AN IMPLEMENTATION AND ANALYSIS OF FÜRER’S FASTER INTEGER MULTIPLICATION ALGORITHM

PETER SUTOR JR.
SPRING 2014

A thesis
submitted in partial fulfillment
of the requirements
for baccalaureate degrees
in Computer Science and Mathematics
with honors in Computer Science

Reviewed and approved* by the following:

Martin Fürer
Professor of Computer Science and Engineering
Thesis Supervisor

John Joseph Hannan
Associate Professor of Computer Science and Engineering
Honors Adviser

* Signatures are on file in the Schreyer Honors College.
ABSTRACT

For over 35 years, the fastest known method of computing the product of two integers has been the Schönhage-Strassen algorithm, until Fürer’s algorithm was created in 2007. Since then, however, no known implementations of Fürer’s algorithm have ever been created. In this thesis, an implementation of Fürer’s algorithm is presented, coded in Java, and analyzed. This implementation is then compared to an implementation of the Schönhage-Strassen algorithm to analyze the running speeds for increasing values of $n$, the length of the two integers to multiply in binary. For large enough $n$, Fürer’s algorithm is asymptotically faster than Schönhage-Strassen’s algorithm. The claim that an implementation of Fürer’s algorithm should then be faster for comparable $n$ is investigated and conclusions are drawn about the practicality of Fürer’s algorithm in practice, using observations from the implementation.
TABLE OF CONTENTS

List of Definitions ........................................................................................................ iii
List of Theorems ........................................................................................................... iv
List of Algorithms ...................................................................................................... v
List of Figures ............................................................................................................. vi
List of Tables ............................................................................................................... vii
Acknowledgements ................................................................................................... viii

Chapter 1: Introduction ............................................................................................... 1

Chapter 2: The Schönhage-Strassen Algorithm of Integer Multiplication .................. 4

  Section 2.1: Integer Multiplication ......................................................................... 4
  Section 2.2: The Discrete and the Fast Fourier Transform (DFT / FFT) ................. 7
  Section 2.3: The Convolution Theorem .................................................................. 13
  Section 2.4: The Schönhage-Strassen Algorithm ................................................... 16

Chapter 3: Fürer’s Algorithm of Faster Integer Multiplication .................................. 19

  Section 3.1: The Idea ............................................................................................. 19
  Section 3.2: Fürer’s Fast Fourier Transform .......................................................... 21
  Section 3.3: Fürer’s Ring $R$ ................................................................................ 25
  Section 3.4: The Faster Integer Multiplication Algorithm ................................... 28

Chapter 4: Java Implementations of the Algorithms .................................................. 35

  Section 4.1: Classes and Implementation Details ................................................. 35
  Section 4.2: The Schönhage-Strassen Implementation ............................................ 36
  Section 4.3: The Faster Integer Multiplication Implementation ............................ 38

Chapter 5: Analysis of the Implementations ................................................................ 40

  Section 5.1: Runtime Comparisons ...................................................................... 40
  Section 5.2: Implementation Analysis .................................................................... 43
  Section 5.3: Conclusions ....................................................................................... 44

Chapter 6: Appendices and References ...................................................................... 46

  Appendix A: Fürer’s Faster Integer Multiplication Java Implementation ............... 47
  Appendix B: Fürer’s Faster Integer Multiplication Java Implementation ............... 49
  Appendix C: Fürer’s Faster Integer Multiplication Java Implementation ............... 52
  Appendix D: Fürer’s Faster Integer Multiplication Java Implementation ............... 54
  Appendix E: Fürer’s Faster Integer Multiplication Java Implementation ............... 56
  Appendix F: Fürer’s Faster Integer Multiplication Java Implementation ............... 59
  References ............................................................................................................... 66


**LIST OF DEFINITIONS**

Definition 1-1. The Log Star Operation.................................................................3
Definition 2.1-1. Integer .........................................................................................4
Definition 2.1-2. Integer Multiplication.................................................................5
Definition 2.1-3. Polynomial Interpolation.............................................................7
Definition 2.2-1. The Discrete Fourier Transform..................................................9
Definition 2.2-2. Primitive \( n \)th Root of Unity.....................................................10
Definition 2.2-3. Principal \( n \)th Root of Unity.........................................................10
Definition 2.2-4. The Inverse Discrete Fourier Transform......................................10
Definition 2.3-1. The Convolution.........................................................................13
Definition 2.3-2. The Circular Convolution............................................................14
Definition 2.3-3. The Discrete Circular Convolution...............................................14
Definition 3.1-1. Rings. .........................................................................................19
Definition 3.1-2. The Fermat Numbers.................................................................20
Definition 3.2-1. Twiddle Factors...........................................................................23
Definition 3.2-2. \( N \)-point Half-DFT. .................................................................23
Definition 3.3-1. A Polynomial Ring......................................................................25
LIST OF THEOREMS

Theorem 2.1-1. Unisolvence Theorem .................................................................6
Theorem 2.2-1. DFT Bijectivity. ........................................................................10
Theorem 2.2-2. Euler’s Formula..........................................................................12
Theorem 2.3-1. The Convolution Theorem ........................................................15
Theorem 2.3-2. The Modified Convolution Theorem ..........................................15
Theorem 2.4-1. Floating Point Precision For SSA .............................................17
Theorem 3.3-1. The Chinese Remainder Theorem ..........................................25
Theorem 3.3-2. The Half-DFT Isomorphism.......................................................26
Theorem 3.3-3. Principal Nth Root of Unity for N a Power of 2 .......................27
Theorem 3.3-4. The Variable $x$ in $\mathbb{R}$.........................................................27
Theorem 3.3-5. The Norm of $\rho(x)$....................................................................27
LIST OF ALGORITHMS

