A Rigorous Extension of the Schönhage-Strassen Integer Multiplication Algorithm Using Complex Interval Arithmetic*

Raazesh Sainudiin
Laboratory for Mathematical Statistical Experiments
& Department of Mathematics and Statistics
University of Canterbury, Private Bag 4800
Christchurch 8041, New Zealand
r.sainudiin@math.canterbury.ac.nz

Thomas Steinke
Harvard School of Engineering and Applied Sciences
Maxwell Dworkin, Room 138, 33 Oxford Street
Cambridge, MA 02138, USA
tsteinke@seas.harvard.edu

Abstract

Multiplication of n-digit integers by long multiplication requires $O(n^2)$ operations and can be time-consuming. A. Schönhage and V. Strassen published an algorithm in 1970 that is capable of performing the task with only $O(n \log(n))$ arithmetic operations over $\mathbb{C}$; naturally, finite-precision approximations to $\mathbb{C}$ are used and rounding errors need to be accounted for. Overall, using variable-precision fixed-point numbers, this results in an $O(n(\log(n))^{2+\varepsilon})$-time algorithm. However, to make this algorithm more efficient and practical we need to make use of hardware-based floating-point numbers. How do we deal with rounding errors? and how do we determine the limits of the fixed-precision hardware? Our solution is to use interval arithmetic to guarantee the correctness of results and determine the hardware's limits. We examine the feasibility of this approach and are able to report that 75,000-digit base-256 integers can be handled using double-precision containment sets. This demonstrates that our approach has practical potential; however, at this stage, our implementation does not yet compete with commercial ones, but we are able to demonstrate the feasibility of this technique.

Keywords: integer multiplication, complex rectangular interval arithmetic, first Schönhage-Strassen algorithm
AMS subject classifications: 65G20,65Y04,65T50,11Y35

*Submitted: February 13, 2013; Revised: August 18, 2013 and September 5, 2013; Accepted: September 13, 2013.

1 Introduction

Multiplication of very large integers is a crucial subroutine of many algorithms such as the RSA cryptosystem [10]. We are interested in the problem of mulipyling integers with up to 75,000 digits or 600,000 bits (beyond the 1024 to 4096 bit range used in RSA). Consequently, much effort has gone into finding fast and reliable multiplication algorithms; [6] discusses several methods. The asymptotically-fastest known algorithm [3] requires $n \log(n) 2^{O(\log^*(n))}$ steps, where $\log^*$ is the iterated logarithm — defined as the number of times one has to repeatedly take the logarithm before the number is less than 1. However, Fuerer's algorithm is only suitable in practice for specialized applications dealing with astronomically large integers. We shall concern ourselves with the practicalities of the subject; we will demonstrate our algorithm's feasibility on a finite range of numbers with 75,000 or fewer digits using double-precision containment sets. In this study we are unable to address the second-order question of comparing the computational efficiency of a proof-of-concept implementation of our algorithm (Section 5) with other optimized integer multiplicaiton algorithms in current use.

The algorithm we are studying here is based on the first of two asymptotically fast multiplication algorithms by A. Schönhage and V. Strassen [11]. These algorithms are based on the convolution theorem and the fast Fourier transform. The first algorithm (the one we are studying) performs the discrete Fourier transform over $\mathbb{C}$ using finite-precision approximations. The second algorithm uses the same ideas as the first, but it works over the finite ring $\mathbb{Z}_{2^{2^n}+1}$ rather than the uncountable field $\mathbb{C}$. We wish to point out that "*the* Schönhage-Strassen algorithm" usually refers to the second algorithm. However, in this document we use it to refer to the first $\mathbb{C}$-based algorithm.

From the theoretical viewpoint, the second algorithm is much nicer than the first. The second algorithm does not require the use of finite-precision approximations to $\mathbb{C}$. Also, the second algorithm requires $O(n \log(n) \log(\log(n)))$ steps to multiply two n-bit numbers, making it asymptotically-faster than the first algorithm. However, the second algorithm is much more complicated than the first, and it is outperformed by asymptotically-slower algorithms, such as long multiplication, for small-to-medium input sizes.

The first Schönhage-Strassen Algorithm is more elegant, if the finite-precision approximations are ignored. More importantly, it is faster in practice. Previous studies [4] have shown that the first algorithm can be faster than even highly-optimised implementations of the second. However, the first algorithm's reliance on finite-precision approximations, despite exact answers being required, leads to it being discounted.

The saving grace of the Schönhage-Strassen algorithm is that at the end of the computation an integral result will be obtained. So the finite-precision approximations are rounded to integers. Thus, as long as rounding errors are sufficiently small for the rounding to be correct, an exact answer will be obtained. Schönhage and Strassen showed that fixed-point numbers with a variable precision of $O(\log(n))$ bits would be sufficient to achieve this.

For the Schönhage-Strassen algorithm to be practical over the range of integers under consideration, we need to make use of hardware-based floating-point numbers; software-based variable-precision numbers are simply too slow. However, we need to be able to manage the rounding errors. At the very least, we must be able to detect when the error is too large and more precision is needed. The usual approach to this is to prove some kind of worst-case error bound (for an example, see [9]). Then we can be sure that, for sufficiently small inputs, the algorithm will give correct results. However, worst-case bounds are rarely tight. We propose the use of dynamic error

bounds using existing techniques from computer-aided proofs.

Dynamic error detection allows us to move beyond worst-case bounds. For example, using standard single-precision floating-point numbers, our naïve implementation of the Schönhage-Strassen algorithm sometimes gave an incorrect result when we tried multiplying two 120-digit base-256 numbers, but it usually gave correct results. Note that by a 'naïve implementation' we simply mean a modificiation of the Schönhage-Strassen algorithm that uses fixed-precison floating-point arithmetic and does not guarantee correctness. A worst-case bound would not allow us to use the algorithm in this case, despite it usually being correct. Dynamic error detection, however, would allow us to try the algorithm, and, in the rare instances where errors occur, it would inform us that we need to use more precision.

Interval arithmetic has been used to guarantee a correct solution set in problems defined by integers (see for e.g. [2, 8, 1]). We will use complex interval containment sets for all complex arithmetic operations. This means that at the end of the computation, where we would ordinarily round to the nearest integer, we simply choose the unique integer in the containment set. If the containment set contains multiple integers, then we report an error. This rigorous extension of the Schönhage-Strassen algorithm therefore constitutes a computer-aided proof of the desired product. When an error is detected, we must increase the precision being used or we must use a different algorithm.

For those unfamiliar with the Schönhage-Strassen algorithm or with interval arithmetic, we describe these in section 2. Then, in section 3, we show the empirical results of our study using a highly non-optimized proof-of-concept implementation. Section 4, our conclusion, briefly discusses the implications of our results.

2 The Algorithm

For the sake of completeness we explain the Schönhage-Strassen algorithm, as it is presented in [11]. We also explain how we have modified the algorithm using interval arithmetic in subsection 2.4. Those already familiar with the material may skip all or part of this section.

