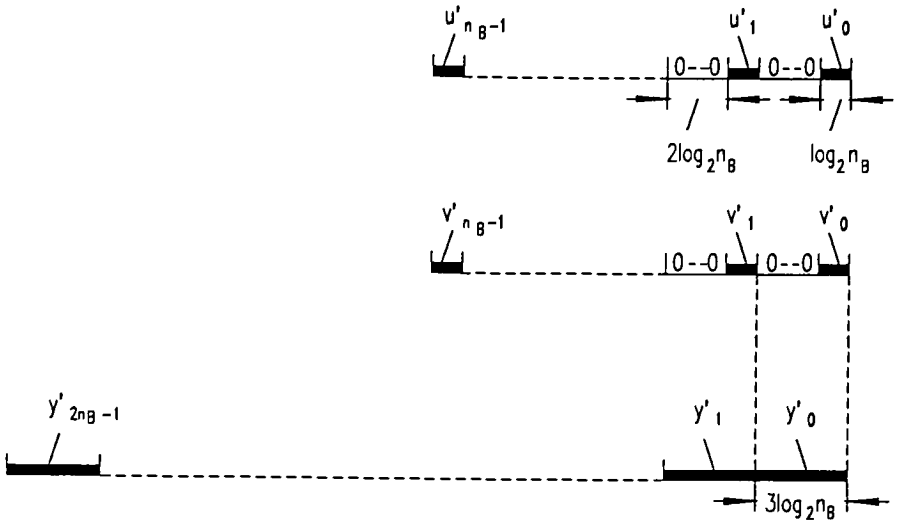
and get the product $\hat{u}\hat{v}$ by employing any multiplication algorithm for large numbers. Note, that n_B is much smaller than 2^{2l}. From the result $\hat{u}\hat{v}$,

$$\hat{u}\hat{v} = \sum_{i=0}^{2n_B-1} y_i' 2^{(3\log_2 n_B)i},$$

we can easily pick out the y_i' and calculate $w_i' = (y_i' - y_{n_B+i}') \bmod n_B$.

According to [2, p.272] and fig. 2.9 we summarize the algorithm as follows:

Figure 2.10: Illustration of the numbers $\hat{u}$, $\hat{v}$ and their product $\hat{u}\hat{v}$, used in computation of $w_i \bmod n_B$



Algorithm S1 (*SCHÖNHAGE-STRASSEN integer multiplication algorithm (1971)*)

Input. n_2-bit integers u and v, where $n_2 = 2^k$.

Output. $uv \bmod (2^{n_2} + 1)$.

Method. If n_2 is small, multiply u and v modulo $(2^{n_2} + 1)$ using an appropriate fast algorithm. For large n_2 use this algorithm with the parameters n_B, l, ω_{n_B} and m of table 2.1. Split u and v into n_B blocks of l bits and express

$$u = \sum_{i=0}^{n_B-1} u_i 2^{li}, \quad v = \sum_{i=0}^{n_B-1} v_i 2^{li} \quad \text{for} \quad 0 \le u_i, v_i < 2^l.$$

1. Compute the Fourier transform modulo $m = 2^{2l} + 1$ of

$$[u_0, \psi u_1, \ldots, \psi^{n_B-1} u_{n_B-1}]^T, \quad [v_0, \psi v_1, \ldots, \psi^{n_B-1} v_{n_B-1}]^T$$

with $\psi^2 = \omega_{n_B}$.

2. Compute the coordinatewise product of the Fourier transforms of step 1, mod m, by using algorithm S1 recursively. (The situation in which a factor is 2^{2l} is handled as a special easy case, see page 28.)
3. Compute the inverse Fourier transform of the vector of step 2, mod m, which is $[w_0, \psi w_1, \ldots, \psi^{n_B-1} w_{n_B-1}]^T \bmod (2^{2l}+1)$. Compute $w_i'' = w_i \bmod (2^{2l}+1)$ by multiplying $\psi^i w_i$ by $\psi^{-i} \bmod (2^{2l}+1)$.
4. Compute $w_i' = w_i \bmod n_B$ as follows:
   - Compute $u_i' = u_i \bmod n_B, \quad v_i' = v_i \bmod n_B$, for $0 \le i < n_B$.
   - Construct $\hat{u} = \sum_{i=0}^{n_B-1} u_i' 2^{(3\log_2 n_B)i}$, $\hat{v} = \sum_{i=0}^{n_B-1} v_i' 2^{(3\log_2 n_B)i}$ by stringing the u_i' and v_i' together with $2\log_2 n_B$ intervening 0's (fig. 2.10).
   - Compute $\hat{u}\hat{v} = \sum_{i=0}^{2n_B-1} y_i' 2^{(3log_2 n_B)i}$ using an appropriate fast multiplication algorithm.
   - Compute $w_i' = (y_i' - y_{n_B+i}') \bmod n_B$, for $0 \le i < n_B$.
5. Compute $w_i = w_i'' + [(w_i' - w_i'') \bmod n_B] \cdot (2^{2l}+1)$.
6. Compute $\sum_{j=0}^{n_B-1} w_i 2^{li} \bmod (2^{n_2}+1)$. This is the desired result.

Steps 1 through 3 compute $w_i'' = w_i \bmod (2^{2l}+1)$, step 4 computes $w_i' = w_i \bmod n_B$, step 5 recovers w_i from w_i' and w_i'' by using the Chinese remainder theorem and the last step evaluates the product mod $(2^{n_2}+1)$ with radix 2^l.

Example 2.5
For simplicity we apply algorithm S1 only once and do not exploit its recursive character. Let

$$u = (1011011)_2 = (91)_{10}, \qquad v = (10001111)_2 = (143)_{10}.$$

To get the exact result uv we must add leading zeros and compute uv mod $(2^{16}+1)$. The parameters are

$$n_2 = 16, \qquad n_B = 4, \qquad l = 4, \qquad \omega_4 = 16, \qquad m = 2^8 + 1 = 257,$$

and we split u and v into 4 blocks of 4 bits each:

$$u = 0000 \quad 0000 \quad 0101 \quad 1011$$

$$v = 0000 \quad 0000 \quad 1000 \quad 1111.$$

In positional notation we have

$$u = 0 \cdot 2^{12} + 0 \cdot 2^8 + 5 \cdot 2^4 + 11 = [11,5,0,0]^T,$$
$$v = 0 \cdot 2^{12} + 0 \cdot 2^8 + 8 \cdot 2^4 + 15 = [15,8,0,0]^T.$$

Step 1 With $\psi = 4$ (ψ is the 8-th root of unity) we have

$$u_\psi = [1 \cdot 11, 4 \cdot 5, 0, 0]^T = [11,20,0,0]^T,$$
$$v_\psi = [1 \cdot 15, 4 \cdot 8, 0, 0]^T = [15,32,0,0]^T.$$

To calculate the Fourier transform modulo 257 we use

$$\omega_4^0 = 1, \quad \omega_4^1 = 16, \quad \omega_4^2 = -1, \quad \omega_4^3 = -16$$

and obtain

$$FT(u_\psi) = [31,74,248,205]^T \bmod 257$$
$$FT(v_\psi) = [47,13,240,17]^T \bmod 257.$$

Step 2

$$FT(u_\psi) \cdot FT(v_\psi) = [31 \cdot 47, 74 \cdot 13, 248 \cdot 240, 205 \cdot 17]^T \bmod 257$$
$$\equiv [172,191,153,144]^T \bmod 257.$$

Step 3 The inverse transform requires multiplication by $\omega_{n_B}^{-1}$ and by n_B^{-1}. But $\omega_{n_B}^{-i} \equiv \omega_{n_B}^{n_B - i} \bmod m$ because $\omega_{n_B}^{i} \omega_{n_B}^{n_B - i} \equiv 1 \bmod m$ and $n_B^{-1} \equiv 2^{4l-p} \bmod m$, for $n_B = 2^p$. Since $\omega_{n_B}^{-i}$ and n_B^{-1} are powers of 2 the multiplication reduces to simple shifts. We use

$$\omega_4^0 = 1, \quad \omega_4^{-1} = \omega_4^3 = -16, \quad \omega_4^{-2} = \omega_4^2 = -1, \quad \omega_4^{-3} = \omega_4^1 = 16,$$

$$\psi^0 = 1, \quad \psi^{-1} = \psi^7 = 2^{14}, \quad \psi^{-2} = \psi^6 = 2^{12}, \quad \psi^{-3} = \psi^5 = 2^{10}$$

and $n_B^{-1} = 2^{14}$ and have $FT^{-1}(FT(u_\psi) \cdot FT(v_\psi)) = [165,138,126,0]^T$

$$w_0'' = 165 \cdot \psi^0 \equiv 165 \bmod 257$$
$$w_1'' = 138 \cdot \psi^{-1} \equiv 163 \bmod 257$$
$$w_2'' = 126 \cdot \psi^{-2} \equiv 40 \bmod 257$$
$$w_3'' = 0$$

Step 4

- $u'_0 = 3,\ u'_1 = 1,\ u'_i = 0 \quad \text{if} \quad i > 1,\ v_0 = 3,\ v_i = 0 \quad \text{if} \quad i > 0$
- $\hat{u} = 0 \ldots\ldots 1\ 00\ 00\ 11, \quad \hat{v} = 0 \ldots\ldots 0\ 00\ 00\ 11$
- $\hat{u}\hat{v} = 0 \ldots\ldots 00\ 00\ 11\ 00\ 10\ 01$
- $y_0 = (00\ 10\ 01)_2 = (9)_{10}, \quad y_1 = (00\ 00\ 11)_2 = (3)_{10}, \quad y_i = 0 \quad \text{if} \quad i > 1,$

$$\begin{aligned} w'_0 &= 9 \bmod 4 = 1 \\ w'_1 &= 3 \bmod 4 = 3 \\ w'_i &= 0, \quad \text{if} \quad i > 1 \end{aligned}$$

Step 5

$$\begin{aligned} w_0 &= 165 + [(1 - 165) \bmod 4] \cdot 257 = 165 + 0 = 165 \\ w_1 &= 163 + [(3 - 163) \bmod 4] \cdot 257 = 163 \\ w_2 &= 40 \\ w_3 &= 0 \end{aligned}$$

Step 6

$$uv = 0 \cdot 2^{12} + 40 \cdot 2^8 + 163 \cdot 2^4 + 165 \cdot 2^0 = 13\ 013$$

Since $91 \cdot 143 = 13\ 013$, the result checks!

2.5.9 Modified SCHÖNHAGE-STRASSEN Algorithm (1982)

A. SCHÖNHAGE published an improved version of the SCHÖNHAGE-STRASSEN multiplication algorithm in 1982 [28]. In his introduction he admitted, that the multiplication method in its original version was "somewhat clumsy" [28]. By choosing different parameters, the application of the Chinese remainder theorem can be avoided and the algorithm reduces to step 1, 2, 3 and 6 of the original algorithm S1. The simplified version also achieves $O(n \log n \log\log n)$ bit complexity and seems to be more feasible especially for numbers of moderate length.

Imagine two n_2-bit numbers u and v. In the original version we restricted n_2 to a power of two, $n_2 = 2^k$. Now the formula for n_2 is

$$n_2 = \nu 2^k, \qquad k - 1 \leq \nu \leq 2k - 1, \qquad k \geq 4 \qquad [28]. \tag{2.23}$$

Note, that this representation, if possible, is unique. To compute the product $uv \bmod (2^{n_2}+1)$ we split u and v into n_B blocks of l bits, $n_2 = n_B \cdot l$, with

$$\begin{aligned} n_B &= 2^{\lfloor \frac{k}{2} \rfloor + 1} \\ l &= \nu 2^{\lceil \frac{k}{2} \rceil - 1}. \end{aligned}$$

Further parameters of the admissible triple (ω_{n_B}, n_B, m) are

$$\begin{aligned} m &= 2^{\lceil \frac{\nu+1}{2} \rceil \cdot 2^{\lceil \frac{k}{2} \rceil + 1}} + 1, \\ \omega_{n_B} &= 2^{\lceil \frac{\nu+1}{2} \rceil \cdot 2}, \quad \text{if } k \text{ is even}, \\ \omega_{n_B} &= 2^{\lceil \frac{\nu+1}{2} \rceil \cdot 4}, \quad \text{if } k \text{ is odd}. \end{aligned}$$

The parameters are summarized in table 2.2.