Algorithm 2.1-1. The Long Multiplication Algorithm .......................................................... 5
Algorithm 2.2-1. The Cooley-Tukey Fast Fourier Transform .............................................. 11
Algorithm 2.2-2. The Inverse Cooley-Tukey Fast Fourier Transform .............................. 13
Algorithm 2.3-1. The Fast Convolution Algorithm ............................................................. 16
Algorithm 2.4-1. The Schönhage-Strassen Algorithm ......................................................... 17
Algorithm 3.4-1. Decomposition of $a$ ................................................................................. 29
Algorithm 3.4-2. Composition of $a$ ..................................................................................... 29
Algorithm 3.4-3. The Half Fast Fourier Transform ............................................................. 30
Algorithm 3.4-4. The Inverse Half Fast Fourier Transform ................................................. 31
Algorithm 3.4-5. Führer’s Fast Fourier Transform ............................................................... 32
Algorithm 3.4-6. Component-wise Multiplication ............................................................... 32
Algorithm 3.4-7. Modular Integer Multiplication ................................................................. 33
Algorithm 3.4-8. Integer Multiplication .................................................................................. 33
LIST OF FIGURES

Figure 2.2-1. Fourier Transform Example.................................................................8
Figure 2.2-2. Visualization of the Fourier Transform...................................................9
Figure 3.2-1. A Butterfly Graph of a 16 point FFT ..................................................22
Figure 3.2-2. An Element of the Result of Fürer’s FFT ..............................................24
LIST OF TABLES

Table 2.2-1. Efficiency of the FFT ................................................................. 12
Table 5.1-1. SSA and FIMA Runtime Measurements ........................................ 40
Table 5.1-2. Time Spent on Important Methods in FIMA Execution .................. 41
Table 5.1-3. Time Spent on FFT and MultiplyInR in Execution of FIMA .............. 43
ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to all those who provided me the help necessary to complete this thesis.

Firstly, I would like to thank my thesis supervisor Dr. Martin Fürer for his assistance and guidance in helping me understand his algorithm and for helping me sort through any implementation difficulties I had while working on this thesis. His readiness and patience to answer any of my questions throughout this process is greatly appreciated.

I also would like to extend my gratitude to my honors advisor John Hannan, who has expertly guided me through the Schreyer Honors College experience and helped me with any administrative details along the way, and for advising me through my time working on this thesis. In addition to this, I would also like to thank the Schreyer Honors College for giving me the opportunity to write this thesis and for assisting in accelerating my learning throughout my undergraduate years.

Last, but not least, I would like to thank all my friends and loved ones who have supported me throughout my undergraduate years and helped make writing this thesis possible. In particular, I would like to thank my parents, who have supported me all throughout my college experience and pushed me to succeed; without them, none of this would have been possible.
Chapter 1

Introduction

Up until the year 1960, it was thought that the quadratic integer multiplication algorithm, often referred to as long multiplication, grade school multiplication, or the Standard algorithm, was the asymptotically fastest algorithm for integer multiplication. The Standard algorithm is essentially the multiplication algorithm taught to children in school, multiplying integers digit by digit and then adding the products to obtain the overall product – this algorithm requires a number of elementary operations that is proportional to $n^2$, or a tight bound of $\Theta(n^2)$. In 1952, Andrey Kolmogorov conjectured that all algorithms for integer multiplication were asymptotically similar to the Standard algorithm, thus imposing a lower bound for the operation of multiplying integers of $\Omega(n^2)$, as stated in [Kar95].

However, in 1960, Kolmogorov hosted a seminar on mathematical problems in cybernetics at the Moscow State University, which involved his $\Omega(n^2)$ conjecture on integer multiplication. Soon after this, Anatoly Alexeevitch Karatsuba (a then 23 year student) devised a faster algorithm for the multiplication of two $n$ digit integers that thwarted Kolmogorov’s conjecture. This became known as the Karatsuba algorithm (see [KO63]), which has a tight bound of $\Theta(n^{\log_2 3})$. Although this asymptotic speed up seems just a tad faster when the exponents are compared (approximately 1.585 vs. 2), this translates into staggering computation time differences for large integers in actual implementations. For small values of $n$, the difference isn’t too amazing, but for large enough values it can mean the difference between waiting seconds and minutes for computation to finish.
After Karatsuba’s discovery, faster algorithms were quickly developed. The Toom-Cook algorithm, for example, is a faster generalization of Karatsuba’s algorithm. The Karatsuba algorithm splits each of the two integers into two smaller ones, recursively multiplying the lower halves, the upper halves, and the sums of the lower and upper halves. Using appropriate bit shifts, these three products are then added together to produce the overall product. See [KO63] for the full algorithm. The Toom-Cook algorithm instead splits the numbers into \( k \) parts, forming two polynomials using the coefficients of these parts, then evaluates these polynomials at certain points, multiplies them point-wise, then interpolates the result. The integer product is then reconstructed by again shifting results appropriately and adding them together. However, the Toom-Cook algorithm is actually slower than the Standard algorithm for small integers, and is used only for very large integers.