We make the convention that a positive integer x is represented in base b (usually $b = 2^k$ for some $k \in \mathbb{N}$) as a vector $x \in \mathbb{Z}_b^n := \{0, 1, 2, \ldots, b-1\}^n$; the value of x is

$$x = \sum_{i=0}^{n-1} x_i b^i.$$

2.1 Basic Multiplication Algorithm

The above definition immediately leads to a formula for multiplication. Let x and y be positive integers with representations $x \in \mathbb{Z}_b^n$ and $y \in \mathbb{Z}_b^m$. Then

$$xy = \left(\sum_{i=0}^{n-1} x_i b^i\right)\left(\sum_{j=0}^{m-1} y_j b^j\right) = \sum_{i=0}^{n+m-2} \sum_{j=\max\{0,i-m+1\}}^{\min\{n-1,i\}} x_j y_{i-j} b^i = \sum_{i=0}^{n+m-1} z_i b^i.$$

We cannot simply set $z_i = \sum_{j=\max\{0,i-m+1\}}^{\min\{n-1,i\}} x_j y_{i-j}$; this may violate the constraint that $0 \le z_i \le b-1$ for every i. We must 'carry' the 'overflow'. This leads to the long multiplication algorithm (see [6]).

The Long Multiplication Algorithm

```
1.   Input:  x ∈ Z_b^n and y ∈ Z_b^m
2.   Output:  z ∈ Z_b^{n+m} # z = xy
3.   Set c = 0.  # c = carry
4.   For i = 0 up to n + m − 1 do {
5.        Set s = 0.  # s = sum
6.        For j = max{0, i − m + 1} up to min{n − 1, i} do {
7.             Set s = s + x_j y_{i−j}.
8.        }.
9.        Set z_i = (s + c) mod b.
10.       Set c = ⌊(s + c)/b⌋.
11.  }.
12.  # c = 0 at the end.
```

This algorithm requires $O(mn)$ steps (for a fixed b). Close inspection of the long multiplication algorithm might suggest that $O(mn \log(\min\{m, n\}))$ steps are required as the sum s can become very large. However, roughly speaking, adding a uniformly distributed bounded number ($x_j y_{i-1} < b^2$) to an unbounded number (s) is, *on average*, a constant-time operation.

2.2 Preliminaries

The basis of the Schönhage-Strassen algorithm is the discrete Fourier transform and the convolution theorem. Readers familiar with this material may skip the following two sections.

2.2.1 The Discrete Fourier Transform

The discrete Fourier transform is a map from $\mathbb{C}^n$ to $\mathbb{C}^n$. In this section we will define the discrete Fourier transform and we will show how it and its inverse can be calculated with $O(n \log(n))$ complex additions and multiplications. See [6] for further details.

Definition 2.1 (Discrete Fourier Transform)**.** *Let $x \in \mathbb{C}^n$ and let $\omega := e^{\frac{2\pi \mathrm{i}}{n}}$. Then define the discrete Fourier transform $\hat{x} \in \mathbb{C}^n$ of x by*

$$\hat{x}_i := \sum_{j=0}^{n-1} x_j \omega^{ij} \quad (0 \leq i \leq n-1).$$

There is nothing special about our choice of ω; the consequences of the following lemma are all that we need ω to satisfy. Any other element of $\mathbb{C}$ with the same properties would suffice.

Lemma 2.2. *Let $n > 1$ and $\omega = e^{\frac{2\pi \mathrm{i}}{n}}$. Then*

$$\omega^n = 1 \text{ and } \omega^k \neq 1 \text{ for all } 0 < k < n$$

and, for all $0 < k < n$,

$$\sum_{i=0}^{n-1} \omega^{ik} = 0.$$

Note that the case where $n = 1$ is uninteresting, as $\omega = 1$ and the discrete Fourier transform is the identity mapping $\hat{x} = x$.

Proof. Firstly,

$$\omega^n = \left(e^{\frac{2\pi \mathrm{i}}{n}}\right)^n = e^{2\pi \mathrm{i}} = 1.$$

We know that $e^\theta = 1$ if and only if $\theta = 2\pi \mathrm{i} m$ for some $m \in \mathbb{Z}$. Thus, if $\omega^k = 1$, then k must be a multiple of n, which eliminates the possibility that $0 < k < n$.

Fix k with $0 < k < n$ and let $s_k := \sum_{i=0}^{n-1} \omega^{ik}$. Then

$$\begin{aligned} \omega^k s_k &= \sum_{i=0}^{n-1} \omega^{(i+1)k} = \sum_{i=1}^{n} \omega^{ik} = \sum_{i=1}^{n-1} \omega^{ik} + \omega^{kn} = \sum_{i=1}^{n-1} \omega^{ik} + 1 = \sum_{i=1}^{n-1} \omega^{ik} + \omega^{0k} \\ &= \sum_{i=0}^{n-1} \omega^{ik} = s_k. \end{aligned}$$

So $\omega^k s_k = s_k$. If $s_k \neq 0$, then we can divide by s_k to get $\omega^k = 1$, which is impossible. So $s_k = 0$. □

Now we can prove that the discrete Fourier transform is a bijection.

Proposition 2.3 (Inverse Discrete Fourier Transform)**.** *Let $x \in \mathbb{C}^n$ and let $\omega = e^{\frac{2\pi \mathrm{i}}{n}}$. Define $\check{x} \in \mathbb{C}^n$ by*

$$\check{x}_i := \frac{1}{n} \sum_{j=0}^{n-1} x_j \omega^{-ij} \quad (0 \leq i \leq n-1).$$

Then this defines the inverse of the discrete Fourier transform — that is, if $y = \hat{x}$, then $\check{y} = x$.

Proof. Fix $x \in \mathbb{C}^n$, let $y = \hat{x}$ and let $z = \check{y}$. We wish to show that $z = x$. If $n = 1$, then this is trivial, as $x = y = z$, so we may assume that $n > 1$. First of all, it follows from Lemma 2.2 that, if $l \in \mathbb{Z}$ and n does not divide l, then

$$\sum_{i=0}^{n-1} \omega^{il} = 0.$$

If, on the other hand, n divides l, then

$$\sum_{i=0}^{n-1} \omega^{il} = n.$$

Now, fixing i with $0 \leq i \leq n-1$, we have

$$\begin{aligned} z_i &= \frac{1}{n} \sum_{j=0}^{n-1} y_j \omega^{-ij} = \frac{1}{n} \sum_{j=0}^{n-1} \left(\sum_{k=0}^{n-1} x_k \omega^{jk} \right) \omega^{-ij} \\ &= \frac{1}{n} \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} x_k \omega^{jk} \omega^{-ij} = \frac{1}{n} \sum_{k=0}^{n-1} x_k \sum_{j=0}^{n-1} \omega^{j(k-i)} \\ &= \frac{1}{n} \sum_{k=0}^{n-1} x_k \left\{ \begin{array}{ll} n, & \text{if } n \text{ divides } k-i \\ 0, & \text{otherwise} \end{array} \right\} = \sum_{k=0}^{n-1} x_k \left\{ \begin{array}{ll} 1, & \text{if } k-i=0 \\ 0, & \text{otherwise} \end{array} \right\} \\ &= x_i. \end{aligned}$$

□

Now we explain the fast Fourier transform; this is simply a fast algorithm for computing the discrete Fourier transform and its inverse.