Observe that m is not restricted to Fermat numbers F_t but rather is composed of Fourier primes. Since $\lfloor \frac{k}{2} \rfloor + 1 \leq \lceil \frac{k}{2} \rceil + 1$ we can employ a fast Fourier transform of length $n_B = 2^{\lfloor \frac{k}{2} \rfloor + 1}$.

Fortunately, we don't have to apply the Chinese remainder theorem. Let w_i be the elements of the convolution, then we will show that $m \geq max(w_i)$ is true, and we get the product $uv \bmod (2^{n_2}+1)$ simply by applying the convolution theorem, calculating the w_i's once and adding the w_i's together after appropriate shifts.

Using $max(w_i) = n_B \cdot 2^{2l} = 2^{\lfloor \frac{k}{2} \rfloor + 1 + \nu 2^{\lceil \frac{k}{2} \rceil}}$ we show that $\log_2(m-1) \geq \log_2(max(w_i))$ by noting that

$$\log_2(m-1) = \lceil \frac{\nu+1}{2} \rceil \cdot 2^{\lceil \frac{k}{2} \rceil + 1} \geq \lfloor \frac{k}{2} \rfloor + 1 + \nu 2^{\lceil \frac{k}{2} \rceil}, \tag{2.24}$$

Table 2.2: Parameters for SCHÖNHAGE-STRASSEN algorithm, version 1982

	l	n_B	ω_{n_B}	m
k: even	$\nu 2^{\lceil \frac{k}{2} \rceil - 1}$	$2^{\lfloor \frac{k}{2} \rfloor + 1}$	$2^{\lceil \frac{\nu+1}{2} \rceil \cdot 2}$	$2^{\lceil \frac{\nu+1}{2} \rceil \cdot 2^{\lceil \frac{k}{2} \rceil + 1}} + 1$
k: odd			$2^{\lceil \frac{\nu+1}{2} \rceil \cdot 4}$	

since

$$2^{\lceil \frac{k}{2} \rceil} \geq \lfloor \frac{k}{2} \rfloor + 1, \qquad \text{if } \nu + 1 \text{ is even,} \tag{2.25}$$

$$2^{\lceil \frac{k}{2} \rceil + 1} \geq \lfloor \frac{k}{2} \rfloor + 1, \qquad \text{if } \nu + 1 \text{ is odd.} \tag{2.26}$$

By dropping step 4 and 5 of algorithm S1 (section 2.5.8) and using the parameters of table 2.2 instead of table 2.1 we obtain the fast multiplication algorithm in its version from 1982:

Algorithm S2 (*Modified SCHÖNHAGE-STRASSEN integer multiplication algorithm (1982)*)

Input. n_2-bit integers u and v, where $n_2 = \nu 2^k$.

Output. $uv \bmod (2^{n_2} + 1)$.

Method. If n_2 is small, multiply u and v modulo $(2^{n_2} + 1)$ using an appropriate fast algorithm. For large n_2 use this algorithm with the parameters n_B, l, ω_{n_B} and m of table 2.2. Split u and v into n_B blocks of l bits and express

$$u = \sum_{i=0}^{n_B-1} u_i 2^{li}, \quad v = \sum_{i=0}^{n_B-1} v_i 2^{li} \quad \text{for} \quad 0 \leq u_i, v_i < 2^l.$$

1. Compute the Fourier transform modulo m of

$$[u_0, \psi u_1, \ldots, \psi^{n_B-1} u_{n_B-1}]^T, \quad [v_0, \psi v_1, \ldots, \psi^{n_B-1} v_{n_B-1}]^T$$

with $\psi^2 = \omega_{n_B}$.

2. Compute the coordinatewise product of the Fourier transforms of step 1, mod m, by using algorithm S2 recursively. (The situation in which a factor is $(m-1)$ is handled as a special easy case.)

3. Compute the inverse Fourier transform of the vector of step 2, mod m, which is $[w_0, \psi w_1, \ldots, \psi^{n_B-1} w_{n_B-1}]^T \bmod m$. Compute $w_i \bmod m$ by multiplying $\psi^i w_i$ by $\psi^{-i} \bmod m$.

4. Compute $\sum_{j=0}^{n_B-1} w_i 2^{li} \bmod (2^{n_2} + 1)$. This is the desired result.

The next example demonstrates the recursive character of algorithm S2.

Example 2.6
Given two n_2-bit numbers u and v, where $n_2 = 13 \cdot 2^{12} = 53\ 248$, we split u and$v$ into $2^7(= 128)$ blocks of $13 \cdot 2^5(= 416)$ bits and perform step 1 of algorithm S2 using the parameters $m = 2^{2 \cdot 2^7} + 1 = 2^{896} + 1, \omega_{128} = 2^{14}$ and $\psi = 2^7$. In step 2 we do not multiply directly the Fourier tranforms pairwise but rather apply the algorithm recursively. Therefore we split each 897-bit number into $2^4(= 16)$ blocks of $7 \cdot 2^3(= 56)$ bits and perform step 1 once more, using the parameters $m = 2^{4 \cdot 2^5} + 1 = 2^{128} + 1, \omega_{16} = 2^{16}$ and $\psi = 2^8$. In step 2 we stop the recursion and compute the pairwise products of the Fourier transforms directly. This way a multiplication mod $(2^{53\ 248} + 1)$ requires 2048 multiplications mod $(2^{128} + 1)$ and the connecting FFT's [28].

Chapter 3

Implementation

3.1 Preliminary Remarks

This chapter focuses on the actual implementation of the multiplication algorithms that have been developed from theoretical concepts in chapter 2. Section 3.2 gives an outline of the elements, which are fundamental to all multiplication algorithms. Section 3.3 describes the individual algorithms, namely the classical, KARATSUBA's, the ordinary convolution, the SCHÖNHAGE-STRASSEN (1971), and the modified SCHÖNHAGE-STRASSEN from 1982. For the theoretical background for the algorithms see sections 2.3, 2.4, 2.5.7, 2.5.8, and 2.5.9, respectively. Section 3.3.3 comprises uniform elements of the fast convolution algorithms, the ordinary one, and the SCHÖNHGAE-STRASSEN from 1971 and 1982.

Although the discussion of implemention issues essentially refers to the underlying ideas and principles, at certain situations detailed programming code will be included. It is assumed, that the reader is used to reading programming code, and that programming code is superior to a purely verbal description with respect to transparency, succinctness and exactness. In general, a simplified C-like code will be used. To increase the readability, internal type definitions may be replaced by well-known C-types, multiple-precision operations may be substituted by arithmetic operators of C, names of variables and functions may be changed, and auxiliary variables may be added, for example.

3.2 Fundamental Elements

3.2.1 Hardware and Software

The programs were first developed on an Intel 8088 16-bit microprocessor and then tested on an Intel 80386 32-bit microprocessor with a clock frequency of 25 MHz. Especially for larger numbers we took advantage of the higher speed of the latter one, but the compiler supported only the 8086/8088 instruction set, which is a subset of the 80386 instruction set.

The code is written in C, and to assure portability the American National Standards Institute (ANSI) C and the KERNIGHAN and RITCHIE definitions [14] are observed with minor exceptions. One exception is the procedure for time measurements (section 3.2.5) which is at least available on UNIX [1] System V systems. Since the Intel 8088 is designed to address only 64 k of memory within one segment serious problems arose for large numbers. The Turbo C [2] compiler (version 2.0) which was used, overcomes this restriction by employing special library functions. Therefore a second version of the program which allows us to handle larger numbers but is not portable (section 3.2.3) was developed. Note, the first standard version is portable and the restriction to 64 k is only due to the DOS memory management.

BigNum [3], a C package for large integers, supplied basic procedures, in particular the procedures for addition, subtraction and for the classical multiplication. Further development of the package made it possible to port it to the 8088 Intel processor.

The self-written multiple-precision package comprises about 3500 lines of C programming code and comments. In addition, about 4500 lines of test programs were developed. In particular, the complexity and size of the recursive SCHÖNHAGE-STRASSEN algorithms made their implementation to the most demanding task of the work.

The analysis of different algorithms naturally focuses on a relative comparison, and any arbitrary time unit could be used. The relative comparison does not require speed in terms of absolute time units, and therefore the usage of assembly code, for example, would provide no principal new insights

[1] UNIX is a trademark of Bell Laboratories.

[2] Turbo C is a trademark of Borland.

[3] BigNum is a trademark of Digital Equipment Corporation and INRIA.

into the performance of the algorithms. By dispensing with assembly code the portability of the programs was ensured.

3.2.2 Number Representation

The critical question in multiple-precision arithmetic programs is how to represent long numbers which apparently do not fit into a register or memory word. Using radix B positional notation an integer u can be written as

$$u = \sum_{i=0}^{n_B-1} u_i B^i, \qquad 0 \leq u_i < B,$$

with u_i the digits of u. In our implementation B is the largest power of 2 such that one digit u_i fits into one memory word, and the instruction set supports radix B unsigned double precision arithmetic. Therefore B is usually 2^{16} or 2^{32}. For simplicity we will concentrate on $B = 2^{16}$, but the programs are designed for both $B = 2^{16}$ and $B = 2^{32}$ [4].

Let us define

```
typedef unsigned short BigNumDigit
typedef BigNumDigit  * BigNum,
```

then a long integer u is essentially a pointer to an array of consecutive digits of type *BigNumDigit*. Usually we work with pairs (u, n_B), where n_B is the number of digits of u in radix B notation. By applying simple pointer arithmetic we may access arbitrary contiguous parts of the *BigNum* u. This is especially useful when we implement splitting algorithms.

In addition to the type BigNum a type *BigZ* is defined, including the sign and size information. We define:

```
struct BigNumHeader {
unsigned long   Size;
int             Sign;
}
```

[4] See Section 3.2.6 for details.

```
struct BigZStruct {
struct BigNumHeader Header;
BigNum              Digits;
}
typedef struct BigZStruct *BigZ;
```

Although our procedures exclusively work with numbers of type BigNum, simple mechanism to convert from one type to the other are provided. In particular high level procedures on top of the basic multiplication algorithms may profit from them. Calling the function *BnCreate()*, a BigNum is created with

```
BigNum BnCreate(Sign, Size) ...
{
BigNum u;

u = malloc(sizeof(struct BigNumHeader) + Size * sizeof(BigNumDigit));
u = (BigNum)(((struct BigNumHeader *) u) +1);
...
/* set sign, set size, initialize u */
...
return(u);
}.
```

We convert a BigNum u into a BigZ z by using the macro *BNTOBZ(u)*

```
#define BNTOBZ(u)       (BigZ)(((struct BigNumHeader *)u)-1)
```

and vice versa

```
#define BZTOBN(z)       ((z)->Digits).
```

At certain situations the resulting overhead may be too disadvantageous. This is true in particular for the fast convolution algorithms and a simplified function to create BigNum's is used there internally.

3.2.3 Memory Management

During execution of the multiplication algorithms, additional temporary workspace has to be provided. A sufficient amount of workspace could be allocated in advance and self-developed memory management procedures could control it. On the other hand the memory management provided by the programming language and the operating system could be used and the memory could be allocated and deallocated whenever it is needed. In [33] the authors state: "Many important arithmetic computations run an order of magnitude faster when all forms of storage allocation are removed from the inner loop, storing intermediate and final results exclusively in the memory area used by input variables. Yet some advanced form of automatic storage management and garbage collection must be part of any useful arithmetic package ... "

The two different forms of memory management essentially can be characterized by the time requirements for allocation and deallocation of memory. If we allocate the memory in advance we get along with just one function call for allocation and deallocation of memory. Dynamic storage allocation may force us to allocte and deallocate memory a few hundreds, thousands or even more times. The SCHÖNHAGE-STRASSEN algorithm (1982), for example, requires for splitting two 65536-bit numbers into blocks of bits 512 allocation and deallocation function calls, when we apply one level of recursion. Two levels of recursion lead to 8704 function calls. Table 3.1 shows the times measured. Thereby, the "standard" memory model can address only 64 k of memory whereas the "huge" memory model overcomes this restriction [5].