In 1971, Arnold Schönhage and Volker Strassen (see [SS71]) developed an even faster algorithm for integer multiplication, known simply as the Schönhage-Strassen algorithm, achieving an upper asymptotic bound of \( O(n \log n \log \log n) \), where \( n \) represents the number of digits in the two integers to multiply. The algorithm made use of the Fast Fourier Transform (FFT) to quickly and recursively (or iteratively) multiply together polynomial encodings of the integers until the partitions are small enough to be multiplied using a different algorithm, such as Karatsuba’s. With even one level of recursion, leaving the subsequent smaller multiplications to a slower algorithm, the Schönhage-Strassen algorithm already becomes very fast. For over 35 years after the algorithm’s creation, the Schönhage-Strassen algorithm was the fastest known method for multiplying large integers.

In the next chapter, this algorithm is thoroughly explained and pseudo code is provided to give the necessary background knowledge to understand Fürer’s algorithm, which is the subject of this thesis. See [Für09] for the full theory (and proofs) behind the algorithm. Fürer’s algorithm is based on the Schönhage-Strassen algorithm, improving the theoretical running time to
In this running time, the operation \( \log^* n \), read as “log star of \( n \),” is the iterated logarithm, which returns the number of times that the logarithm operation must be iteratively applied on \( n \) up until the resulting value \( n^* \) satisfies \( n^* \leq 1 \). More formally:

**Definition 1-1. (The Log Star Operation):** The “log star,” \( \log^* n \), is recursively defined as:

\[
\log^* n := \begin{cases} 
0 & \text{if } n \leq 1, \\
1 + \log^*(\log n) & \text{if } n > 1
\end{cases}
\]

The theoretical running time of F"urer’s algorithm is asymptotically faster than Schön­hage-Strassen for extremely large numbers, and so is closer to Schön­hage and Strassen’s conjectured optimal upper bound for integer multiplication of \( O(n \log n) \). Note that this conjectured bound is for a yet unknown algorithm. It is also important to note that up until now, F"urer’s algorithm has had no known implementations in existence.

In this thesis, a Java implementation of F"urer’s algorithm is presented and then compared to a Java implementation of Schön­hage-Strassen’s algorithm. Their average running times for increasing input sizes of length \( n \) are recorded to see if and at what point F"urer’s algorithm begins to outperform Schön­hage-Strassen’s. Both algorithms are analyzed to see if they match their theoretical running times for the input sizes of \( n \), and the difficulties of implementing F"urer’s algorithm are discussed at large, as well as any potential improvements to the algorithm. First, we begin with discussing the theory behind the Schön­hage-Strassen algorithm and how it is implemented in Chapter 2.
Chapter 2

The Schönhage-Strassen Algorithm of Integer Multiplication

In this chapter we will discuss the theory behind Schönhage-Strassen’s algorithm and provide pseudo-code for the algorithm, for the sake of completeness, as it is shown in [SS71]. Fürer’s algorithm is a modification to this algorithm and so it is integral that one understands the reasoning behind Schönhage and Strassen’s method of integer multiplication. First we define the problem of integer multiplication explicitly.

2.1 Integer Multiplication

We define an integer as follows, according to [SS10]:

Definition 2.1-1. (Integer): An integer \( x \) in base \( b \), comprised of \( n \) digits, is defined as a vector \( x \in \mathbb{Z}_b^n := \{0, 1, ..., b - 1\}^n \) by the following summation, where \( x_i \) is the \( i \)th entry in the vector:

\[
x = \sum_{i=0}^{n-1} x_i b^i
\]

The base \( b \) can be any positive value, but the most common bases are some integer \( 2^k \) for \( k \in \mathbb{N} \). For simplicity, we use the binary base \( b = 2 \) for our implementations. From the definition above, we can easily derive an equation for calculating the product of two integers.
Definition 2.1-2. (Integer Multiplication): Let \( x \) and \( y \) be positive integers such that \( x \in \mathbb{Z}_b^n \) and \( y \in \mathbb{Z}_b^m \). Then the product \( z = xy \in \mathbb{Z}_b^{n+m} \) is:

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

However, the coefficients of \( z \) cannot just be set as \( z_i = \sum_{j=\max\{0,i-m+1\}}^{\min\{n-1,i\}} x_j y_{i-j} \) since we may exceed the range of possible values \( \{0,1,...,b-1\} \) for any \( i \). So, we have to carry the overflowing values to the next digit. Interpreting the above integer multiplication algorithmically, we can deduce an algorithm (see [SS71]) to multiply these integers and handle the carries for us:

**ALGORITHM 2.1-1: The Long Multiplication Algorithm**

**Input:** Integers \( x \in \mathbb{Z}_b^n \) and \( y \in \mathbb{Z}_b^m \)

**Output:** Integer \( z = xy \in \mathbb{Z}_b^{n+m} \)

1. \( c := 0 \)
2. \( \text{For } i=0 \text{ to } n+m-1 \) do
3. \( s := 0 \)
4. \( \text{For } j = \max\{0,i-m+1\} \text{ to } \min\{n-1,i\} \) do
5. \( s := s + x_j y_{i-j} \)
6. \( z_i := (s+c) \mod b \)
7. \( c := \text{floor}\left(\frac{(s+c)}{b}\right) \)

When addition is treated as a constant-time operation (in a computer, it normally is, so long as the addition is contained to the size of the registers), then we have that this algorithm runs in \( O(nm) \) time. If we let \( n \) be the length of both integers, then we clearly see that the Long Multiplication algorithm is \( O(n^2) \). For large \( n \), we would like to reduce the number of operations necessary to multiply the two integers so that we do not have a quadratic blow-up in running time.
If one closely examines definition 2.1.1, what we call an integer, it becomes apparent than we essentially convert an integer into a polynomial represented as a vector of its coefficients, with the base $b$ as the input variable. So in definition 2.1.2, Integer Multiplication is comparable to multiplying two polynomials that give us our two integers when evaluated at $b$, to get a product polynomial that gives us our product when evaluated at $b$ as well. The algorithm uses a similar principle. So then, polynomial multiplication is very pertinent to integer multiplication.