Let n be a power of 2 and $x \in \mathbb{C}^n$ be given. Now define $x_{\text{even}}, x_{\text{odd}} \in \mathbb{C}^{n/2}$ by

$$(x_{\text{even}})_i = x_{2i}, \qquad (x_{\text{odd}})_i = x_{2i+1},$$

for all i with $0 \leq i \leq n/2 - 1$.

Now the critical observation of the Cooley-Tukey fast Fourier transform algorithm is the following. Fix i with $0 \leq i \leq n-1$ and let $\omega = e^{\frac{2\pi i}{n}}$. Then we have

$$\begin{aligned}
\hat{x}_i &= \sum_{j=0}^{n-1} x_j \omega^{ij} \\
&= \sum_{j=0}^{n/2-1} x_{2j} \omega^{2ij} + \sum_{j=0}^{n/2-1} x_{2j+1} \omega^{2ij+i} \\
&= \sum_{j=0}^{n/2-1} (x_{\text{even}})_j \left(\omega^2\right)^{ij} + \omega^i \sum_{j=0}^{n/2-1} (x_{\text{odd}})_j \left(\omega^2\right)^{ij} \\
&= \sum_{j=0}^{n/2-1} (x_{\text{even}})_j \left(\omega^2\right)^{(i \bmod n/2)j} + \omega^i \sum_{j=0}^{n/2-1} (x_{\text{odd}})_j \left(\omega^2\right)^{(i \bmod n/2)j} \\
&= (\hat{x}_{\text{even}})_{i \bmod n/2} + \omega^i (\hat{x}_{\text{odd}})_{i \bmod n/2}.
\end{aligned}$$

Note that $\left(\omega^2\right)^{n/2} = 1$, so taking the modulus is justified. This observation leads to the following divide-and-conquer algorithm.

The Cooley-Tukey Fast Fourier Transform

```
Input:  n = 2^k and x ∈ C^n
Output:  x̂ ∈ C^n
function FFT(k, x) {
     If k = 0, then x̂ = x.
     Partition x into x_even, x_odd ∈ C^{n/2}.
     Compute x̂_even = FFT(k − 1, x_even) by recursion.
     Compute x̂_odd = FFT(k − 1, x_odd) by recursion.
     Compute ω = e^{2πi/n}.
     For i = 0 up to n − 1 do {
          Set x̂_i = (x̂_even)_{i mod n/2} + ω^i (x̂_odd)_{i mod n/2}.
     }.
}.
```

It is easy to show that this algorithm requires $O(n \log(n))$ complex additions and multiplications. With very little modification we are also able to obtain a fast algorithm for computing the inverse discrete Fourier transform.

Note that, to compute ω, we can use the recurrence

$$\omega_1 = 1, \quad \omega_2 = -1, \quad \omega_4 = \mathrm{i}, \quad \omega_{2n} = \frac{1+\omega_n}{|1+\omega_n|} \quad (n \geq 3),$$

where $\omega_n = e^{\frac{2\pi \mathrm{i}}{n}}$. Other efficient methods of computing ω are also available.

2.2.2 The Convolution Theorem

We start by defining the convolution. Let $a, b \in \mathbb{C}^n$. We can interpret a and b as the coefficients of two polynomials — that is,

$$f_a(z) = a_0 + a_1 z + \cdots + a_{n-1} z^{n-1} \quad \text{and} \quad f_b(z) = b_0 + b_1 z + \cdots + b_{n-1} z^{n-1}.$$

The convolution of a and b — denoted by $a * b$ — is, for our purposes, the vector of coefficients obtained by multiplying the polynomials f_a and f_b. Thus we have $f_{a*b}(z) = f_a(z) f_b(z)$ for all $z \in \mathbb{C}$. Note that we can add 'padding zeroes' to the end of the coefficient vectors without changing the corresponding polynomial.

The convolution theorem relates convolutions to Fourier transforms. We only use a restricted form.

Theorem 2.4 (Convolution Theorem)**.** *Let $a, b \in \mathbb{C}^n$ and $c := a * b \in \mathbb{C}^m$, where $m = 2n - 1$. Pad a and b by setting*

$$a' = (a_0, a_1, \cdots, a_{n-1}, 0, \cdots, 0), b' = (b_0, b_1, \cdots, b_{n-1}, 0, \cdots, 0) \in \mathbb{C}^m.$$

Then, for every i with $0 \leq i \leq m-1$,

$$\hat{c}_i = \hat{a'}_i \hat{b'}_i.$$

Proof. By definition, if $\omega = e^{\frac{2\pi i}{m}}$ and $0 \leq i \leq m-1$, then

$$\hat{c}_i = f_c(\omega^i) = f_a(\omega^i) f_b(\omega^i) = f_{a'}(\omega^i) f_{b'}(\omega^i) = \hat{a'}_i \hat{b'}_i.$$

□

The convolution theorem gives us a fast method of computing convolutions and, thus, of multiplying polynomials. Given $a, b \in \mathbb{C}^n$, we can calculate $c = a * b$ using only $O(n \log(n))$ arithmetic operations as follows.

- Let $k = \lceil \log_2(2n-1) \rceil$. (We need a sufficiently large power of two for the fast Fourier transform algorithm to work.)
- First we pad a and b to get a' and b' in $\mathbb{C}^{2^k}$.
- We calculate $\hat{a'}$ and $\hat{b'}$ using the fast Fourier transform.
- We calculate $\hat{c}$ using the convolution theorem — that is, $\hat{c}_i = \hat{a'}_i \hat{b'}_i$ $(0 \leq i \leq 2n-2)$.
- We calculate c from $\hat{c}$ using the inverse fast Fourier transform.

The Fast Convolution Algorithm

1. `Input:` $a, b \in \mathbb{C}^n$
2. `Output:` $c = a * b \in \mathbb{C}^m$
3. `Set` $k = \lceil \log_2(2n-1) \rceil$ `and` $m = 2^k$.
4. `# Pad` a `and` b `so they are in` $\mathbb{C}^m$.
5. `Set` $a' = (a_0, a_1, \cdots, a_{n-1}, 0, \cdots, 0) \in \mathbb{C}^m$.
5. `Set` $b' = (b_0, b_1, \cdots, b_{n-1}, 0, \cdots, 0) \in \mathbb{C}^m$.
6. `Compute` $\hat{a'} = \mathtt{FFT}(k, a')$ `and` $\hat{b'} = \mathtt{FFT}(k, b')$.
7. `For` $0 \leq i \leq m-1$, `set` $\hat{c}_i = \hat{a'}_i \hat{b'}_i$.
8. `Compute` $c = \mathtt{FFT}^{-1}(k, \hat{c})$.

2.3 The Schönhage-Strassen Algorithm

The Schönhage-Strassen algorithm multiplies two integers by convolving them and then performing carrys. Let two base-b integer representations be x and y. We consider the digits as the coefficients of two polynomials. Then $x = f_x(b)$, $y = f_y(b)$ and

$$xy = f_x(b) f_y(b) = f_{x*y}(b).$$

So, to compute xy, we can first compute $x * y$ in $O(n \log(n))$ steps and then we can evaluate $f_{x*y}(b)$. The evaluation of $f_{x*y}(b)$ to yield an integer representation z is simply the process of performing carrys.