Table 3.1: Time requirements for allocation and deallocation routines.

number of function calls	1	512	8704
standard memory model	0.03 msec	7.8 msec	131 msec
huge memory model	0.2 msec	48 msec	368 msec

Careful comparison of the values of table 3.1 and the run time values of chapter 4 reveals that the memory management is not crucial. That's the

[5] See section 3.2.1 and section 3.2.6 for more details.

reason we adopt the dynamic memory concept. Note, that the multiplication procedures are also consistent with the BigNum package in that way.

3.2.4 User Interface

External user functions on the one hand and internal system functions on the other hand realize requirements which are partly contradictory to each other. Above all, user functions have to provide a convenient user interface which usually leads to an unpopular overhead. Efficiency in space and time are taken into account by the often less user friendly system functions. Recursive functions always call system functions internally to avoid overhead.

All but the classical multiplication algorithm [6] employ the identical syntax for the user function

```
void FunctionName(u, ul, v, vl)
BigNum        u, v;
unsigned long ul, vl;
```

and compute the product uv if $ul \geq vl$ and $u_i = v_i = 0$ for $i \geq \frac{ul}{2}$. The result uv is stored in the place of u.

Note, that all multiplication algorithms, with the one exception of the classical one, cannot be used directly for arbitrary n-bit numbers. For example the SCHÖNHAGE-STRASSEN algorithm (1971) requires $n = 2^k$ (equation 2.16), with $ul \cdot word\ size = n$. The user functions adjust inputs of large numbers with arbitrary length to the requests of the specific algorithm, and then call the system functions.

3.2.5 Time Measurement

Time measurements are essential for the comparison of different multiplication algorithms. The function call of a specific algorithm is augmented by calls to the function *TimeInterval()* which prints the real time consumed in seconds and centiseconds. Since all time measurements were done in a single-user and single-task environment the real time reflects the cpu time spent.

[6] The classical algorithm is a function of the BigNum package and is described in section 2.3 in more detail.

```
#define START    1
#define STOP     0
.
.
TimeInterval(START)

/* execute multiplication algorithm */

TimeInterval(STOP)
.
.
```

TimeInterval() calls the Turbo C library function ftime(), a function which is portable to UNIX System V systems.

```
void TimeInterval(Type)
int  Type;
{
   static struct timeb TimeStart;
   struct timeb        TimeStop;

   if (Type == START)
      ftime(&TimeStart);

   else {
      ftime(&TimeStop);

      /* calculate time in seconds
         and     centiseconds,
         print time                    */
   }
}
```

The function ftime() determines the current time and fills in the fields of the variable passed to it. A variable of type "struct time" contains four fields, but we use only the ones for "time" and "millitm". The time field provides the time in seconds since 00 : 00 : 00 Greenwich Mean Time (GMT), January 1, 1970. The millitm field is the fractional part of a second in milliseconds. At first this may suggest a precision in the order of milliseconds for the time

measurements. In fact, the precision is determined by the hardware of the computer and the frequency of the time updates, which is about $\frac{1}{20}$ sec in our case.

3.2.6 Conditional Compilation

The programming language C supplies an interesting feature of conditional compilation. Making compilation conditional on whether certain identifiers are defined or undefined, or whether a constant is zero or nonzero, is useful in the debugging or testing phase of a program. Debugging and test statements can be turned on or off by a single #define. Also, alternative pieces of code can be compiled and used. This is especially useful in adapting a program to different requirements of specific devices, and thus in providing portability.

Four applications of conditional compilation are included:

- **constant COUNT:** Counting how often individual functions get called.
- **constant REC:** Determining the depth of recursive functions.
- **constant HUGE:** Selecting a memory model (more or less than 64 k).
- **constant DIGITon16BITS:** Choosing word size 16 bit or 32 bit.

If COUNT is set to 1 every function call of an individual function, including the functions of the BigNum package, are counted. During program development it was checked how often the functions were called and whether a function was crucial with respect to the performance of an algorithm. As a result some functions were modified or completely replaced by macros. COUNT is also a good tool in a final analysis of an algorithm.

The performance of recursive algorithms is significantly influenced by the depth at which recursion is stopped. The final versions of the algorithms employ the well-defined breakeven points in the decision when to stop the recursion. This version requires REC to be 1. During the test phase REC is set to 0 and the depth of recursion is regulated flexibly by global variables.

The Intel 8088 microprocessor has a segmented memory architecture. Its total address space is 1 M, but it is designed to address directly only 64 k of memory, called a segment, at a time. A memory address is composed of two 16-bit values: the segment address and the offset. The resulting

20-bit address is accomplished by shifting the value of the segment register 4 bits to the left and adding the offset. A 20-bit address is not unique to a specific segment/offset pair, since the segment and offset bits overlap in its calculation. The standard memory model HUGE is set ot zero can address 64 k by using 16-bit pointers, which contain the offset. Pointer arithmetic is fast, since it is done with the offset and without worrying about the segment.

In the huge memory model HUGE is set to one the pointers are 32 bits long and contain both a segment address and an offset. By normalizing the pointers, a unique relation between 20-bit addresses and segment/offset pairs is guaranteed. That is the only way which allows the usage of the comparison operators $>, \geq, <, \leq, =$ in a memory space larger than 64 k. Unfortunately, pointer arithmetic is quite time consuming when huge pointers are engaged. Also, different functions have to be used to allocate and deallocate memory and pointers have to be explicitely of type huge.

The programs are developed and tested for radix $B = 2^{16}$, reflecting the word size of the Intel 8088. If DIGITo16BITS is defined, the 16-bit implementation is chosen. Since the parameters of the fast convolution algorithms often are multiples of 2^{16}, they are especially well-suited for a 16-bit implementation. By minor modifications of the splitting process and its inverse [7] and the fast calculation of the modulo corresponding to lemma 2.1 [8] all procedures are portable to 32-bit processors.

3.3 Implementation of the Multiplication Algorithms

3.3.1 Classical Algorithm

The classical multiplication algorithm is adopted from the BigNum package. Using algorithm C (section 2.3) the implementation is straightforward. An advantage of the classical multiplication algorithm is its high degree of flexibility, which allows to employ it for numbers of arbitrary length.

[7] In the S1 algorithm the usage of the functions BnnAssignDigits() and BnnProdFromDigits() have to be augmented by BnnAssignDigitsAndBits() and BnnProdFromDigitsAndBits(), respectively.

[8] The functions BnnModFDigit() and BnnModFBlock() must be adapted to the requirements of a 32-bit implementation.

The description of the function is as follows:

Function declaration:

```
int BnnMultiply (p, pl, u, ul, v. vl)
BigNum          p, u, v;
unsigned long pl, ul, vl;
```

Input parameters:

$p = (p_{pl-1}, \ldots, p_0)_B$, sum [9]
$u = (u_{ul-1}, \ldots, u_0)_B$, factor
$v = (v_{vl-1}, \ldots, v_0)_B$, factor

The length of the operands must be such that $pl \geq ul + vl$ and $ul \geq vl$. The last condition allows a faster implementation.

Output:

$p = (p_{pl-1}, \ldots, p_0)_B = p + u \cdot v$
Return value: carry out $p_{pl} \in \{0, 1\}$.

The BigNum package includes also a high-level multiplication function, *BzMultiply()*, which handles signed integers. Furthermore, the user is freed from caring about admissible lengths of the operands.

Function declaration:

```
BigZ BzMultiply(y, z)
BigZ y, z;
```

Output:

Return value: $p = y \cdot z$.

[9] $B = 2^{word\ size}$

3.3.2 KARATSUBA's Algorithm

After supplying the description of the function, the implementation of the algorithm will be described in some detail.

Function declaration:

```
void BnnKaratsuba(p, u, v, pl)
BigNum          p, u, v;
unsigned long pl;
```

Input parameters:

$p = (p_{pl-1}, \ldots, p_0)_B, = (0, \ldots, 0)_B$ product
$u = (u_{\frac{pl}{2}-1}, \ldots, u_0)_B,$ factor
$v = (v_{\frac{pl}{2}-1}, \ldots, v_0)_B,$ factor

The length of the factors u and v must be $\frac{pl}{2}$. The original version requires

$$pl = 2^k \tag{3.1}$$

and the user function (section 3.2.4) adjusts inputs of an arbitrary length corresponding to equation 3.1. When called internally from a convolution algorithm this condition is relaxed, and we have

$$pl = 2^l \cdot lBreak, \qquad \text{where } lBreak \leq BREAKEVEN^{10}. \tag{3.2}$$

Output:

$$p = (p_{pl-1}, \ldots, p_0)_B = u \cdot v$$

Conditions 3.1 and 3.2 are not inherent to the KARATSUBA algorithm in general. The implementation can be modified such that for an arbitrary size pl it is

$$size(u) \neq \frac{pl}{2}, \quad size(v) \neq \frac{pl}{2}, \quad \text{and } size(u) \neq size(v). \tag{3.3}$$

[10] BREAKEVEN is the break even point of the KARATSUBA algorithm and the classical one.

Figure 3.1: Programming code of the KARATSUBA algorithm.

```
BnnKaratsuba(p, u, v, l)
BigNum        p, u, v;
unsigned long l;
{
   if (l > BREAKEVEN) { /* apply Karatsuba for large l */

       p0 = p;         u0 = u;        v0 = v;
       p1 = p + l/4;   u1 = u + l/4;  v1 = v + l/4;
       p2 = p + l/2;

       /* compute 1. partial product */

       BnnKaratsuba(aux, u0, v0, l/2);
       p0 = p0 + aux;
       p1 = p1 + aux;

       /* compute 2. partial product */

       BnnKaratsuba(aux, u1, v1, l/2);
       p1 = p1 + aux;
       p2 = p2 + aux;

       /* compute 3. partial product */

       BnnKaratsuba(aux, u1-u0, v0-v1, l/2);
       p1 = p1 +aux;
    }
    else {    /* apply classical algorithm for small l */

       BnnMultiply(p, l, u, l/2, v, l/2);
    }
}
```

The comparison of the multiplication algorithms does not necessarily require this more universal implementation. On the other hand, the fast convolution algorithms automatically meet equation 3.2, and by implementing the algorithm in the specialized form as described an unwelcome overhead was avoided.

The programming code of the KARATSUBA algorithm is given in figure 3.1. Besides its shortness and clearness, it perfectly demonstrates the way pointer arithmetic is used to access arbitrary parts of a BigNum. Simple pointer arithmetic allows to access $u0$ and $v0$, the "least significant half", and $u1$ and $v1$, the "most significant half" of u and v. Temporary results are stored in the auxiliary variable "aux". Note, that all additions "+" and subtraction "-" are multiple-precision operations. Furthermore, the computation of the third partial product involves some additional coding to consider $u1 - u0 < 0$, $v0 - v1 < 0$ and $(u1 - u0)(v0 - v1) < 0$.

3.3.3 Basic Elements of Fast Convolution Algorithms

The fundamental structure and the principles of the fast convolution algorithms are identical. Therefore, an outline of the basic elements of their implementation will precede the description of the individual functions.

Convolution algorithms are most powerful when multiplying two large numbers u and v of equal length. The classical multiplication algorithm, for example, achieves a linear bit complexity $O(n)$ if one of the factors u or v is of length 1. The fast convolution functions are written for inputs of equal length, since this reflects best the most reasonable application area of the algorithms. By minor modifications they can be applied to numbers of different sizes.