It is clear that we can first compute the product of the two polynomials and then evaluate the result to get their product at $b$. Alternatively, we can first evaluate the two polynomials at $b$ and then multiply them to get the evaluation of the product at $b$. If we use real numbers for evaluation instead of integers, such as some $k \in \mathbb{R}$, and let $x$, $y$ and $z$ be polynomials that accept a real number, then we see that the following holds for all $k$:

$$z(k) = x(k) \cdot y(k)$$

Now consider the following theorem on polynomials, known as the Unisolvence Theorem:

**Theorem 2.1-1. (Unisolvence Theorem):** A polynomial of degree $d - 1$ is determined uniquely by its values at $d$ distinct points.

According to this theorem, the product polynomial $z$ can be determined by evaluating the products of $x$ and $y$ at $n$ points. Thus, we naturally have that polynomial multiplication can be reduced to evaluating $x$ and $y$ at $n$ points, multiplying them pairwise, and then interpolating the result. Recall that polynomial interpolation will give us the polynomial $z$ we require, and that it is defined as follows:
Definition 2.1-3. (Polynomial Interpolation): Given \( n + 1 \) evaluations of a polynomial \( P(x) \), each represented in the form \( P(x_i) \), where \( i \in \{0, 1, ..., n\} \), we can construct \( P(x) \) as:

\[
P(x) = \sum_{i=0}^{n} P(x_i) \prod_{0 \leq j \leq n, j \neq i} \frac{x - x_j}{x_i - x_j}
\]

In fact, Karatsuba was the first to use this principle to multiply integers in his algorithm using polynomials of degree 1 in [KO63]. The Toom-Cook algorithm generalizes this to larger degrees. Unfortunately, this evaluation/interpolation strategy is quite costly; for fixed \( d \) degree polynomials representing two \( n \) length integers, it runs in \( O(dn \log(d)) \) time in the Toom-Cook method. Thus we require a faster way of interpolating and evaluating polynomials. This leads us to the first major component of the Schönhage-Strassen algorithm: the Fast Fourier Transform.

2.2 The Discrete and the Fast Fourier Transform (DFT / FFT)

The Discrete Fourier Transform has the potential to help us with the integer multiplication algorithm, provided we can carry out the transformation fast enough – this is where the Fast Fourier Transform (FFT) comes into play. For brevity’s sake, however, the Discrete Fourier Transform (DFT) will be covered in detail before introducing the FFT and describing its important role in the Schönhage-Strassen algorithm; see [SS71] for more details.

Loosely speaking, the DFT converts a finite (discrete) list of equally spaced samples of a particular function into a list of coefficients of a finite combination of complex sinusoids in order of their frequencies. These have the same values as the sample values. A short, but rather apt, way of putting it is that the DFT converts the function samples from their domain (such as time or linear position) to the frequency domain. Thus, it views the data given to it as a periodic function...
involving that data. Figure 2.2-1 shows an example of this. Theoretically, the input is a list of complex numbers, though most restrict themselves to real numbers in applications, and the output is a list of complex numbers. The frequencies the output represents (the sinusoids) are integer multiples of a base frequency that has a period of equal length to the length of the interval in which the samples are taken. This is similar to the traditional Fourier Transform that most are familiar with, but discrete. Figure 2.2-2 gives a great visual interpretation of the relationship between the original function and the output of the Fourier Transform.

**Figure 2.2-1. (Fourier Transform Example):** A list of 10 data points in (a) being converted into a periodic function in (b). The DTF treats the data in (a) as if it were a function like (b).
Figure 2.2-2. (Visualization of the Fourier Transform): A visual interpretation of the Fourier Transform. The input function is in red, the domain of which the Fourier Transform relates to its frequency domain, shown in blue. Each component frequency (of the related sinusoids) is represented as peaks in the frequency domain.

Now that we have given an informal description of the DFT, the next step is to define it more formally. Over the field of complex numbers, the DTF is essentially a map from $\mathbb{C}^n$ to $\mathbb{C}^n$. We define the DFT as in [SS10], by following (note that the $i$ in $\omega$ is the imaginary unit):

**Definition 2.2-1. (The Discrete Fourier Transform):** Let $x \in \mathbb{C}^n$ and $\omega := e^{2\pi i / n}$. Then the Discrete Fourier Transform (DTF) $\hat{x} \in \mathbb{C}^n$ of $x$, where $\hat{x}_i$ is the $i$th entry in $\hat{x}$, is:

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

This is also known as an $n$-point DTF. The heart of this transformation lies in the term multiplying $x_j$, and thus is dependent on the choice of $\omega$. It is convenient to let $\omega = e^{2\pi i / n}$, as it
is a principal $n$th root of unity. The definition of the primitive $n$th root of unity, and the stronger definition of the principal $n$th root of unity is as follows, according to [Fü 09]:

**Definition 2.2-2. (Primitive $n$th Root of Unity):** An element $\omega$ in a ring $R$ is a primitive $n$th root of unity if it has the following properties:

1. $\omega^n = 1$
2. $\omega^k \neq 1$ for $1 \leq k < n$.