The Schönhage-Strassen Algorithm

1. `Input:` $x \in \mathbb{Z}_b^n$ `and` $y \in \mathbb{Z}_b^n$
2. `Output:` $z \in \mathbb{Z}_b^{2n}$ `#` $z = xy$
3. `Compute` $x * y$ `using the fast convolution Algorithm.`
4. `Set` $c = 0$. `#carry`
5. `For` $i = 0$ `up to` $2n-2$ `do` $\{$
6. `Set` $z_i = \left((x * y)_i + c \right) \mod b$.
7. `Set` $c = \lfloor \left((x * y)_i + c \right) / b \rfloor$.
8. $\}$.
9. `Set` $z_{2n-1} = c$.

Thus the Schönhage-Strassen algorithm performs the multiplication using $O(n \log(n))$ complex arithmetic operations.

When finite-precision complex arithmetic is done, rounding errors are introduced. However, this can be countered: We know that $x * y$ must be a vector of integers. As long as the rounding errors introduced are sufficiently small, we can round to the nearest integer and obtain the correct result. Schönhage and Strassen [11] proved that $O(\log(bn))$-bit floating point numbers give sufficient precision.

2.4 Interval Arithmetic

Our rigorous extension of the algorithm uses containment sets. By replacing all complex numbers with complex containment sets, we can modify the Schönhage-Strassen algorithm to find a containment set of $x * y$; if the containment set only contains one integer-valued vector, then we can be certain that this is the correct value. We have used rectangular containment sets of machine-representable floating-point intervals with directed rounding to guarantee the desired integer product. A brief overview of the needed interval analysis [7] is given next.

Let $\underline{x}, \overline{x}$ be real numbers with $\underline{x} \leq \overline{x}$. Let $[\underline{x}, \overline{x}] = \{x \in \mathbb{R} : \underline{x} \leq x \leq \overline{x}\}$ be a closed and bounded real interval and let the set of all such intervals be $\mathbb{IR} = \{[\underline{x}, \overline{x}] : \underline{x} \leq \overline{x}\,;\, \underline{x}, \overline{x} \in \mathbb{R}\}$. Note that $\mathbb{R} \subset \mathbb{IR}$ since we allow thin or point intervals with $\underline{x} = \overline{x}$. If $\star$ is one of the arithmetic operators $+$, $-$, $\cdot$, $/$, we define arithmetic over operands in $\mathbb{IR}$ by $[\underline{a}, \overline{a}] \star [\underline{b}, \overline{b}] := \{a \star b : a \in [\underline{a}, \overline{a}], b \in [\underline{b}, \overline{b}]\}$, with the exception that $[\underline{a}, \overline{a}]/[\underline{b}, \overline{b}]$ is undefined if $0 \in [\underline{b}, \overline{b}]$. Due to continuity and monotonicity of the operations and compactness of the operands, arithmetic over $\mathbb{IR}$ is given by real arithmetic operations with the bounds:

$$
\begin{aligned}
[\underline{a}, \overline{a}] + [\underline{b}, \overline{b}] &= [\underline{a} + \underline{b}, \overline{a} + \overline{b}] \\
[\underline{a}, \overline{a}] - [\underline{b}, \overline{b}] &= [\underline{a} - \overline{b}, \overline{a} + \underline{b}] \\
[\underline{a}, \overline{a}] \cdot [\underline{b}, \overline{b}] &= [\min\{\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}\}, \max\{\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}\}] \\
[\underline{a}, \overline{a}]/[\underline{b}, \overline{b}] &= [\underline{a}, \overline{a}] \cdot [1/\overline{b}, 1/\underline{b}], \text{ if } 0 \notin [\underline{b}, \overline{b}] \ .
\end{aligned}
$$

In addition to the above elementary operations over elements in $\mathbb{IR}$, our algorithm requires us to contain the range of the square root function over elements in $\mathbb{IR} \cap [0, \infty)$. Once again, due to the monotonicity of the square root function over non-negative reals it suffices to work with the real image of the bounds $\sqrt{[\underline{x}, \overline{x}]} = [\sqrt{\underline{x}}, \sqrt{\overline{x}}]$, if $0 \leq \underline{x}$. To complete the requirements for our rigorous extension of the Schönhage-Strassen algorithm we need to extend addition, multiplication and division by a non-zero integer to elements in

$$
\mathbb{IC} := \left\{ [\underline{z}, \overline{z}] := [\underline{z_1}, \overline{z_1}] + \mathrm{i}[\underline{z_2}, \overline{z_2}] : [\underline{z_1}, \overline{z_1}], [\underline{z_2}, \overline{z_2}] \in \mathbb{IR} \right\}.
$$

Interval arithmetic over $\mathbb{IR}$ naturally extends to $\mathbb{IC}$, the set of rectangular complex intervals. Addition and subtraction over $[\underline{z}, \overline{z}], [\underline{w}, \overline{w}] \in \mathbb{IC}$ given by

$$
[\underline{z}, \overline{z}] \pm [\underline{w}, \overline{w}] = ([\underline{z_1}, \overline{z_1}] \pm [\underline{w_1}, \overline{w_1}]) + \mathrm{i}([\underline{z_2}, \overline{z_2}] \pm [\underline{w_2}, \overline{w_2}])
$$

are sharp (i.e. $[\underline{z}, \overline{z}] \pm [\underline{w}, \overline{w}] = \{z \pm w : z \in [\underline{z}, \overline{z}], w \in [\underline{w}, \overline{w}]\}$) but not multiplication or division. Complex interval multiplication and division of a complex interval by a non-negative integer can be contained with real interval multiplications given by

$$
[\underline{z}, \overline{z}] \cdot [\underline{w}, \overline{w}] = ([\underline{z_1}, \overline{z_1}] \cdot [\underline{w_1}, \overline{w_1}] - [\underline{z_2}, \overline{z_2}] \cdot [\underline{w_2}, \overline{w_2}]) + \mathrm{i}([\underline{z_1}, \overline{z_1}] \cdot [\underline{w_2}, \overline{w_2}] + [\underline{z_2}, \overline{z_2}] \cdot [\underline{w_1}, \overline{w_1}]).
$$

See [5] for details about how `C-XSC` manipulates rectangular containment sets over $\mathbb{IR}$ and $\mathbb{IC}$.