Parameter Handling

The basic parameters of the fast convolution algorithms are n_2, n_B, l, ω_{n_B} and m. The parameters of the SCHÖNHAGE-STRASSEN algorithm (1971) are given by equation 2.16 and table 2.1, and the parameters of both the SCHÖNHAGE-STRASSEN algorithm (1982) and the ordinary convolution algorithm are defined in equation 2.23 and table 2.2. Two global tables are used to guarantee a fast retrieval of the parameters. In tables 3.2 and 3.3 extracts of both global tables are shown. Observe, that the distance between

Table 3.2: Global parameter table for S1 algorithm.

n_2	n_2	n_B	l	$\log_2(\omega_{n_B})$	$\log_2(m-1)$
$(00128)_{10}$	$(00080)_{16}$	$(008)_{16}$	$(010)_{16}$	$(8)_{16}$	$(020)_{16}$
$(00256)_{10}$	$(00100)_{16}$	$(010)_{16}$	$(010)_{16}$	$(4)_{16}$	$(020)_{16}$
$(00512)_{10}$	$(00200)_{16}$	$(010)_{16}$	$(020)_{16}$	$(8)_{16}$	$(040)_{16}$
$(01024)_{10}$	$(00400)_{16}$	$(020)_{16}$	$(020)_{16}$	$(4)_{16}$	$(040)_{16}$
$(02048)_{10}$	$(00800)_{16}$	$(020)_{16}$	$(040)_{16}$	$(8)_{16}$	$(080)_{16}$
$(04096)_{10}$	$(01000)_{16}$	$(040)_{16}$	$(040)_{16}$	$(4)_{16}$	$(080)_{16}$
$(08192)_{10}$	$(02000)_{16}$	$(040)_{16}$	$(080)_{16}$	$(8)_{16}$	$(100)_{16}$
$(16384)_{10}$	$(04000)_{16}$	$(080)_{16}$	$(080)_{16}$	$(4)_{16}$	$(100)_{16}$
$(32768)_{10}$	$(08000)_{16}$	$(080)_{16}$	$(100)_{16}$	$(8)_{16}$	$(200)_{16}$
$(65536)_{10}$	$(10000)_{16}$	$(100)_{16}$	$(100)_{16}$	$(4)_{16}$	$(200)_{16}$

Table 3.3: Global parameter table for S2 and ordinary convolution algorithm.

n_2	n_2	n_B	l	$\log_2(\omega_{n_B})$	$\log_2(m-1)$
$(128)_{10}$	$(080)_{16}$	$(08)_{16}$	$(10)_{16}$	$(0C)_{16}$	$(30)_{16}$
$(160)_{10}$	$(0A0)_{16}$	$(08)_{16}$	$(14)_{16}$	$(0C)_{16}$	$(30)_{16}$
$(192)_{10}$	$(0C0)_{16}$	$(08)_{16}$	$(18)_{16}$	$(10)_{16}$	$(40)_{16}$
$(224)_{10}$	$(0E0)_{16}$	$(08)_{16}$	$(1C)_{16}$	$(10)_{16}$	$(40)_{16}$
$(256)_{10}$	$(100)_{16}$	$(08)_{16}$	$(20)_{16}$	$(14)_{16}$	$(50)_{16}$
$(288)_{10}$	$(120)_{16}$	$(08)_{16}$	$(24)_{16}$	$(14)_{16}$	$(50)_{16}$
$(320)_{10}$	$(140)_{16}$	$(10)_{16}$	$(14)_{16}$	$(06)_{16}$	$(30)_{16}$
$(384)_{10}$	$(180)_{16}$	$(10)_{16}$	$(18)_{16}$	$(08)_{16}$	$(40)_{16}$
$(448)_{10}$	$(1C0)_{16}$	$(10)_{16}$	$(1C)_{16}$	$(08)_{16}$	$(40)_{16}$
$(512)_{10}$	$(200)_{16}$	$(10)_{16}$	$(20)_{16}$	$(0A)_{16}$	$(50)_{16}$

two adjacent suitable n_2's is quite different for the two different versions of the SCHÖNHAGE-STRASSEN algorithm.

For small n_2, $n_2 < (80)_{16}$, other multiplication algorithms are used. The careful study of the tables reveals some interesting features of the parameters. For example, since n_2 is always a multiple of the word size the algorithms are simplified signigicantly. Also, for *word size* = 16 and algorithm S1 the splitting of an n_2-bit number into n_B blocks of l bits is done with complete digits. For S2 and the ordinary convolution algorithm shift and bit masking procedures must be employed to access parts of digits. The application of the fast calculation of the modulo corresponding to lemma 2.1 can be elegantly solved in a 16-bit implementation, since $log_2(m-1)$ is a multiple of 16.

There are some additional parameters which are frequently used. These parameters are essentially calculated once and kept in a data record which is passed on to the functions.

We define

```
struct SstParameters {
unsigned long          SizeN;
unsigned long          SizeEl;
short int              OffsetBits;
unsigned long          DigitsPerL;
...
}
typedef struct SstParameters SstPar;
```

with

$$SizeN = \frac{n_2}{sizeof(BigNumDigit)} + 1 \tag{3.4}$$

$$SizeEl = \frac{\log_2(m-1)}{sizeof(BigNumDigit)} + 1 \tag{3.5}$$

$$OffsetBits = l \bmod sizeof(BigNumDigit) \tag{3.6}$$

$$DigitsPerL = \frac{l}{sizeof(BigNumDigit)}. \tag{3.7}$$

Representation of Coefficient Blocks

In all fast convolution algorithms the n_2-bit numbers u and v are split into n_B blocks of l bits (section 2.5.7, 2.5.8 and 2.5.9). A new type, an array of BigNums, is defined to keep the n_B BigNums:

```
typedef BigNum * BigNArray;
```

Generally, the function BnCreate() is used to create BigNums (section 3.2.2). Since BnCreate() also provides space for the size and sign information, an overhead is introduced which is not compensated by any advantages when used for fast convolution algorithms. Therefore a new function *BnnCreateArray()* was written, which ignores sign and size:

```
BigNArray BnnCreateArray(SizeArray, SizeEl)
unsigned long SizeArray, SizeEl;
{
   BigNArray a;
   unsigned long l;

   a = (BigNArray)malloc(SizeArray * sizeof(BigNum));
   for (l = 0; l < SizeArray; l++) {
         *(a+l) = (BigNum)malloc(SizeEl * sizeof(BigNumDigit));

         /* initialize *(a+l) */
    }
    return(a);
}
```

Note, that $SizeArray = n_B$. The size "SizeEl" of each coefficient block is determined by the modulus m and given in equation 3.5. Observe, that for all $u_i = u \bmod m$ the first digit is one if $u_i \equiv -1 \bmod m$ or zero in all other cases.

Example 3.1
In the example of section 2.5.9 we split two 53248-bit numbers u and v into 128 blocks of 416 bits and employed $m = 2^{896} + 1$. Therefore we choose

$$SizeEl = \frac{log_2(2^{896} + 1 - 1)}{16} + 1 = \frac{896}{16} + 1 = 57.$$

Modular Arithmetic

The fast convolution algorithms, we implemeted, work in a finite field or ring Z of integers with arithmetic carried out modulo an integer m. Therefore, procedures which support basic modular multiple-precision arithmetic operations have to be provided. In particular, functions for addition, subtraction, multiplication and shift left (multiplication by powers of two) have to be supplied.

Let u, v be two multiple-precision numbers with $0 \leq u, v < m$. Since the first digit of u and v is only used if $u \equiv -1 \bmod m$ or $v \equiv -1 \bmod m$, respectively, the size of the sum $u + v$, the difference $u - v$ and the product $u \cdot v$ is given by

$$size(w) = size(u + v) \leq SizeEl \tag{3.8}$$

$$size(w) = size(u - v) \leq SizeEl \tag{3.9}$$

$$size(w) = size(u \cdot v) \leq (SizeEl - 1) \cdot 2, \tag{3.10}$$

where $SizeEl$ is defined in equation 3.5.

By writing w in radix B notation, with $B = 2^{16(SizeEl-1)}$, we have

$$w = w_o + w_i B, \qquad 0 \leq w_i < B \tag{3.11}$$

and by applying lemma 2.1 we get

$$w \bmod m = w_0 - w_i. \tag{3.12}$$

Thus, fast arithmetic operations can be incorporated. Two functions are provided to compute 3.12:

- *BnnModFDigit()* for $w_i < 2^{16}$,
- *BnnModFBlock()* for $0 \leq w_i < 2^{16(SizeEl-1)}$.

BnnModFDigit() is used by the modular addition and subtraction, and BnnModFBlock() by the modular multiplication. Modular multiple-precision arithmetic operations are basically a straightforward extension of the non-modular operations. By its simplicity the modular addition is most qualified to illustrate the principle:

```
void BnnAddMF(u, v, l)
BigNum        u, v;
unsigned long l;
{
    u = u + v;
    if (GetFirstDigit(u) != 0)
          BnnModFDigit(u, l);
}
```

The next example of modular multiplication is included, since it is a quite fundamental procedure. The modular multiplication is used by the fast convolution algorithms whenever pairs of Fourier transforms are multiplied directly. Dependent on the size of the factors the appropriate multiplication algorithm, either the classical or the KARATSUBA algorithm, is chosen.

```
void BnnMultiplyMF(u, v, l)
BigNum        u, v;
unsigned long l;
{
   if (...)

      /* handle special case, u or v is -1 */

   else {
      auxl = (l-1) * 2;
      if (auxl < BREAKEVEN)  /* apply classical alg.   */

         BnnMultiply(aux, auxl, u, l-1, v, l-1);

      else                   /* apply KARATSUBA's alg. */

         BnnKaratsuba(aux, u, v, auxl);
   }
   BnnModFBlock(aux, l);
   u = aux;
}
```

In contrast to the modular addition, the modular multiplication required an auxiliary variable "aux". This is due to the fact, that the size of $u \cdot v$

extends the size of u, since $size(u) = size(v) = SizeEl$ and $size(u \cdot v) \leq (SizeEl - 1) \cdot 2$ (equation 3.10).

Mod z Fast Fourier Transform

The principal structure of the fast Fourier transform procedure is given in figure 2.4. With minor modifications the FFT procedure can be employed by the multiple-precision fast convolution algorithms. In particular, arithmetic operations have to be replaced by fast modular multiple-precision operations. Multiplications by powers of ω_{n_B} and n_B^{-1} become modular shift left operations, since both ω_{n_B} and n_B^{-1} are powers of two.

The performance of the fast Fourier transform was improved by employing an iterative version of the algorithm [2] [10], and the storage requirements were kept at a minimum by overwriting the original data array with the temporary arrays [11].

3.3.4 Ordinary Fast Convolution Algorithm

The scheme of the ordinary fast convolution algorithm is illustrated in figure 2.6. The description of the function is as follows:

Function declaration:

```
void BnnOrdinaryConvolution(u, v, l)
BigNum          u, v;
unsigned long l;
```

Input parameters:

$u = (u_{l-1}, \ldots, u_0)_B$, factor
$v = (v_{l-1}, \ldots, v_0)_B$, factor

The "most significant half" of u and v must be 0

$$u_i = v_i = 0, \qquad \text{if } i \geq \frac{l}{2}. \tag{3.13}$$

[11] See chapter 2.5.4 and figure 2.5 for details.

Furthermore, equation 2.23 must be satisfied:

$$n_2 = \nu 2^k, \qquad k-1 \leq \nu \leq 2k-1, \qquad k \geq 4$$

where

$$(l-1)(sizeof(BigNumDigit)) \cdot 8 = n_2. \tag{3.14}$$

The user function (section 3.2.4) adjusts inputs of an arbitrary length correspondingly.