**Definition 2.2-3. (Principal $n$th Root of Unity):** An element $\omega$ in a ring $R$ is a principal $n$th root of unity if it is a primitive $n$th root of unity and has the following property:

$$\sum_{j=0}^{n-1} \omega^{jk} = 0 \text{ for } 1 \leq k < n$$

The DFT is a bijection, which the following definition of the Inverse DFT (IDFT) will help us show (for a proof that Definition 2.2-4 is the inverse of the DTF, see [Fü 09]):

**Definition 2.2-4. (The Inverse Discrete Fourier Transform):** Let $x \in \mathbb{C}^n$ and $\omega := e^{2\pi i/n}$.

Then the IDFT $\tilde{x} \in \mathbb{C}^n$ is defined as follows, where $\tilde{x}_i$ is the $i$th element in $\tilde{x}$:

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

Thus we arrive at the following trivial theorem about DTFs and IDFTs:

**Theorem 2.2-1. (DFT Bijectivity):** For $y = \hat{x} = \text{DFT}(x)$, where $x$ is a vector, we have that $\tilde{y} = \text{IDFT}(y) = x$. 
Now that we have defined the DFT, we can now define a fast method of computing the DFT, known as the Fast Fourier Transform (FFT). According to [SS10], if we let $n$ be a power of 2 and we have a $x \in \mathbb{C}^n$, then we can create even and odd partitions of $x$ by setting $x_{e,i} := x_{2i}$ and $x_{o,i} := x_{2i+1}$ for all $i \in \{0, 1, \ldots, n-1\}$. According to [CT65], if we fix $i$, then by the definition of the DFT we have:

$$
\hat{x}_i = \sum_{j=0}^{n-1} x_j \omega^{ij} = \sum_{j=0}^{n/2-1} x_{e,j} (\omega^2)^{ij} + \omega^i \sum_{j=0}^{n/2-1} x_{o,j} (\omega^2)^{ij} = \hat{x}_{e,i \mod (n/2)} + \omega^i \hat{x}_{o,i \mod (n/2)}
$$

As $(\omega^2)^{n/2} = 1$, the modulus is justified in the above equation. Note that the $mod$ here means “take $i$, divide it by $n/2$ and return the remainder.” Thanks to this result, we can create a divide-and-conquer (recursive) algorithm to compute the FFT of a given $x$. This is known as the Cooley-Tukey FFT algorithm (see [CT65]) and according to [SS10], we can formulate from it the following algorithm:

**ALGORITHM 2.2-1: The Cooley-Tukey Fast Fourier Transform**

**Input:** $n = 2^k$ and $x \in \mathbb{C}^n$

**Output:** $\hat{x} \in \mathbb{C}^n$, the FFT of $x$

1. function CT-FFT($k, x$)
2.     if $k = 0$ then $\hat{x} = x$
3.     Partition $x$ into $x_e, x_o \in \mathbb{C}^n$
4.     $\hat{x}_e = \text{CT-FFT}(k-1, x_e)$ [by recursion]
5.     $\hat{x}_o = \text{CT-FFT}(k-1, x_o)$ [by recursion]
6.     $\omega := e^{2\pi i/n}$
7.     for $i = 0$ to $n-1$ do
8.         $\hat{x}_i = \hat{x}_{e,i \mod (n/2)} + \omega^i \hat{x}_{o,i \mod (n/2)}$

The only problem with the algorithm above is the issue of computing $\omega$, which is complex. To do this, we can employ Euler’s Formula, a famous result of Leonhard Euler that relates the complex exponential to trigonometric functions. Euler’s Formula is as follows:
**Theorem 2.2. (Euler’s Formula):** Given any \( x \in \mathbb{R} \), the following holds true:

\[
e^{xi} = \cos x + i \sin x
\]

With this formula, we can represent \( \omega \) as a complex number in the form \( a + bi \), where \( a \) and \( b \) are real numbers. This then completes our algorithm for computing the FFT. Note that evaluating the DFT directly by its definition would require \( O(n^2) \) operations, whereas the FFT runs in \( O(n \log n) \) time, which drastically reduces computation time for large \( n \). A look at table 2.2-1 shows the difference in the number of complex operations between the FFT and evaluating the DFT from its definition for increasing values of \( n \), to illustrate the efficiency of the FFT. Additionally, with respect to round-off error in floating point representations of numbers (as in a computer), the FFT also reduces the amount of error in the final result in comparison to direct evaluation, as less operations are necessary and so less error is introduced.

<table>
<thead>
<tr>
<th>( N )</th>
<th>DFT</th>
<th>FFT</th>
<th>Computational Effort of FFT Compared to DFT (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>64</td>
<td>24</td>
<td>37.50</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>160</td>
<td>15.62</td>
</tr>
<tr>
<td>256</td>
<td>65536</td>
<td>2048</td>
<td>3.12</td>
</tr>
<tr>
<td>1024</td>
<td>1048576</td>
<td>10240</td>
<td>0.98</td>
</tr>
<tr>
<td>4096</td>
<td>16777216</td>
<td>49152</td>
<td>0.29</td>
</tr>
</tbody>
</table>

**Table 2.2-1. (Efficiency of the FFT):** This table highlights the efficiency of the FFT in comparison to the DTF by measuring the number of simple operations required for input of size of \( N \) between the two, and showing the fraction of operations required in the FFT in percent, relative to the DTF.