3 Proof of Concept Demonstration

We have implemented the Schönhage-Strassen algorithm, our containment-set version with rectangular complex intervals and long multiplication in `C++` using the `C-XSC`

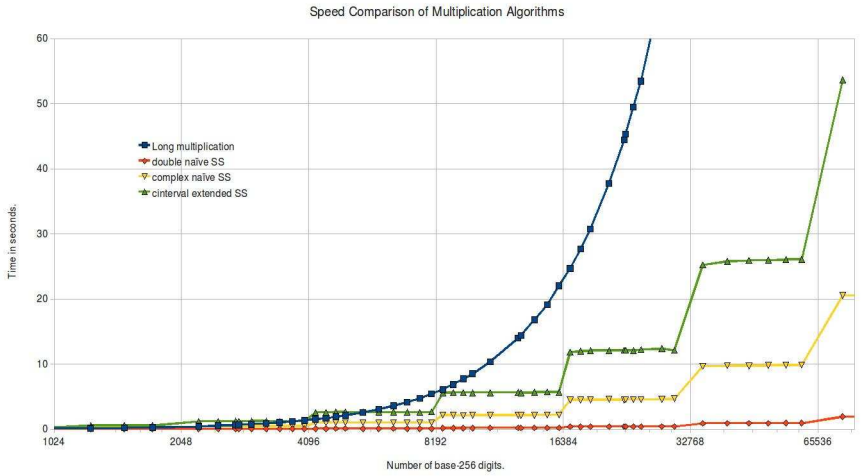

Figure 1: CPU timing comparisons between three implementations of the first Schönhage-Strassen algorithm and the long multiplication algorithm.

library [5]. Our implementation is available at
`http://www.math.canterbury.ac.nz/~r.sainudiin/codes/capa/multiply/` [1].
Results show that, using base 256, our version of the algorithm is usually able to guarantee correct answers for up to 75,000-digit numbers.

The following graph compares the speed of long multiplication (labelled 'Long multiplication'), the conventional Schönhage-Strassen algorithm with different underlying data types (the implementation using the `C-XSC complex` data type is the line labelled 'complex naïve SS' and the one using our own implementation of complex numbers based on the `C++ double` data type is labelled 'double naïve SS') and our containment-set version ('cinterval extended SS') on uniformly-random n-digit base-256 inputs. All tests were performed on a 2.2 GHz 64-bit AMD Athlon 3500+ Processor running Ubuntu 9.04 using `C-XSC` version 2.2.4 and gcc version 4.3.1. Times were recorded using the `C++ clock()` function — that is to say, CPU time was recorded. Note that only the 'Long multiplication' and 'cinterval extended SS' implementations are guaranteed to produce correct results. The 'double naïve SS' and 'complex naïve SS' implementations may have produced erroneous results, as the implementations do not necessarily provide sufficient precision; these are still shown for comparison. Note also that by 'naïve' we mean that these implementations use fixed-precision floating-point arithmetic, whereas the 'real' Schöhnage-Strassen algorithm uses variable-precision, which is much slower.

Figure 1 shows that the Schönhage-Strassen algorithm is much more efficient than long multiplication for large inputs. However, our modified version of the algorithm is slower than the naïve Schönhage-Strassen algorithm. We believe that `C-XSC` is not well-optimised; for example, their point `complex` data type (used in the 'complex naïve SS' implementation) is much slower than our `double`-based complex data type (used in the 'double naïve SS' implementation), even though ostensibly they are the

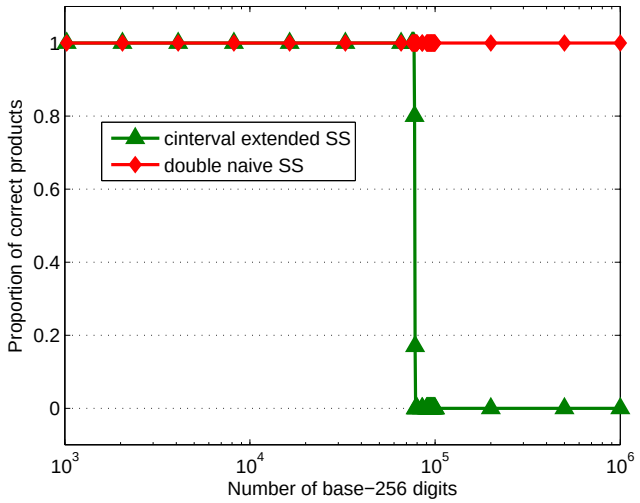

Figure 2: Proportion of correct products returned by the algorithms.

same thing. We see that the `C-XSC cinterval` type (used in the 'cinterval extended SS' implementation) is about three times as slow as the `complex` type. This leaves the possibility that a more optimised implementation of containment sets would be able to compete with commercial algorithms. Investigations such as [4] have shown that the Naïve Schönhage-Strassen algorithm is able to compete with commercial implementations.

Note that the "steps" seen in the graph can be explained by the fact that the algorithm will always round the size up to the nearest power of two. Thus there are steps at the powers of two. The most important feature of our results is the range of input sizes for which our algorithm successfully determines the answer. Using only standard double-precision IEEE floating-point numbers, we are able to use the algorithm to multiply 75,000-digit, or 600,000-bit, integers; this range is more than sufficient for most applications, and at this point the second Schönhage-Strassen algorithm will become competitive.

Figure 2 shows the proportion of correct products returned by 'double naïve SS' and 'cinterval extended SS' algorithms. The correctness was verified against the output from the long multiplication algorithm for 100 pairs of random input integers with up to 10^6 base-256 digits. Both 'complex naïve SS' and 'double naïve SS' returned the same correct answer in our experiments. Unlike 'double naïve SS' or 'complex naïve SS', if 'cinterval extended SS' algorithm fails (between 75000 and 10^6 base-256 digits) due to its enclosure containing more than one integer then we know that the result is incorrect and can take appropriate measures to increase precision of 'cinterval extended SS' or resort to the second SS algorithm. On the other hand we have no guarantees that 'double naïve SS' and 'cinterval extended SS' algorithms have indeed produced the correct result without further confirmation by a verified algorithm. This makes 'cinterval extended SS' algorithm auto-validating.

4 Conclusion

Our investigation has demonstrated that the Schönhage-Strassen algorithm with containment sets is a practical algorithm that could be used reliably for applications requiring the multiplication of large integers. However, as our implementation was not optimised, this is more of a feasibility study than a finished product.

Note that the advantage of our algorithm over the original Schönhage-Strassen algorithm is that we make use of hardware-based floating-point arithmetic, whereas the original is designed to make use of much slower software-based arithmetic. Both the original algorithm and our adaptation always produce correct results. However, we use a different approach to guaranteeing correctness. The naïve algorithms we mention are not guaranteed to be correct because they are modifications of the Schönhage-Strassen algorithm which does not take measures to ensure correctness — they simply use fixed-precision floating-point arithemtic and hope for the best; these are only useful for speed comparisons.

It remains to optimise our implementation of the algorithm to compete with commercial libraries. This is dependent on a faster implementation of interval arithmetic. It may also be interesting to use circular containment sets rather than rectangular containment sets. The advantage of circular containment sets is that they easily deal with complex rotations — that is, multiplying by $e^{\mathrm{i}\theta}$; this is in fact the only type of complex multiplication (other than division by an integer) that our algorithm performs.

Acknowledgements

We thank an anonymous referee and the editor for their valuable comments.

References

[1] Krzysztof R. Apt and Peter Zoeteweij. An analysis of arithmetic constraints on integer intervals. *Constraints*, 12(4):429–468, December 2007.

[2] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.

[3] Martin Fürer. Faster integer multiplication. In *39th ACM STOC*, pages 57–66, San Diego, California, USA, June 2007.

[4] Pierrick Gaudry, Alexander Kruppa, and Paul Zimmermann. A gmp-based implementation of schönhage-strassen's large integer multiplication algorithm. In *ISSAC '07: Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, pages 167–174, New York, NY, USA, 2007. ACM.