Output:

$u = (u_{l-1}, \ldots, u_0)_B = u \cdot v.$

3.3.5 SCHÖNHAGE-STRASSEN's Algorithm (1971)

Algorithm S1 in section 2.5.8 decribes the SCHÖNHAGE-STRASSEN algorithm in its original version from 1971, and figure 2.9 illustrates the structure of the algorithm. Before emphasizing the characteristics of the algorithm, the decription of the function will be supplied:

Function declaration:

```
void BnnSstVers1(u, v, l)
BigNum          u, v;
unsigned long l;
```

Input parameters:

$u = (u_{l-1}, \ldots, u_0)_B$, factor
$v = (v_{l-1}, \ldots, v_0)_B$, factor

The size l has to satisfy equation 2.16, $n_2 = 2^k$, where

$$(l-1)(sizeof(BigNumDigit)) \cdot 8 = n_2.$$

The user function (section 3.2.4) adjusts inputs of an arbitrary length correspondingly.

Output:

$$u = (u_{l-1}, \ldots, u_0)_B = u \cdot v \bmod (2^{n_2} + 1).$$

The SCHÖNHAGE-STRASSEN algorithm (1971) computes uv mod $(2^{n_2} + 1)$ and the exact product uv can be achieved by adding an appropriate number of leading zeros. In contrast to the ordinary convolution algorithm the *negative wrapped* convolution is employed. This requires the multiplication by powers of ψ in step 1 and step 3 of algorithm S1, where $\psi^2 = \omega_{n_B}$. Since ψ is a power of two the multiplications reduce to a fast modular shift left operation. Recall, that the algorithm requires to calculate the w_i's twice, once mod n_B and once mod $(2^{2l}+1)$, and that w_i mod $n_B(2^{2l}+1)$ is recoverd by applying the Chinese remainder theorem. Corresponding to equation 2.20 the w_i are in the range

$$-(n_B - 1 - i)2^{2l} < w_i < (i+1)2^{2l}.$$

Observe, that w_i can be negative. The calculation of $w_i' = w_i \bmod n_B$ is done in the interval $[0, n_B - 1]$, the calculation of $w_i'' = w_i \bmod (2^{2l} + 1)$ in the interval $[0, 2^{2l}]$. By applying equation 2.21, $w_i = w_i'' + [((w_i' - w_i'') \cdot (2^{2l} + 1)^{-1}) \bmod n_B] \cdot (2^{2l} + 1)$, the unique values of w_i can be recovered. Take

$$w_i \bmod n_B(2^{2l}+1) \equiv w_i - n_B(2^{2l}+1), \text{ if } w_i \geq 2^{2l}(i+1)$$
$$w_i \bmod n_B(2^{2l}+1) \equiv w_i, \text{ if } w_i < 2^{2l}(i+1).$$

Hence, we don't have to care about the sign in the calculation of S1 from step 1 to step 4.

3.3.6 SCHÖNHAGE-STRASSEN's Algorithm (1982)

The SCHÖNHAGE-STRASSEN algorithm (1982) is an improved version of the original version from 1971. By choosing different parameters, the application of the Chinese remainder theorem is avoided and the new algorithm S2 comprises steps 1, 2, 3 and 6 of the original algorithm S1. Furthermore, n_2 is no longer restricted to powers of two:

Function declaration:

```
void BnnSstVers2(u, v, l)
BigNum          u, v;
unsigned long l;
```

Input parameters:

$u = (u_{l-1}, \ldots, u_0)_B$, factor
$v = (v_{l-1}, \ldots, v_0)_B$, factor

The size l has to satisfy equation 2.23,

$$n_2 = \nu 2^k, \qquad k - 1 \leq \nu \leq 2k - 1, \qquad k \geq 4,$$

where

$$(l-1)(sizeof(BigNumDigit)) \cdot 8 = n_2.$$

The user function (section 3.2.4) adjusts inputs of an arbitrary length correspondingly.

Output:

$$u = (u_{l-1}, \ldots, u_0)_B = u \cdot v \bmod (2^{n_2} + 1).$$

Analogously to the considerations in section 3.3.5, the w_i can be negative. In steps 1 through step 3 of algorithm S2 the calculations are done in the interval $[0, m-1]$. Finally, in step 4 we take

$w_i \bmod n_B(2^{2l} + 1) \equiv w_i - m$, if $w_i \geq 2^{2l}(i+1)$
$w_i \bmod n_B(2^{2l} + 1) \equiv w_i$, if $w_i < 2^{2l}(i+1)$,

and the unique w_i's are recovered.

Chapter 4

Results and Discussion

4.1 Preliminary Remarks

This chapter discusses the performance of the multiplication algorithms which have been implemented corresponding to the principles developed in chapters 2 and 3. Five algorithms will be examined:

- the classical algorithm (*C*)
- KARATSUBA's algorithm (1962) (*K*)
- the ordinary fast convolution algorithm (*O*)
- the original SCHÖNHAGE-STRASSEN algorithm (1971) (*S1*)
- the modified SCHÖNHAGE-STRASSEN algorithm (1982) (*S2*).

The algorithms were tested with random numbers. The correctness of the results has been checked by comparing the results of the implemented algorithms and the classical algorithm of the BigNum package. Thereby, the correctness of the classical algorithm was verified by applying FERMAT's theorem [1].

Both the modular multiplication (mod (2^n+1)) and the non-modular multiplication will be discussed, since both forms are employed by specific applications (chapter 1). In general, the time measurements are done twice, once

[1] FERMAT's theorem: Let p be prime. Then for every a such that $\gcd(a,p) = 1$, $a^{p-1} \bmod p = 1$.

for the "standard" and once for the "huge" memory model [2]. The empirically tested timing error is less than 1% and stems from the finite resolution of the system clock (about $\frac{1}{20}$ seconds [3]). It could easily be reduced by increasing the number of computations within the timing loops. However, some precision had to be sacrificed to get the results within a reasonable span of time.

Section 4.2 is dedicated to the KARATSUBA algorithm and focuses on the dependency on the depth of recursion. A comparative discussion of the five multiplication algorithms is given in section 4.3, and section 4.4 enriches the presentation by results from other authors. Finally, section 4.5 completes the chapter with a summary and an outlook.

4.2 KARATSUBA's Algorithm - The Influence of the Depth of Recursion

Table 4.1 shows the data of the time measurements of the KARATSUBA algorithm for the instance of the non-modular multiplication and the huge memory model. The depth of the recursion is ranging from 1 to 13. The splitting algorithm was stopped if the size of the numbers was equal to the size of a short integer, and an empty entry reflects that no further splitting was done. The symbol "– – –" denotes that the test exceeded the memory capacities of the system used. The size of the factors is n_2, the size of the product is $2n_2$. For each n_2 the best time value is framed.

The analysis of the individual rows of table 4.1 reveals an interesting feature of the algorithm. The execution times decrease continously until the minimal value is reached and increase continously thereafter. For small depths of the recursion the algorithm can't develop its complete power, whereas for large depths the overhead of the KARATSUBA algorithm begins to dominate. Thereby the case of too many recursion levels imposes the more serious time penalty. For $n_2 = 64$, $n_2 = 128$ and $n_2 = 256$ the size of the input numbers remains below the size for the break even point of the KARATSUBA and the classical algorithm, which will be shown later.

[2] See section 3.2.1 and section 3.2.6 for the explanation of the "standard" and the "huge" memory model.

[3] See section 3.2.5 for details.

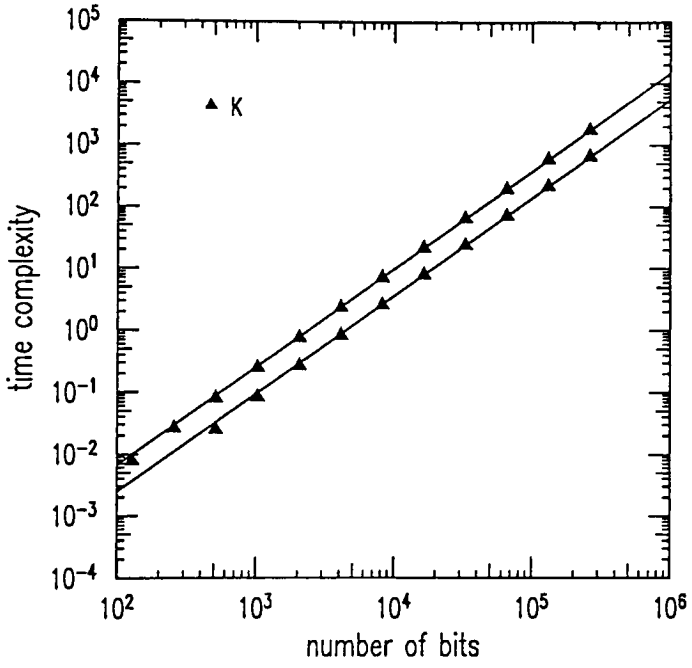
Table 4.1: KARATSUBA's algorithm with the depth of recursion varied from 1 through 13 (non-modular multiplication, huge memory model). The boxed entries are minima. (Execution times in seconds.)

n_2	1	2	3	4	5	6	7
64	0.0019						
128	0.0037	0.0077					
256	0.0085	0.0134	0.0263				
512	0.0252	0.0305	0.0452	0.0810			
1024	0.0859	0.0848	0.0998	0.1445	0.2527		
2048	0.3153	0.2756	0.2711	0.3177	0.4513	0.7758	
4096	1.215	0.987	0.8580	0.8490	0.9890	1.389	2.379
8192	4.762	3.715	3.015	2.645	2.615	3.037	4.254
16384	18.84	14.43	11.26	9.18	8.09	7.99	9.25
32768	75.02	56.86	43.40	34.02	27.83	24.49	24.19
65536	299.36	225.9	170.5	130.8	103.4	84.0	74.0
131072	1196.6	900.5	676.0	512.6	394.2	309.9	253.1
262144	4844.0	3599.6	2691.0	2030.0	1539.9	1183.3	929.7
524288	19451.0	– – –	– – –	– – –	– – –	– – –	– – –

n_2	8	9	10	11	12	13
64						
128						
256						
512						
1024						
2048						
4096						
8192	7.213					
16384	12.92	21.79				
32768	28.07	39.03	65.7			
65536	73.9	84.6	117.6	198.0		
131072	223.3	221.0	255.4	355.3	593.9	
262144	761.7	671.7	665.1	769.9	1066.1	1771.2
524288	– – –	– – –	– – –	– – –	– – –	– – –

KARATSUBA's algorithm has a bit complexity of $O(n_2^{\log_2 3})$, and for a doubling of n_2 a triblicating of the execution times is expected. The diagonal built by the framed numbers meets this expectation within the accuracy of the experiment, and so do all of the parallel diagonals. Figure 4.1 shows two different diagonals of table 4.1 in a double logarithmic representation. The measured values are already very close to the theoretical value of $\log_2 3$ for the slope of the asymptotic behavior, which is illustrated by the straight lines in figure 4.1. Each diagonal represents the values for a specific radix

Figure 4.1: KARATSUBA's algorithm for two different radices n_B.



n_B, where $n_B = 2^{256}$ for the diagonal with the framed values, for example. In terms of arithmetic operations [4] the complexity is $O(n_B^{\log_2 3})$ and the best performance of the algorithm is achieved for $n_B = 2^{256}$. The recursion was stopped when the size of the numbers was equal to the size of the radix n_B.

[4] See section 2.2 for an explanation of complexity in bit operations and complexity in arithmetic operations.

4.3 Performance of the Algorithms

Intuitively one expects that a higher degree of sophistication of an algorithm implies an increase of the time complexity by a constant factor. This section presents and discusses the results of the time measurements and examines the resulting break even points, and the range in which a specific multiplication algorithm is superior to the others.

Table 4.2 shows the data for the modular multiplication in the huge memory model, table 4.3 for the non-modular multiplication in the huge memory model, tables 4.4 and 4.5 show the corresponding data for the standard memory model. The more time consuming memory management and pointer arithmetic of the huge memory model increases the time values for this model. The effect is slightly different for the individual algorithms and depends on the extend allocation and deallocation of memory and pointer arithmetic is used.