Finally, from the definition of the FFT, we can create an algorithm for the inverse FFT using the Cooley-Tukey Algorithm above. To do this, we need to add another input to the FFT
algorithm that specifies whether or not to use a negative power for $\omega$ — **true** to do so and **false** to use a positive power. The algorithm for the inverse FFT (IFFT) then would be as follows:

**ALGORITHM 2.2-2: The Inverse Cooley-Tukey Fast Fourier Transform**

**Input:** $n = 2^k$ and $x \in \mathbb{C}^n$

**Output:** $\hat{x} \in \mathbb{C}^n$, the IFFT of $x$

1. **function** $\text{CT}_-\text{IFFT}(k, x)$
2. $\hat{x} := \text{CT}_-\text{FFT}(k, x, \text{true})$
3. **For** $i = 0$ **to** $n - 1$ **do**
4. $\hat{x}_i := \hat{x}_i / n$

### 2.3 The Convolution Theorem

The FFT is at the heart of the computations performed in the Schönhage-Strassen Algorithm (SSA), but the Convolution Theorem is the central aspect of the process that allows us to multiply large integers so quickly. In this section, we describe convolutions and the Convolution Theorem and how it relates to the SSA.

We first start by loosely describing a convolution. A convolution is a mathematical operator that takes two functions as input and produces a third function that expresses the area of overlap between these two functions. This is a function of the amount that the second function is translated; i.e. shifted around the first function. As we are measuring the area under a curve by doing this, the convolution’s definition involves integrals. It is defined as follows:

**Definition 2.3-1. (The Convolution):** The convolution of functions $f$ and $g$, written $f \ast g$, is defined as the integral of the product of the two functions with $g$ reversed and shifted, or:
\[(f \ast g)(t) := \int_{-\infty}^{\infty} f(x)g(t - x)dx\]

There are many variations to this definition; we will focus on only one of them – the circular convolution. In this case, the second function is periodic, with a given period of \(T\), and so the convolution is also periodic:

**Definition 2.3-2. (The Circular Convolution):** Given functions \(f\) and \(g\), where \(g\) is periodic with a period of \(T\), the convolution (if it exists) is also periodic and is equivalent to the following, where \(t_0\) is arbitrary:

\[(f \ast g)(t) := \int_{t_0}^{t_0+T} \left( \sum_{t=\infty}^{-\infty} f(x + it) \right) g(t - x)dx\]

When discretized within a period \(N\), the above becomes:

**Definition 2.3-3. (The Discrete Circular Convolution):** Given functions \(f\) and \(g\), where \(g\) is periodic with a period of \(N\), the convolution (if it exists) is defined as:

\[(f \ast g)(n) := \sum_{m=0}^{N-1} \left( \sum_{t=\infty}^{-\infty} f(m + iN) \right) g(n - m)\]

The convolution is important because of the following theorem, which relates Fourier Transforms and convolutions. This is the convolution theorem.
Theorem 2.3-1. (The Convolution Theorem): Let $f$ and $g$ be two functions with the convolution $f \ast g$, let $F$ and $F^{-1}$ symbolize the Fourier Transform operator and its inverse, and let the sign $\cdot$ symbolize point-wise multiplication. Then:

1. The Fourier Transform $F[f \ast g] = F[f] \cdot F[g]$ 
2. The point-wise product $F[f \cdot g] = F[f] \ast F[g]$ 
3. The convolution $f \ast g = F^{-1}[F[f] \cdot F[g]]$

This theorem states that the Fourier Transform of a convolution is a point-wise product of Fourier Transforms. By restricting the convolution theorem slightly using the discrete circular convolution and modifying it to fit our problem of multiplying integers, we have the following theorem, according to [SS10]:

Theorem 2.3-2. (The Modified Convolution Theorem): Let $f, g \in \mathbb{C}^n$ and $h = f \ast g \in \mathbb{C}^m$, with $m = 2n - 1$. By padding $f$ and $g$ with 0’s until they are also in $\mathbb{C}^m$, we have that for every $i$ such that $0 \leq i \leq m - 1$:

$$F[h] = F[f] \cdot F[g]$$

and

$$F^{-1}[F[h]] = F^{-1}[F[f] \cdot F[g]] = h = f \ast g$$

This is the key to the SSA. We can obtain our polynomial representing the product of the polynomial forms of our integers by performing a FFT on each integer as a polynomial, multiplying the result point-wise, and then computing the inverse FFT of the result. This gives rise to a fast algorithm to perform a convolution, as follows:
Algorithm 2.3-1: The Fast Convolution Algorithm

Input: $x, y \in \mathbb{C}^n$

Output: $z = x \ast y \in \mathbb{C}^m$

1. $k := \lceil \lg(2n - 1) \rceil$
2. $m := 2^k$
3. $x_{pad} := (x_0, \ldots, x_{n-1}, 0, \ldots, 0) \in \mathbb{C}^m$
4. $y_{pad} := (y_0, \ldots, y_{n-1}, 0, \ldots, 0) \in \mathbb{C}^m$
5. $\hat{x} := CT_{\text{FFT}}(k, x_{pad})$
6. $\hat{y} := CT_{\text{FFT}}(k, y_{pad})$
7. For $i = 0$ to $m - 1$ do
   8. $\hat{z}_i = \hat{x}_i \hat{y}_i$
   9. $z = CT_{\text{IFFT}}(k, \hat{z})$

Note that this algorithm executes in $O(n \log n)$, which is much faster than the naïve alternative, helping keep the complexity of the SSA low.