[5] Werner Hofschuster and Walter Krämer. C-XSC 2.0: A `C++` library for extended scientific computing. In R. Alt, A. Frommer, R. B. Kearfott, and W. Luther, editors, *Numerical Software with Result Verification*, volume 2991 of *Lecture Notes in Computer Science*, pages 15–35. Springer-Verlag, 2004.

[6] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, third edition, 1998.

[7] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[8] Arnold Neumaier and Oleg Shcherbina. Safe bounds in linear and mixed-integer programming. *Mathematical Programming*, 99:283–296, 2004.

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

[10] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, September 1977.

[11] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen (fast multiplication of large numbers). *Computing: Archiv für elektronisches Rechnen (Archives for electronic computing)*, 7:281–292, 1971. (German).

5 Code

```
/*
 * Copyright (C) 2009, 2010 Thomas Steinke
 * (schonhagestrassen@thomassteinke.org) 2010-03-09
 * This is a program to test the Schonhage-Strassen Algorithm using
 * complex interval arithmetic.
 * It uses the C-XSC library for validated arithmetic.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

// to compile:
// g++ multiply.cpp -I/usr/local/include -I/usr/local/cxsc/include
//     -L/usr/local/cxsc/lib -lcxsc -lm -o SSmultiply
/*
Include standard libraries
*/
#include <iostream>
#include <cassert>
#include <fstream>
#include <ctime>

/*
Include C-XSC libraries
*/
#include "real.hpp"
#include "interval.hpp"
#include "complex.hpp"
#include "cinterval.hpp"
#include "rmath.hpp"
#include "imath.hpp"


/*
Clock resolution
*/
#ifndef CLOCKS_PER_SEC
#ifdef CLK_PER_SEC
#define CLOCKS_PER_SEC CLK_PER_SEC
#else
#error CLOCKS_PER_SEC Not Defined
#endif
#endif

using namespace std;
using namespace cxsc;

/*****************************************************************************
Random number generation code for generating random test cases
*****************************************************************************/

/*
Delegate random number generation to the operating system.
```

```
This will not work on all systems; if it doesn't, implement something else here.
*/
ifstream urand("/dev/urandom");

/*
Generate a random bit
*/
bool RandomBit() {
  return ((urand.get() % 2) == 0);
}

/*****************************************************************************
We define a class to represent natural numbers that we can then multiply. We
define basic operations. Later we implement the Schonhage-Strassen fast
multiplication algorithm.
*****************************************************************************/
typedef unsigned long long unint; //The basic hardware integer we use


/*
This class represents a natural number.
Since there are several multiplication algorithms that need to access the data
contained in Natural, they are left public and much of the management is done
outside of the class. Bad karma, but meh!
*/
struct Natural {
  unint * x; //digits
  int n; //number of digits
  int k; //number of bits per digit
  //value = x[0] + 2^k*x[1] + 2^(2*k)*x[2] + ... + 2^((n-1)*k)*x[n-1]

  /*
  Get the lth bit of n.
  */
  unint GetBit(int l) const {
    if (l < 0 || l >= n * k) return ((unint) 0);
    else return (x[l / k] >> (l % k)) & ((unint) 1);
  }

  /*
  Compare -- used for checking correctness of results
  */
  bool operator==(const Natural & other) const {
    assert(k == other.k);
    if (n != other.n) return false;
    if (n == 0) return true;
    for (int i = 0; i < n; i++)
      if (x[i] != other.x[i]) return false;
    return true;
  }

  bool operator!=(const Natural & other) const {
    return !(*this == other);
  }
};

/*
Create a random n-digit number in base 2^k
*/
Natural RandomNumber(int n, int k) {
  Natural a;
  a.n = n;
  a.k = k;
  a.x = new unint[n];
  for (int i = 0; i < n; i++) {
    a.x[i] = 0;
    for (int j = 0; j < k; j++) {
      a.x[i] = ((a.x[i] << 1) | (RandomBit() ? (unint) 1 : (unint) 0));
    }
  }
  a.x[n - 1] = (a.x[n - 1] | (((unint) 1) << (k - 1)));
  return a;
}

/*
Output hexadecimal
*/
ostream & operator<<(ostream & s, const Natural & n) {
  if (n.n == 0) {
    s << '0';
  } else {
    const char * hexdigits = "0123456789abcdef";
    assert(n.n > 0 && n.x[n.n - 1] != 0);
    int nbits = 0;
    while (n.x[n.n - 1] >= (((unint) 1) << nbits)) nbits++;
    nbits += (n.n - 1) * n.k;
    int ndigits = nbits / 4 + (nbits % 4 == 0 ? 0 : 1);
    for (int i = ndigits - 1; i >= 0; i--) {
      int k = n.GetBit(4*i) + 2 * n.GetBit(4 * i + 1) + 4 * n.GetBit(
                4 * i + 2) + 8 * n.GetBit(4 * i + 3);
      s << hexdigits[k];
```

```
    }
  }
  return s;
}

/*
Multiply two numbers using the O(n^2) elementary method
*/
Natural Multiply(const Natural & a, const Natural & b) {
  assert(a.k == b.k);
  if (a.n == 0 || b.n == 0) {//zero
    Natural cc;
    cc.x = NULL;
    cc.n = 0;
    cc.k = a.k;
    return cc;
  }
  assert((int) sizeof(unint) * 4 >=
         a.k); // if this isn't true we are entering overflow territory
  unint base = (((unint) 1) << a.k);
  Natural c;
  c.n = a.n + b.n;
  c.k = a.k;
  c.x = new unint[c.n];
  unint carry = 0;
  for (int i = 0; i < c.n; i++) {
    unint newcarry = carry / base;
    carry = carry % base;
    for (int j = (i - b.n + 1 > 0 ? i - b.n + 1 : 0); j <= i && j < a.n; j++) {
      carry += a.x[j] * b.x[i - j];
      newcarry += carry / base;
      carry = carry % base;
    }
    c.x[i] = carry;
    carry = newcarry;
  }
  assert(carry == 0);
  while (c.n > 0 && c.x[c.n - 1] == 0) c.n--;
  return c;
}

/*****************************************************************************
The Complex number handling routines. These need to provide complex arithmetic
and they need to round complex numbers to unints. Most of this is done by the
C-XSC library.
*****************************************************************************/

/*
Complex class based on a real type such as float or double
*/
template <class R> class Complex {
  R x, y; //Real and imaginary parts.
public:
  //Just the basic operations
  Complex<R>(R xx = 0, R yy = 0) : x(xx), y(yy) {}
  Complex<R> operator+(const Complex<R> & other) const {
    return Complex<R>(x + other.x, y + other.y);
  }
  Complex<R> operator-(const Complex<R> & other) const {
    return Complex<R>(x - other.x, y - other.y);
  }
  Complex<R> operator-() const {
    return Complex<R>(-x, -y);
  }
  Complex<R> operator*(const Complex<R> & other) const {
    return Complex<R>(x * other.x - y * other.y, x * other.y + y * other.x);
  }
  Complex<R> operator/(const Complex<R> & other) const {
    R t = (other.x * other.x + other.y * other.y);