Table 4.2: Modular multiplication for the huge memory model (classical (C), KARATSUBA (K), ordinary convolution (O), SCHÖENHAGE-STRASSEN 1971 (S1) and SCHÖNHAGE-STRASSEN 1982 (S2) algorithm). (Execution times in seconds.)

n_2	C	K	O	S1	S2
128	0.0027	0.0045	0.0602	0.0670	0.0657
256	0.0080	0.0097	0.1548	0.1590	0.0872
512	0.0276	0.0273	0.2534	0.2067	0.2090
1024	0.1031	0.0888	0.609	0.5075	0.3402
2048	0.4009	0.2660	1.181	0.788	0.814
4096	1.5795	0.8699	2.362	1.948	1.543
8192	6.270	2.653	5.33	3.332	3.040
16384	24.960	8.069	12.82	8.67	7.09
32768	99.68	24.42	27.84	16.29	16.14
65536	398.35	73.66	70.1	43.45	38.60
131072	1614.9	222.0	149.2	90.6	91.6
262144	6480.0	667.6	– – –	257.3	220.4
524288	25916.2	– – –	– – –	– – –	– – –

Table 4.3: Non-modular multiplication for the huge memory model (classical (C), KARATSUBA (K), ordinary convolution (O), SCHÖENHAGE-STRASSEN 1971 (S1) and SCHÖNHAGE-STRASSEN 1982 (S2) algorithm). (Execution times in seconds.)

n_2	C	K	O	S1	S2
64	0.0006	0.0019	0.0453	0.0626	0.0594
128	0.0019	0.0037	0.0606	0.1419	0.0787
256	0.0068	0.0085	0.1539	0.1894	0.1928
512	0.0257	0.0252	0.2514	0.4554	0.3131
1024	0.1002	0.0848	0.605	0.691	0.727
2048	0.3948	0.2711	1.167	1.620	1.382
4096	1.567	0.8490	2.349	2.737	2.713
8192	6.248	2.615	5.30	6.34	6.02
16384	24.94	7.99	12.75	12.14	14.14
32768	99.64	24.19	27.69	26.93	30.45
65536	398.55	73.2	69.8	58.1	75.2
131072	1614.6	221.0	148.5	130.1	159.3
262144	6479.3	665.1	– – –	– – –	– – –
524288	26597.2	– – –	– – –	– – –	– – –

Table 4.4: Modular multiplication for the standard memory model (classical (C), KARATSUBA (K), ordinary convolution (O), SCHÖENHAGE-STRASSEN 1971 (S1) and SCHÖNHAGE-STRASSEN 1982 (S2) algorithm). (Execution times in seconds.)

n_2	C	K	O	S1	S2
128	0.0012	0.0018	0.0228	0.0267	0.0258
256	0.0035	0.0041	0.0587	0.0632	0.0330
512	0.0121	0.0115	0.0918	0.0793	0.0798
1024	0.0454	0.0365	0.220	0.1949	0.1236
2048	0.1756	0.1133	0.426	0.275	0.292
4096	0.6916	0.3472	0.855	0.685	0.547
8192	2.746	1.056	1.91	1.120	1.059
16384	10.938	3.197	4.66	2.93	2.44
32768	43.67	9.67	– – –	5.50	5.65
65536	174.73	– – –	– – –	– – –	– – –

Table 4.5: Non-modular multiplication for the standard memory model (classical (C), KARATSUBA (K), ordinary convolution (O), SCHÖENHAGE-STRASSEN 1971 (S1) and SCHÖNHAGE-STRASSEN 1982 (S2) algorithm). (Execution times in seconds.)

n_2	C	K	O	S1	S2
64	0.0003	0.0009	0.0180	0.0244	0.0236
128	0.0009	0.0016	0.0227	0.0582	0.0302
256	0.0032	0.0037	0.0587	0.0728	0.0738
512	0.0117	0.0110	0.0905	0.1747	0.1140
1024	0.0445	0.0354	0.218	0.244	0.265
2048	0.1741	0.1111	0.419	0.572	0.494
4096	0.689	0.3430	0.843	0.947	0.970
8192	2.741	1.0530	1.88	2.20	2.14
16384	10.93	3.18	4.58	4.26	5.06
32768	43.72	9.63	– – –	– – –	– – –
65536	174.3	– – –	– – –	– – –	– – –

The SCHÖNHAGE-STRASSEN algorithm is basically a modular multiplication algorithm and calculates the product of two n_2-bit numbers u and v mod $(2^{n_2}+1)$ for suitable n_2's. To get the non-modular product uv the $\frac{n_2}{2}$ leading bits of u and v must be zero. Therefore it is expected that the time to compute the non-modular product of two $\frac{n_2}{2}$-bit numbers is close to the time required to compute the modular product of two n_2-bit numbers. On the other hand, the classical, the KARATSUBA and the ordinary convolution algorithm are in the first instance non-modular multiplicaiton algorithms, which can be extended to a fast modular multiplication according to the principles developed in section 3.3.3 (modular arithmetic). The time required by the modular multiplication of two n_2-bit numbers is close to the time required by a straight-forward conventional non-modular multiplication, but slightly increased. The different characteristics of the SCHÖNHAGE-STRASSEN algorithm on the one hand and the classical and the KARATSUBA on the other hand result in lower break even points for the SCHÖNHAGE-STRASSEN algorithm in case of the modular multiplication as compared to the non-modular multiplication.

Figure 4.2 illustrates the modular multiplication for the huge memory model corresponding to the values of table 4.2. The asymptotic behavior of the algorithms according to the formula for the bit complexity is illustrated by the solid lines. In particular for large numbers of bits n_2 the measured values are already very close to the theoretical curve. Note, that an increase or decrease of the constant factor due to the actual implementation, for example results in a shift of the data parallel to the y-axis in the double logarithmic graph. The form of the curves remains unchanged. The theoretical curves in the figures were calculated by fitting the asymptotic functions to the respective 4 or 5 data points with the highest number of bits using a nonlinear least square regression. Figure 4.3, which is the corresponding graph for the standard memory model, demonstrates that the simplified pointer handling results in slightly reduced break even points. More clearness in the graphical illustration in figures 4.2 and 4.3 is achieved by choosing the modified SCHÖNHAGE-STRASSEN algorithm (1982) to represent the three fast convolution algorithms. This decision is supported by the similarity of the convolution algorithms which is graphically demonstrated in figure 4.4. Note, that the window of the y-axis is zoomed. The ordinary convolution algorithm computes, in contrast to the two versions of the SCHÖNHAGE-STRASSEN algorithm, the standard convolution rather than the negative wrapped convolution. Therefore it is superior for the non-

Figure 4.2: Time complexity of the modular multiplication for the huge memory model.

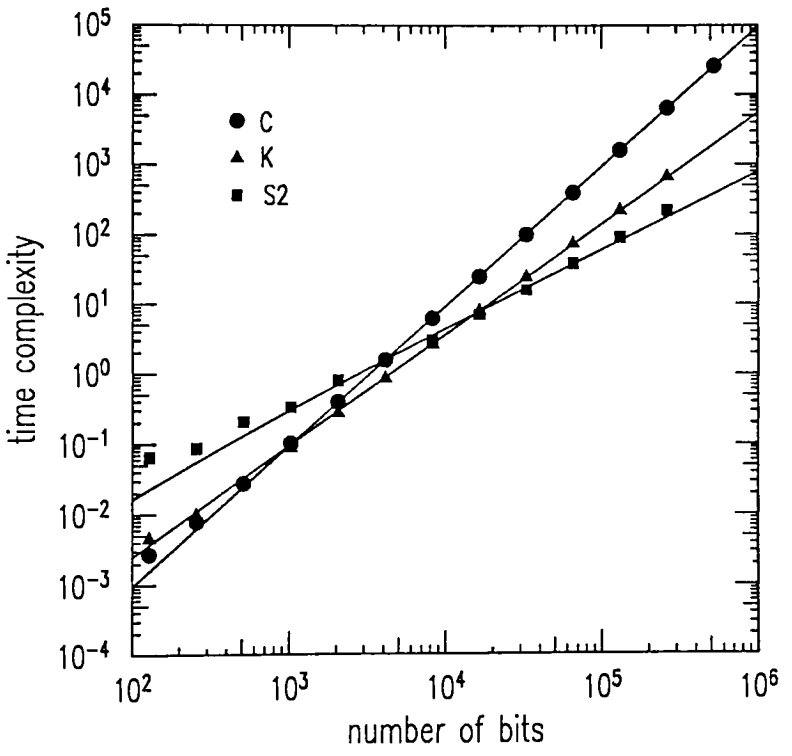


Figure 4.3: Time complexity of the modular multiplication for the standard memory model.

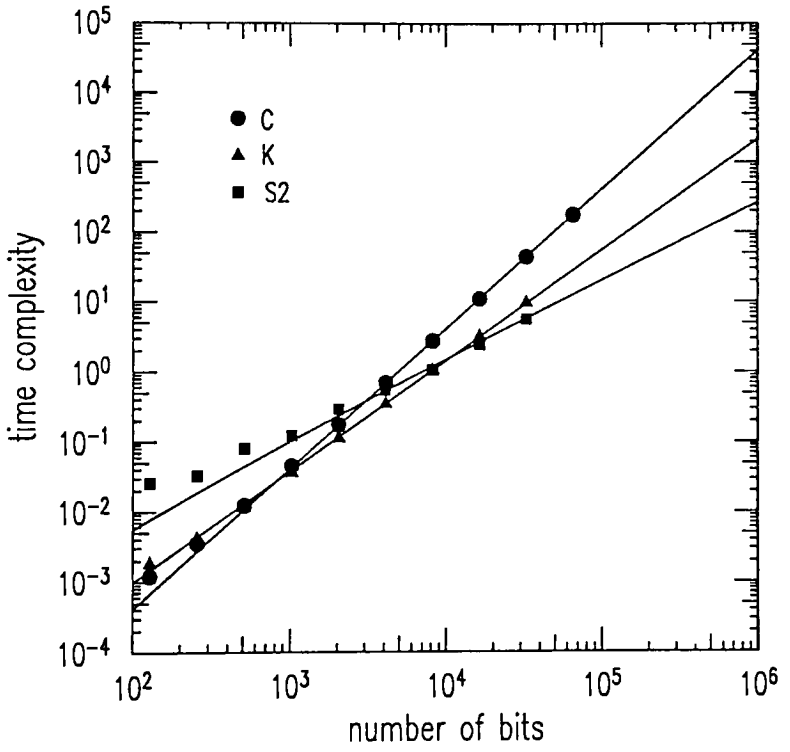
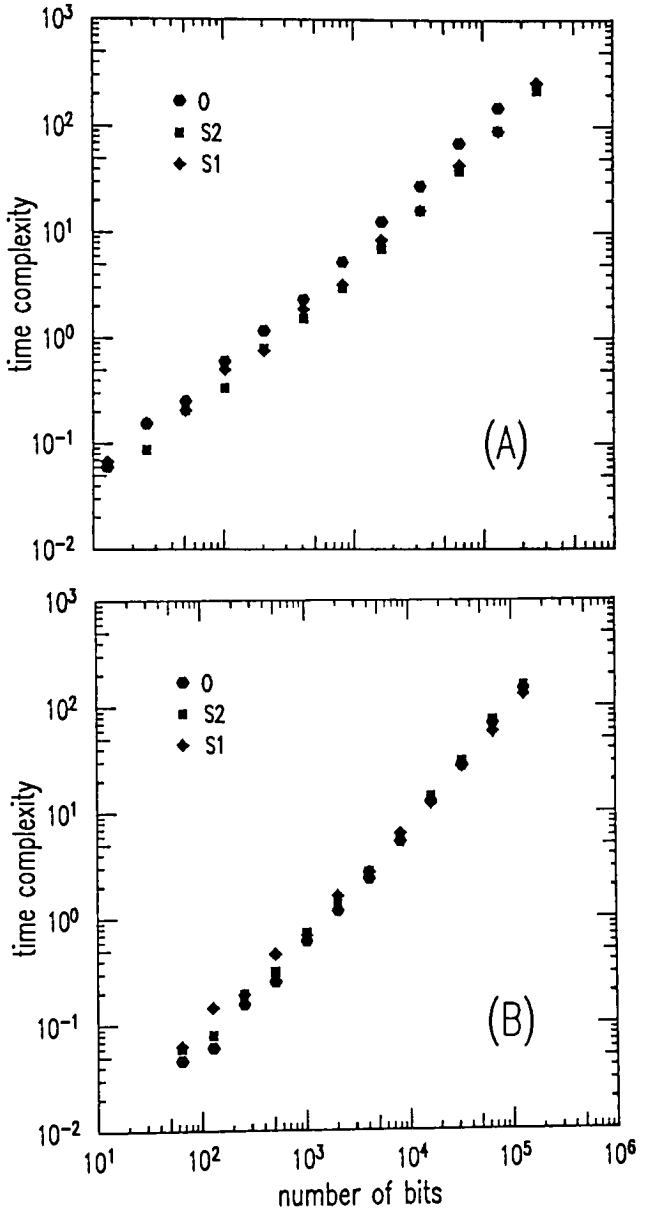


Figure 4.4: Comparison of the time complexity of the convolution algorithms for the huge memory model: (A) modular multiplication, (B) non-modular multiplication.



modular multiplication but since it lost the modular characteristic inherent to the SCHÖNHAGE-STRASSEN algorithm it is inferior for the modular multiplication.