    return *this * Complex<R>(other.x / t, - other.y / t);
  }
  R re() const {
    return x;
  }
  R im() const {
    return y;
  }
};

/*
Round Complex<R> x to the nearest non-negative integer
*/
template <class R> unint Roundunint(const Complex<R> & x) {
  //unint i = floor(Re(x));
  //This is sinfully inefficient, but there is no automatic function for
  //converting a real to an int should take O(log(x)) time
  unint min = 0;
  unint max = 1;
  while (((R) (double) max) <= x.re()) max *= 2;
  while (min + 1 < max) {
    unint mid = (min + max) / 2;
```

```
    if (((R) (double) mid) <= x.re()) {
      min = mid;
    } else {
      max = mid;
    }
  }
  unint i = (abs(x.re() - (R) (double) min) <= abs(x.re() - (R) (
              double) max) ? min : max);
  return i;
}

/*
Do the same with a C-XSC complex point
*/
unint Roundunint(const complex & x) {
  //unint i = floor(Re(x));
  //This is sinfully inefficient, but there is no automatic function for
  //converting a real to an int should take O(log(x)) time
  unint min = 0;
  unint max = 1;
  while (((real) (double) max) <= Re(x)) max *= 2;
  while (min + 1 < max) {
    unint mid = (min + max) / 2;
    if (((real) (double) mid) <= Re(x)) {
      min = mid;
    } else {
      max = mid;
    }
  }
  unint i = (abs(Re(x) - (real) (double) min) <= abs(Re(x) - (real) (
              double) max) ? min : max);
  return i;
}

/*
Find the unique integer in the cinterval x
If none exists throw an exception
*/
unint Roundunint(const cinterval & x) {
  assert(0 <= Im(x));
  //unint i = floor(Sup(Re(x)));
  //This is sinfully inefficient, but there is no automatic function for
  //converting a real to an int should take O(log(x)) time
  unint i = Roundunint(complex(Sup(Re(x))));
  if (((real) (double) i) <= Re(x)) {
    //do nothing
  } else if (((real) (double) i + 1) <= Re(x)) {
    i++;
  } else if (((real) (double) i - 1) <= Re(x)) {
    i--;
  } else {
    assert(false);
  }
  if (((real) (double) i + 1) <= Re(x) || (i > 0
      && ((real) (double) i - 1) <= Re(x)))  {
    throw diam(Re(x));
  }
  return i;
}

/*************************************************************************
Now we have the Schonhage-Strassen Algorithm proper (with some preceeding
subroutines).
*************************************************************************/

/*
Calculate the vector [1, w, w^2, ..., w ^ (2^n - 1)], where w = exp(2*Pi*i/2^n)
That is, we compute the 2^n-th primitive roots of unity.
R is a real class and C is a corresponding complex class.
R must be compatible with integers and have a sqrt function;
C must have a C(R, R) constructor.
*/
template<class R, class C> C * ComplexRoots(int n) {
  //(k = 0) cos(2*Pi/1) = 1
  //(k = 1) cos(2*Pi/2) = -1
  //(k = 2) cos(2*Pi/4) = 0
  //(k >= 3) cos(2*Pi/2^k) = sqrt((1 + cos(2*Pi/2^(k-1)))/2)  where the
  //square root is of a non-negative real number
  //(k = 0) sin(2*Pi/1) = 0
  //(k >= 1) sin(2*Pi/2^k) = sqrt((1 - cos(2*Pi/2^(k-1)))/2) where the
  //square root is of a non-negative real number
  //(k >= 0) exp(2*Pi*i/2^k) = cos(2*Pi/2^k) + i*sin(2*PI/2^k)
  //cosines[k] = cos(2*Pi/2^k); sines[k] = sin(2*Pi/2^k)
  R * cosines = new R[n + 3];//Extra spaces to avoid going over the end
  R * sines = new R[n + 1];
  cosines[0] = 1;
  cosines[1] = -1;
  cosines[2] = 0;
  for (int k = 3; k <= n; k++)
    cosines[k] = sqrt(((1 + cosines[k-1])/2));
  sines[0] = 0;
```

```
  for (int k = 1; k <= n; k++)
    sines[k] = sqrt(((1 - cosines[k-1])/2));
  //ans[k] = exp(2*Pi*i/2^n)^k
  C * ans = new C[1 << n];
  for (int i = 0; i < (1 << n); i++) {
    C x(1);
    for (int j = 0; j < n; j++) {
      if ((i & (1 << j)) != 0) {
        //x *= exp(2*Pi*i/2^n)^(2^j)
        x = x * C(cosines[n-j], sines[n-j]);
      }
    }
    ans[i] = x; // & exp(cinterval(0, (2 * Pi() * i) / (1 << n)));
  }
  delete[] cosines;
  delete[] sines;
  return ans;
}

/*
Reverse the binary representation of the n-bit integer x
-- used by the DFT algorithm
*/
int ReverseBinary(int n, int x) {
  int y = 0;
  for (int i = 0; i < n; i++) {
    if ((x & (1 << i)) != 0) {
      y = (y | (1 << (n - i - 1)));
    }
  }
  return y;
}

/*
Compute the discrete fourier transform of input and store in output
Both input and output have 2^n elements. Roots is a vector of the 2^n 2^n-th
primitive roots of unity if the inverse flag is set to true, then the inverse
discrete fourier transform is computed instead.
*/
template <class C> void DFT(bool inverse, int n, const C * roots, C * output,
                            const C * input) {
  int m = (1 << n);
  for (int i = 0; i < m; i++) {
    output[ReverseBinary(n, i)] = input[i];
  }
  for (int i = 1; i <= n; i++) {
    //Split into 2^(n-i) blocks of 2^i and do dft
    for (int j = 0; j < (1 << (n - i)); j++) {
      //block is j*2^i to (j+1)*2^i-1
      //w = roots[1 << (n - i)]
      for (int k = 0; k < (1 << (i - 1)); k++) {
        C u, v, w;
        int l = ((1 << (n - i)) * k) % (1 << n);
        if (inverse) {
          l = ((1 << n) - 1) % (1 << n);
        }
        w = roots[l];
        u = output[j * (1 << i) + k];
        v = w * output[j * (1 << i) + (1 << (i - 1)) + k];
        output[j * (1 << i) + k] = u + v;
        output[j * (1 << i) + (1 << (i - 1)) + k] = u - v;
      }
    }
  }
  if (inverse) {
    for (int i = 0; i < (1 << n); i++) {
      output[i] = output[i] / (1 << n);
    }
  }
}

/*
This is the Schonhage-Strassen multiplication algorithm
R and C are as in the function ComplexRoots, and represent the real and
complex types to be used for the floating-point computations.
*/
template <class R, class C> Natural SchonhageStrassen(const Natural & a,
    const Natural & b) {
  assert(a.k == b.k);
  if (a.n == 0 || b.n == 0) {//zero
    Natural cc;
    cc.x = NULL;
    cc.n = 0;
    cc.k = a.k;
    return cc;
  }
  //First we need to find a power of 2 bigger than or equal to a.n + b.n
  int n = 0;
  while ((1 << n) < a.n + b.n) n++;
  unint base = (((unint) 1) << a.k);
  //First, calculate roots
```

```
  C * roots = ComplexRoots<R, C>(n);
  //Now three working arrays
  C * work1 = new C[1 << n];
  C * work2 = new C[1 << n];
  C * work3 = new C[1 << n];
  //Load a in to work1
  for (int i = 0; i < a.n; i++) work1[i] = ((R) (double) a.x[i]);
  for (int i = a.n; i < (1 << n); i++) work1[i] = 0;
  //Do dft
  DFT<C>(false, n, roots, work2, work1);
  //Load b in to work1
  for (int i = 0; i < b.n; i++) work1[i] = ((R) (double) b.x[i]);
  for (int i = b.n; i < (1 << n); i++) work1[i] = 0;
  //Do dft
  DFT<C>(false, n, roots, work3, work1);
  //Pointwise multiplication
  for (int i = 0; i < (1 << n); i++) work1[i] = work2[i] * work3[i];
  //inverse fourier transform