The break even point between the SCHÖNHAGE-STRASSEN algorithm (1982) and the KARATSUBA algorithm for the modular multiplication is close to 12500 bits. This implies that the depth of recursion is always 1 for the size of data measured. The depth of recursion increases to 2 when the size exceeds about $3 \cdot 10^6$ bits. For example, the modular multiplication of two numbers with $n_2 = 262\ 144$ involves one call of the SCHÖNHAGE-STRASSEN algorithm (1982), and the multiplication mod $(2^{262\,144} + 1)$ is reduced to 256 multiplications mod $(2^{2304} + 1)$, each done by recursively calling the KARATSUBA algorithm which calls the classical multiplication algorithm when the splitted numbers are 288 bits long.

Figure 4.5: Time complexity of the non-modular multiplication for the huge memory model.

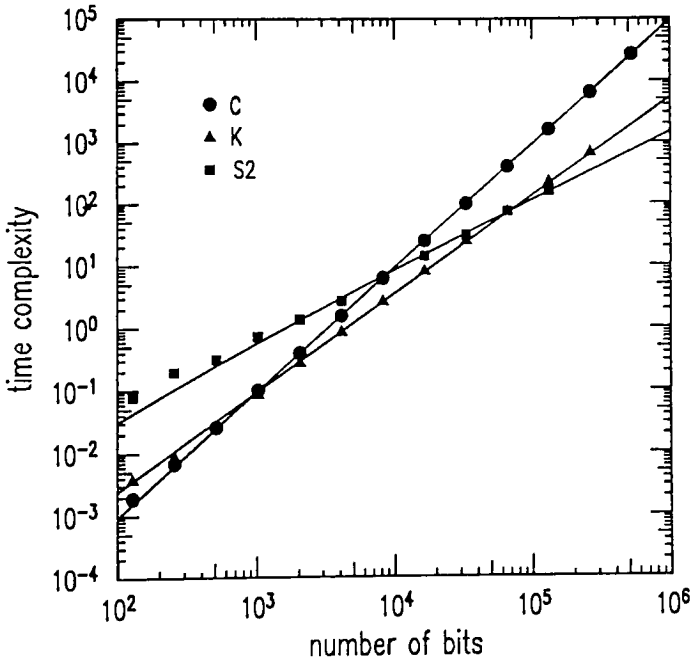


Figure 4.5 shows the graph of the non-modular multiplication for the huge memory model corresponding to the values of table 4.3. The break even

point of the SCHÖNHAGE-STRASSEN algorithm (1982) is significantly increased in comparison to the modular multiplication. This is due to the fact that in contrast to the classical and the KARATSUBA algorithm the SCHÖNHAGE-STRASSEN multiplication algorithm is inherently modular, as mentioned already.

The figures may suggest that the data points form a straight line. Note, that the chosen values of n_2 meet the condition $n_2 = 2^k$ and are therefore suitable n_2's for all of the multiplication algorithms. The distance between two adjacent suitable n_2's is quite different for the individual multiplication algorithms. For example, for the classical multiplication algorithm the distance is always 1, whereas for the SCHÖNHAGE-STRASSEN algorithm (1971) it is $\frac{n_2}{2}$ to the next smaller and n_2 to the next larger suitable number of bits. Therefore, in the case of the convolution algorithms the data points are not connected by straight lines but rather by step-like curves. The step width is smaller for the ordinary convolution algorithm and the SCHÖNHAGE-STRASSEN algorithm (1982) than for the original SCHÖNHAGE-STRASSEN algorithm (1971), which is illustrated in figures 4.6 and 4.7.

The reduction of the step width is one of the important improvements of the modified SCHÖNHAGE-STRASSEN algorithm (1982). In figure 4.6 the time complexity for $n_2 = 2048 \ldots 8192$ is measured in intervals of 256 bits. For the SCHÖNHAGE-STRASSEN algorithm (1971) the corresponding suitable n_2's are 2048, 4096 and 8192. Thus the two steps in the figure run from 2049 through 4096 and 4097 through 8192, respectively. There are more suitable n_2's for the SCHÖNHAGE-STRASSEN algorithm (1982), namely 2048, 2304, 2560, 2816, 3072, 3328, 3584, 3840, 4096, 4608, 5120, 5632, 6144, 6656, 7168, 7680 and 8192. Thus, in the range from 4096 through 8192 we would expect steps of two data points each, but the steps obviously comprise 4 data points. This is true, since parameters like n_B, ω_{n_B} and m are identical for the pairs 4096/4608, 5120/5632, 6144/6656 and 7168/7680, and the only difference, the number l of bits per block, is of no significant influence.

A crucial point in the analysis of the convolution algorithm is the shift operation. By an appropriate choice of the parameters, in particular of ω_{n_B}, and by performing the Fourier transform in Z_m the multiplications by powers of ω_{n_B}, ψ and by n_B^{-1} were substituted with simple shift operations, which are assumend to be fast. Unfortunately, in many computers including the one we used a shift of a binary word by 25 places, for example, takes approx-

Figure 4.6: Detailed shape of the performance curves I (huge memory model).

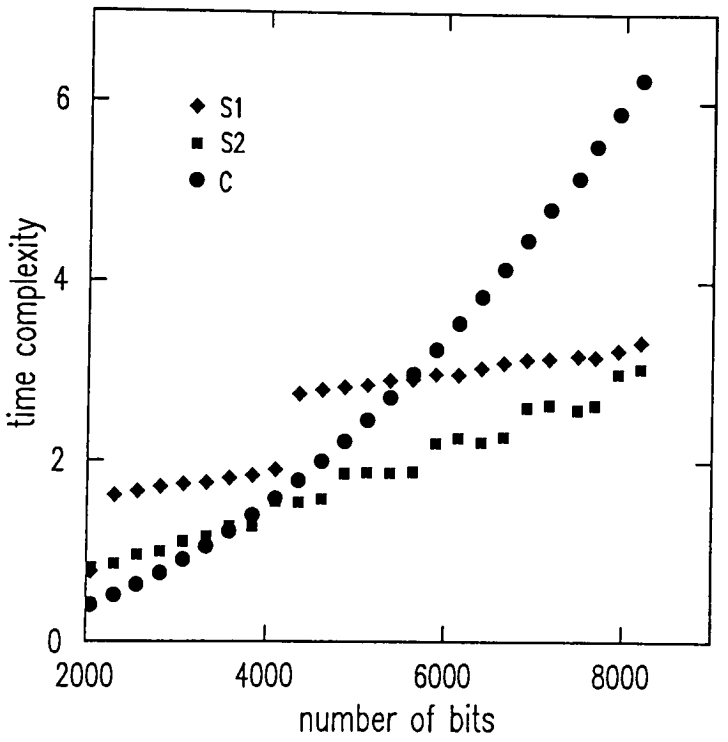
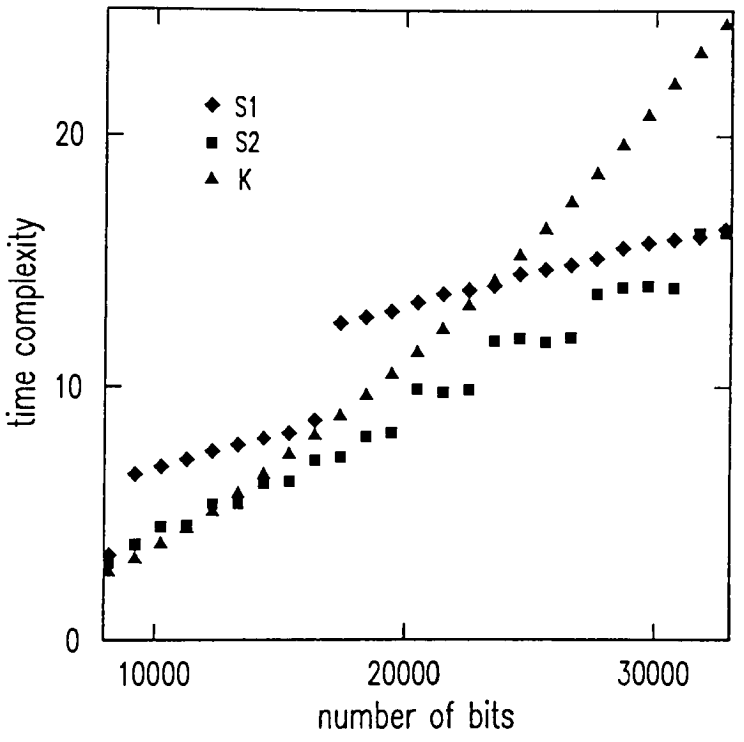


Figure 4.7: Detailed shape of the performance curves II (huge memory model)..



imately the same time as a 32 x 32 bit multiplication. However, there are processors which are equipped with fast multiple-bit shift capabilities. For example, the MC 68020 and MC 68030 are provided with a so called "barrel shift" operation. The comparison of the ordinary convolution algorithm and the SCHÖNHAGE-STRASSEN algorithm (1982) (see tables 4.3 and 4.5), which use both the same set of parameters, already indicates the importance of the shift operations. By employing the standard convolution rather than the negative wrapped convolution the number of shift operations is reduced for the ordinary covolution algorithm, and as a result the execution time decreases.

Table 4.6: Time complexity of the non-modular multiplication for the fast convolution algorithms (O, S1, S2) and the corresponding number of required shift operations (huge memory model).

n_2	O	S1	S2	$shift_O$	$shift_{S1}$	$shift_{S2}$
64	0.0453	0.0626	0.0594	44	68	68
128	0.0606	0.1419	0.0787	44	160	68
256	0.1539	0.1894	0.1928	112	160	160
512	0.2514	0.4554	0.3131	112	368	160
1024	0.605	0.691	0.727	272	368	368
2048	1.167	1.620	1.382	272	832	368
4096	2.349	2.737	2.713	272	832	368
8192	5.30	6.34	6.02	640	1856	832
16384	12.75	12.14	14.14	640	1856	832

Table 4.6 shows an extract of table 4.3 where the corresponding number of required shift operations is added. One can easily see, that the increase of the time complexity is correlated with the increase of the number of shift operations.

Tests for $n_2 = 2048$ suggest that the shift operations take about 40% of the time. The same number of simple 16-bit word shifts by one position would require about 2%, and the availability of barrel shift operations may reduce the time needed to approximately $10-20\%$. The last number is only a rough estimate.

Table 4.7 summarizes the break even points of the multiplication algorithms

Table 4.7: The number of bits for the break even points of the modular and non-modular multiplication for the huge and standard memory model.

Modular Multiplication

	A	B_O	B_{S1}	B_{S2}	C_O	C_{S1}	C_{S2}
huge	512	6656	5632	4096	59392	23552	12800
standard	416	5376	5120	3072	– – –	14848	12288

Non-Modular Multiplication

	A	B_O	B_{S1}	B_{S2}	C_O	C_{S1}	C_{S2}
huge	512	6656	11726	7680	59392	98304	73728
standard	416	5376	7608	6656	– – –	– – –	– – –

which were determined by intensive measurements in the significant data ranges:

- A: break even point classical / KARATSUBA
- B_O: classical / ordinary convolution
- B_{S1}: classical / SCHÖNHAGE-STRASSEN (1971)
- B_{S2}: classical / SCHÖNHAGE-STRASSEN (1982)
- C_O: KARATSUBA / ordinary convolution
- C_{S1}: KARATSUBA / SCHÖNHAGE-STRASSEN (1971)
- C_{S2}: KARATSUBA / SCHÖNHAGE-STRASSEN (1982)

The break even points of the huge memory model are shifted to higher values of n_2 due to its more complicated pointer handling. In addition, the step-like shape of the curves of the convolution algorithms may intensify this effect. For example, C_{S1} is 14848 for the standard and 23552 for the huge memory model. The corresponding steps of the SCHÖNHAGE-STRASSEN algorithm (1982) run from 8193 through 16384 and 16385 through 32768 and are illustrated in figure 4.7.

4.4 Related Studies

During the study of related literature two similar works were found [31] [17]. In [17] the performance of the classical, the KARATSUBA and the SCHÖNHAGE-STRASSEN algorithm (1971) for the non-modular multiplication are examined. The algorithms are implemented in C and tested on a MC 68000. The determined break even points

- A (classical and KARATSUBA) $\approx$ 1400 bits
- B_{S1} (classical and S1) $\approx$ 8000 bits.

are at higher values than the ones we got in our implementation. Furthermore, the break even point between the KARATSUBA and the SCHÖNHAGE-STRASSEN algorithm (1971) is not discovered.

Another interesting approach can be found in [31]. Here the author implemented a multitape Turing machine in assembly code and wrote the multiplication algorithms the classical, KARATSUBA and SCHÖNHAGE-STRASSEN (1982) in the assembly language of the Turing machine. 8 tapes are used to store the data, one has some limited cabability of random access. The finite control is accomplished by the CPU and a separately stored program. In addition to real time units fictitious Turing machine time units were introduced, which are meant "as a fairly realistic approximation to values achievable by a possible hardware implementation of TP" (multitape Turing processor) [31].

An important difference in determining the performance of the algorithm for both time units is the way the shift operation is measured. A single register shift takes one and a double length shift takes 2 fictitious time units, independent of the number of positions the shift is done, whereas the multiplication operation costs 17 fictitious time units, for example. Both a version for a word size of 16 and a word size of 32 are implemented. Employing the fictitious time units the break even points for the modular multiplication are summarized in table 4.8.

It would be interesting to compare these data with the real time data, but the latter are not available to us. The significantly lower break even points of the SCHÖNHAGE-STRASSEN algorithm (1982) is probably due to the different consideration of the shift operations. In section 4.3 the importance of the shift operations was already discussed. Furthermore, the concept of

Table 4.8: Break even points for the modular multiplication for a multitape Turing machine, measured in fictitious time units.

	A	B_{S2}	C_{S2}
16-bit version	224	816	1600
32-bit version	320	?	2400

the tapes in principle embodies a different memory management concept with a static rather than a dynamic memory allocation [5]. In particular the performance of the SCHÖNHAGE-STRASSEN algorithm may profit from this memory concept.

4.5 Conclusions

In this section the detailed presentation and discussion of the performance of the multiplication algorithms throughout chapter 4 are summarized. Corresponding to table 4.7 the ranges for the superior application of the individual multiplication algorithms are:

- modular multiplication:
  - classical: 0 – 500 bits
  - KARATSUBA: 500 – 12 500 bits
  - fast convolution: 12 500 – ... bits
- non-modular multiplication:
  - classical: 0 – 500 bits
  - KARATSUBA: 500 – 60 000 bits
  - fast convolution: 60 000 – ... bits.

The KARATSUBA algorithm stands out for its simplicity and yet efficiency in a quite significant range of data lengths. The fast convolution algorithms are particularly interesting for the modular multiplication of numbers larger

[5] See section 3.2.3 for more details of the memory concept.

than 12500 bits. The modified and polished SCHÖNHAGE-STRASSEN algorithm (1982) is convincing by its improved handyness, and it is especially superior to the original version because of the refinement of the suitable numbers of bits n_2. The ordinary fast convolution algorithm is superior to the SCHÖNHAGE-STRASSEN algorithms for the non-modular multiplication of numbers of "moderate" length, since the negative wrapped convolution is replaced by the conventional convolution.

A critical point for the performance of the fast convolution algorithms is the shift operation. Since many processors don't support fast multiple-position shift operations, the performance is reduced significantly. For that reason the examination of other fast convolution algorithms may be informative. Thereby algorithms analogously to the three prime algorithm (section 2.5.6 and 2.5.7), which employ the Chinese remainder theorem and omit multiple-precision calculations as far as possible, and fast convolution algorithms in the ring of complex numbers (section 2.5.5, [15, p.290]), which have to cope with the precision of the results, suggest themselves.

Acknowledgement

My special thank goes to S. Radziszowski for a motivating and supportive introduction to this topic, and to A. Schönhage for valuable discussions and hints. I'm grateful to Werner for his constant attention to my work and for many constructive comments.

Appendix A

Glossary

Formal symbolism	Meaning	Section introduction
C	classical algorithm	2.3
K	KARATSUBA algorithm	2.4
O	ordinary fast convolution algorithm	2.5.7
S1	SCHÖNHAGE-STRASSEN algorithm (1971)	2.5.8
S2	SCHÖNHAGE-STRASSEN algorithm (1982)	2.5.9
Z	the set of integers	
Z_m	the set of integers modulo m	
C	the set of complex numbers	
R	commutative ring	
F	field	
DFT	discrete Fourier transform	2.5.2
FFT	fast Fourier transform	2.5.4
NTT	number theoretic transform	2.5.5
FT	Fourier transform	2.5.2
FT^{-1}	inverse Fourier transform	2.5.2

Formal symbolism	Meaning	Section introduction
B	radix B	2.2
n	number of digits	2.2
n_2	number of binary digits (bits)	2.2
n_B	number of radix B digits	2.2
l	number of bits per block	2.5.7
m	modulus	2.5.5
ω_n	primitive n-th root of unity	2.5.2
F_t	Fermat number : $F_t = 2^{2^t} + 1$	2.5.5
$(u_{n_B-1}, \ldots, u_1, u_0)_B$	radix-B positional notation: $\sum_{i=0}^{n_B-1} u_i B^i$	2.2
$[u_0, u_1, \ldots, u_{n_B-1}]_B^T$	length n_B column vector	2.2
$u \otimes v$	convolution: $c = u \otimes v$, $c_i = \sum_{j=0}^{n-1} u_j \cdot v_{i-j}$	2.5.3
$x \mid y$	x divides y	2.5.5
$\lceil x \rceil$	the ceiling of x, the smallest integer $\geq x$	
$\lfloor x \rfloor$	the floor of x, the largest integer $\leq x$	
$\gcd(x, y)$	the greatest common divisor of x and y	2.5.5
$\varphi(n)$	EULER's totient function: number of x's, such that $\gcd(x, n) = 1$ for $0 \leq x < n$,	2.5.5
$O(f(n))$	order O of $f(n)$, as the variable $n \to \infty$	2.2

Bibliography

[1] R.C. Agarwal and C.S. Burrus. Number theoretic transforms to implement fast digital convolution. *P IEEE*, 63(4):550–560, 1975.

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

[3] H. Alt. Multiplication is the easiest nontrivial arithmetic function. *Theor Comp*, 36:333–339, 1985.

[4] A. Borodin and I. Munro. *The Computational Complexity of Algebraic and Numeric Problems.* Elsevier, New York, 1975.

[5] J.M. Borwein, P.B. Borwein, and D.H. Bailey. Ramanujan, modular equations, and approximations to pi or how to compute one billion digits of pi. *Am Math Mo*, 96:201–219, 3 1989.

[6] R.P. Brent. Fast multiple-precision evaluation of elementary functions. *J ACM*, 23(2):242–251, 4 1976.

[7] R.P. Brent. A Fortran multiple-precision arithmetic package. *ACM T Math S*, 4(1):57–70, 3 1978.

[8] R.P. Brent and E.M. McMillan. Some new algorithms for high-precision computation of Euler's constant. *Math Comput*, 34:305–312, 1980.

[9] W.L. Briggs, L.B. Hart, R.A. Sweet, and A. O'Gallagher. Multiprocessor FFT methods. *SIAM J Sci Stat Comput*, 8(1):s27–s42, 1 1987.

[10] E.O. Brigham. *The Fast Fourier Transform.* Prentice-Hall, Englewood Cliffs, 1974.

[11] D. V. Chudnovsky and G.V. Chudnovsky. The computation of classical constants. *P NAS US*, 86:8178–8182, 11 1989.

[12] D.E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading, 1982.

[13] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Trans on Inform Th*, IT-22(6):644–654, 11 1976.

[14] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, 1978.

[15] D.E. Knuth. *The Art of Computer Programming. Vol.2 (Seminumerical Algorithms).* Addison-Wesley, Reading, 1981.

[16] D.M. Kodek. Conditions for the existence of fast number theoretic transforms. *IEEE Comput*, 30(5):359–360, 5 1981.

[17] A. Kolb. *Schnelle Multiplikation großer Zahlen Ein Methodenvergleich.* Diploma thesis, Universität Bayreuth, 1989.

[18] E.A. Lamagna. Fast computer algebra. *Computer*, 15:43–56, 9 1982.

[19] J.D. Lipson. *Elements of Algebra and Algebraic Computing.* Addison-Wesley, Reading, 1981.

[20] K. Mehlhorn and F.P. Preparata. Area-time optimal VLSI integer multiplier with minimum computation time. *Inf Contr*, 58:137–156, 1983.

[21] G.L. Miller. Riemann's hypothesis and tests for primality. *J Comput Sy*, 13:300–317, 1976.

[22] L. Monier. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theor sComp*, 12:97–108, 1980.

[23] H.J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer-Verlag, Berlin, 1981.

[24] J. Pintz, W.L. Steiger, and E. Szemerédi. Infinite sets of primes with fast primality tests and quick generation of large primes. *Math Comput*, 53(187):399–406, 7 1989.

[25] J. M. Pollard. The fast Fourier transform in a finite field. *Math of Comp*, 25(114):365–374, 4 1971.

[26] R.L Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm ACM*, 21(2):120–126, 2 1978.

[27] S. Salamin. Computation of pi using arithmetic-geometric mean. *Math Comput*, 30(135):565–570, 7 1976.

[28] A. Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. *LNCS (Lecture Notes in CS)*, 3–15, 1982.

[29] A. Schönhage. *private communication.* 1990.

[30] A. Schönhage. Storage modification machines. *SIAM J Comp*, 9:490–508, 1980.

[31] A. Schönhage. Tapes versus pointers, a study in implementing fast algorithms. ICALP, Rennes, 1986.

[32] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[33] B. Serpette, J. Vuillemin, and J. Hervè. BigNum: A portable and efficient package for arbitrary-precision arithmetic. 5 1989. Digital PRL, 85 Avenue Victor Hugo, 92563 Rueil Malmaison Cedex France (librarian@decprl.dec.com).

[34] W.T. Vetterling et al. *Numerical Recipes Example Book.* Cambridge University Press, Cambridge, 1986.