  DFT<C>(true, n, roots, work2, work1);
  //Now read c out of work2
  Natural c;
  c.n = (1 << n);
  c.k = a.k;
  c.x = new unint[c.n];
  unint carry = 0;
  bool issuccessful = true;
  real ball = 0; //whatever we catch and then pass on
  try {
    for (int i = 0; i < c.n; i++) {
      carry += Roundunint(work2[i]);
      c.x[i] = carry % base;
      carry = carry / base;
    }
  } catch (real exc) {
    issuccessful = false;
    ball = exc;
  }
  //clean up
  while (c.n > 0 && c.x[c.n - 1] == 0) c.n--;
  delete[] roots;
  delete[] work1;
  delete[] work2;
  delete[] work3;
  if (!issuccessful || carry != 0) {
    delete[] c.x;
    c.x = NULL;
    c.n = 0;
    throw ball;
  }
  return c;
}

/*****************************************************************************
Now there are just some analysis routines. Testing and timing the algorithm.
*****************************************************************************/

/*
First we package every algorithm into a nice Algorithm object. Then we can
refer to algorithms just by numbers
*/
struct Algorithm {
  const char * name; // The name of the algorithm
  Natural (*function)(const Natural &,
                      const Natural
                      &); // The algorithm itself -- note that this may
  // throw an exception (in the form of a double) if it fails
  Algorithm(const char * n, Natural (*f)(const Natural &,
                                         const Natural &)) :
                                                  name(n), function(f) {}
};

Natural floatMultiply(const Natural & a, const Natural & b) {
  return SchonhageStrassen<float, Complex<float> >(a, b);
}
Natural doubleMultiply(const Natural & a, const Natural & b) {
  return SchonhageStrassen<double, Complex<double> >(a, b);
}
Natural pointMultiply(const Natural & a, const Natural & b) {
  return SchonhageStrassen<real, complex >(a, b);
}
Natural setMultiply(const Natural & a, const Natural & b) {
  return SchonhageStrassen<interval, cinterval >(a, b);
}

Algorithm algorithms[] = {
  Algorithm("elementary multiplication algorithm", &Multiply),
  Algorithm("float-based Schonhage-Strassen algorithm", &floatMultiply),
  Algorithm("double-based Schonhage-Strassen algorithm", &doubleMultiply),
  Algorithm("basic Schonhage-Strassen algorithm", &pointMultiply),
  Algorithm("Schonhage-Strassen algorithm with containment sets", &setMultiply),
};
```

```
/*
This function will run the algorithms specified by mask on two random n-digit
base-2^k numbers. If (mask & 1 != 0), then the remaining outputs are compared
to this output to check for correctness.
(algorithm 0 is deemed to be the standard for correctness)
If data != NULL, then we output the data in CSV format (for analysis).
*/
void runalgorithm(int mask, int n, int k, int r, ostream * data) {
  int numalgorithms = sizeof(algorithms) / sizeof(
                          Algorithm); // Total number of algorithms
  for(int reps=0; reps < r; reps++) {
    //Generate test data
    Natural a = RandomNumber(n, k);
    Natural b = RandomNumber(n, k);
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "n = " << n << " k = " << k << endl;
    Natural alg0;//this is the output of alg0 to compare with
    for (int i = 0; i < numalgorithms; i++) {
      if ((mask & (1 << i)) != 0) {
        int status = 0; //This identifies the outcome of the computation
        Natural axb;
        clock_t start = clock();
        try {
          axb = (*(algorithms[i].function))(a, b);
        } catch (real e) {
          cout << "The " << algorithms[i].name << " fails (" << e << ")."
               << endl;
          status = status | 1; //set the fail bit
          axb.n = 0;
          axb.k = 0;
          axb.x = NULL;
        }
        clock_t finish = clock();
        double elapsedtime = ((double) (finish - start))
                                          / ((double) CLOCKS_PER_SEC);
        cout << "The " << algorithms[i].name << " took " << elapsedtime
             << "s." << endl;
        //if we have already run algorithm 0
        if (i > 0 && (mask & 1) != 0) {
          //do check
          if (axb.x != NULL && axb != alg0) {
            //error, incorrect result
            cout << "The " << algorithms[i].name
                 << " gave an incorrect result." << endl;
            status = status | 2; //set the incorrect bit;
          }
        } else {
          status = status | 4; //set the not-checked bit
        }
        //status 0: algorithm succeeded
        //status 1: algorithm failed
        //status 2: incorrect result
        //status 3: algorithm failed
        //status 4: no error detected -- not checked
        //status 5: algorithm failed
        //status 6: invalid
        //status 7: algorithm failed
        if (data != NULL) {
          //Output data in .csv format
          // <algorithm number>,<status>,<n>,<k>,<time/s>
          *data << i << ",";
          *data << status << ",";
          *data << n << ",";
          *data << k << ",";
          *data << elapsedtime << endl;
        }
        //clean up or store for later
        if (i == 0) {
          alg0 = axb;
        } else {
          delete[] axb.x;
        }
      }
    }
    //last clean up
    if ((mask & 1) != 0) delete[] alg0.x;
    delete[] a.x;
    delete[] b.x;
    cout << endl;
  }
}

/*
The main function only calls runalgorithm. Essentially all it does is read in
4 integers and call runalgorithm with those parameters.  For example the input
"31 100 8 10" would create two 100-digit base-256 natural numbers and multiply
them with all five available algorithms 10 times.
It does provide facilities to log output. One command line parameter will
specify a file to write CSV format information to.
```

```
*/
/* you can pass 'in' into the executible 'SSmultiply' and output to 'out'
   as follows:
$ cat in | ./SSmultiply out
$ cat in
31          1024            8           100
31          2048            8           100
31          4096            8           100
31          8192            8           100
31         16384            8           100
31         32768            8           100
31         65536            8           100
*/
int main(int argc, char ** argv) {
  ostream * data = NULL;
  if (argc == 2) {
    data = (ostream *) new ofstream(argv[1], ios_base::app);
  }
  while (true) {
    int mask, n, k, r;
    cin >> mask >> n >> k >> r;
    if (cin.eof() || mask < 0
        || mask >= (int) (1 << (sizeof(algorithms) / sizeof(Algorithm)))
        || n <= 0 || k <= 1 || k > (int) sizeof(unint) * 4) break; //invalid
    runalgorithm(mask, n, k, r, data);
  }
  if (data != NULL) delete (ofstream *) data;
  return 0;
}
```