2.4 The Schönhage-Strassen Algorithm

Now we have all the knowledge necessary to derive the SSA. Essentially, we represent our given integers as polynomials in the form of vectors that contain the coefficients for each power. To multiply these two integers, the algorithm performs a discrete circular convolution on these polynomials to obtain the product polynomial. Before this, we may have to convert our integers into the desired base that we use in the algorithm, and after the convolving, the product integer must be rebuilt by adding the resulting coefficients in a shifted way and performing carries. This is done exactly the same way one would add up products at the last step of the long multiplication algorithm in section 2.1, or basically the way each product is added up in grade school multiplication. Thus, we finally arrive at the SSA algorithm, which is as follows:
Algorithm 2.4-1: The Schönhage-Strassen Algorithm

**Input:** Integers \( x \in \mathbb{Z}_b^n \) and \( y \in \mathbb{Z}_b^n \)

**Output:** Integer \( z = xy \in \mathbb{Z}_b^{2n} \)

1. Perform the Fast Convolution Algorithm on \( x \) and \( y \) to obtain \( \text{convolution} = x \ast y \)
2. \( c := 0 \)
3. For \( i = 0 \) to \( 2n - 2 \) do
4. \( z_i = (\text{convolution}_i + c) \mod b \)
5. \( c = \lfloor (\text{convolution}_i + c) / b \rfloor \)
6. \( z_{2n-1} = c \)

Note that \( \text{convolution}_i \) means the \( i \)th element of the vector output from convoluting \( x \) and \( y \). By observing the complexity of the components of this algorithm, we can clearly see that it runs in \( O(n \log n) \) time. However, this is assuming that the point-wise multiplications and operations in the FFT algorithms take constant time. The reason this assumption is somewhat valid is because computer circuitry (namely registers) can perform multiplication in near constant time that is similar to that of something like addition. When we can no longer use circuitry to multiply, say that we exceed the standard 64 bit registers that computers have, then multiplication is no longer constant time and the complexity increases.

This can be avoided by using floating point arithmetic, but then the SSA suffers from error accumulation due to truncation, so there will be a point at which the error causes incorrect results. Schönhage and Strassen in [SS71] proved the following about precision in the SSA:

**Theorem 2.4-1. (Floating Point Precision For SSA):** Floating point numbers operating with \( O(\log(bn)) \)-bits are sufficient to multiply \( n \) digit integers in base \( b \).

There is also another issue with the above SSA; the Cooley-Tukey FFT tends to exhibit poor cache locality which is exacerbated by the large vector sizes required in the SSA. Also, Schönhage and Strassen in [SS71] devised another way of performing their algorithm with
modular arithmetic and number theoretic transforms to avoid certain problems such as these, which runs in $O(n \log n \log (\log n))$ time; actually, the original SSA algorithm is this version of it. The algorithm described above is a simplified version involving Complex numbers and is usually referred to as the Complex SSA. Whenever the SSA is mentioned in this paper, it will refer to the Complex SSA, unless said otherwise.

Some SSA implementations combine both SSA algorithms to form a single two step algorithm. There are also many other optimized versions of the SSA, which are sometimes optimized for specific computation. A common strategy to improve memory usage and execution speed is to make the FFT an iterative procedure, so that recursion is not necessary, and to pre-compute powers of $\omega$ so that they can be accessed from memory in constant time and do not have to be recomputed repeatedly (simply make an array containing these powers with the index of the array being the power of $\omega$). We will only use the general Complex SSA algorithm described above to compare with Fürer’s Algorithm. The complete pseudo code for the SSA is shown in appendix A, for reference.
Chapter 3

Fürer’s Algorithm of Faster Integer Multiplication

3.1 The Idea

From here on out, we will refer to Fürer’s Faster Integer Multiplication Algorithm as FIMA, for conciseness. The FIMA is an extension, and in some sense, a modification of the SSA. It is based on a couple of crucial ideas that give it a faster run time in the long run, as stated in [Für09]:

- A special FFT procedure is used with the convenient property that the majority of the roots of unity that arise are of low order.
- A special ring is used where multiplication operations involving low order roots of unity are simple to compute. Additionally, this ring also contains high order roots of unity.

A previous result of Fürer’s is used (see [Für89]) to show what this special ring should be, and how the properties benefit the algorithm itself. As a consequence of the above ideas, the FFT will certainly operate differently. Instead of a vector of polynomial coefficients being used as input, a vector of polynomials is used.

Before discussing the above at length, we will first point out a few things about the second SSA mentioned at the end of chapter 2. The second SSA performs arithmetic in special rings. A ring is defined as follows:

**Definition 3.1-1. (Rings):** A ring is a set $R$ containing the binary operations $+$ and $\cdot$ such that the following axioms are satisfied, which are known as the ring axioms:
1. \( \forall a, b, c \in R, (a + b) + c = a + (b + c). \)
2. \( \forall a \in R, \exists \text{an additive identity } 0 \in R \text{ such that } a + 0 = a. \)
3. \( \forall a \in R, \exists \text{an additive inverse } -a \in R \text{ such that } a + (-a) = 0. \)
4. \( \forall a, b \in R, a + b = b + a. \)
5. \( \forall a, b, c \in R, a \cdot (b \cdot c) = (a \cdot b) \cdot c. \)
6. \( \forall a \in R, \exists \text{a multiplicative identity } 1 \in R \text{ such that } a \cdot 1 = a \text{ and } 1 \cdot a = a. \)
7. \( \forall a, b, c \in R, a \cdot (b + c) = a \cdot b + a \cdot c. \)
8. \( \forall a, b, c \in R, (b + c) \cdot a = b \cdot a + c \cdot a. \)

The first four axioms indicate that the ring \( R \) is an abelian group under addition, the two after that mean it is also a monoid under multiplication, and the last two indicate that multiplication distributes over addition. The original SSA described in [SS71] performs arithmetic over the ring of integers modulo \( F_M \) – where \( F_M \) are known as the Fermat Numbers. These are defined as:

**Definition 3.1-2. (The Fermat Numbers):** A Fermat Number is an integer of the form

\[
F_M = 2^{2^M} + 1
\]

The ring of integers modulo \( F_M \) is referred to as \( \mathbb{Z}_{F_M} \). The disadvantage of this is that using such a ring reduces the length of factors from \( n \) to \( O(\sqrt{n}) \). The second method recursively operates with \( O(\log (\log n)) \) nested calls. While the complex SSA required more complicated basic steps (multiplications), but only \( O(\log^* n) \) nested recursive calls, the Fermat Number based SSA does only simple shift operations. However, it also needs to perform these \( O(\log (\log n)) \) nested recursive calls, due to the fact the integer length is reduce from \( n \) to \( O(\sqrt{n}) \) only. In \( \mathbb{Z}_{F_M} \),
2 is a convenient root of unity for the FFT computation as multiplying with it is simply a modified cyclic shift. Thus, the time spent at each recursive level of the algorithm is basically fixed, and has no blowup factor.

However, as [Für09] goes on to say, the Complex SSA described in chapter 2 provides a significant length reduction after one level of recursion from $n$ to $O(\log n)$. Applied recursively with $\lg^* n - O(1)$ nested levels reduces the running time to order $n \log n \log (\log n) \cdots 2^{O(\lg^* n)}$. At the first level, $O(n \log n)$ time is spent, and for the $k$th recursion level the amount of time required will increase by a factor of $O(\log \log \cdots \log n)$, where the $\log$ operation is performed $k + 1$ times, as opposed to the time spent at the previous level.

The special FFT algorithm used in FIMA incorporates the advantages of both these forms of the SSA. The special ring used is a bit more complex to explain, however, and will be discussed at length in section 3.3. For now, we first start by describing how this special FFT works in detail.

### 3.2 Fürer’s Fast Fourier Transform

The special FFT used in Fürer’s algorithm uses a divide and conquer approach to the standard N-point FFT, but it is different than the Cooley-Tukey FFT described in section 2.2. Typically, such FFTs break down a DFT of composite size $N = JK$ into $J$ smaller transforms of size $K$, where typically $J$ is the radix used throughout the transform. These smaller DFTs are then combined together through butterflies of size $J$. A butterfly is just a section of the computation path of a FFT that combines the results of smaller DFTs, and are named as such due to how the graphical representation of these computation paths resemble butterfly wings, as figure 3.2-1 shows.
In the case of Fürer’s FFT, we are only interested in $N = JK$ being a power of 2. Most applications of the FFT choose either $J$ or $K$ to be 2, whereas Fürer uses a more balanced approach in the decomposition of $N$. With an appropriate ring, it allows most multiplications by powers of $\omega$ to be cyclic shifts. According to [Für09], it is known that $JK$-point FFTs are 2-staged; the first contains $K$ copies of a $J$-point FFT and the second contains $J$ copies of $K$-point FFT. The factoring of $N$ into $JK$ can be done in any way, and each way of partitioning $N$ would produce a different FFT algorithm with different $\omega$ powers, where $\omega$ is the principal $N$th root of unity. These powers are known as **twiddle factors**. More formally:

![Figure 3.2-1. (A Butterfly Graph of a 16 point FFT): A graph of the computation path of a 16 point FFT, known as a butterfly graph. The name comes from the top of the image where the triangles form butterfly wings. Figure taken from [Für09].]
**Definition 3.2-1. (Twiddle Factors):** A twiddle factor is a trigonometric constant coefficient multiplying the data in a FFT algorithm. Most commonly, it refers to any of the root of unity complex multiplicative constants in the butterfly operations of a Cooley-Tukey FFT.

For Fürer’s FFT, we need to talk about a special DFT, referred to as a Half-DFT by [Für09]. The $N$-point DFT is a transform over a ring and is a linear map mapping an $N$ element vector $x$ to a vector $y$ such that $y = (\omega^{jk})_{0 \leq j, k \leq N-1} x$ for a given principal $N$th root of unity $\omega$.

An $N$-point Half-DFT, in the case of $N$ being a power of 2, evaluations are done at $N$ odd powers of $\zeta$, which is a principal $2N$th root of unity. We pad the $N$ element vector with 0’s until it is a $2N$ element vector and performing a $2N$-point DFT – note that only half the work is necessary due to the 0 padding. The Half-DFT then maps the padded vector $x$ to $y$ by

$$y = (\zeta^{j(2k+1)})_{0 \leq j, k \leq N-1} x.$$ Note that the subscripts in relations such as these indicate matrix entries. So then, any $j$th element of $y$ is expressed as:

$$y_j = \sum_{k=0}^{N-1} \zeta^{(2j+1)k} x_k = \sum_{k=0}^{N-1} \omega^{jk} \zeta^k x_k \quad (0 \leq j < N \text{ and } \omega = \zeta^2)$$

We find that the corresponding matrix multiplying $x$ in the mapping is diagonal with the first $N$ powers of $\zeta$, starting with power 0, in the top left corner of the matrix. So then, as $N$ is a power of 2 and $\zeta$ is a principal $2N$th root of unity, the $N$-point Half-DFT is a scaling operation that is followed by a standard $N$-point DFT with $\omega = \zeta^2$. More formally:

**Definition 3.2-2. (N-point Half-DFT):** An N-point Half-DFT is a linear map between $2N$-element vectors $x$ (input) and $y$ (output), such that for a principal $2N$th root of unity $\zeta$:

$$y = \begin{bmatrix} \omega^{0 \cdot 0} & \cdots & \omega^{0 \cdot (N-1)} \\ \vdots & \ddots & \vdots \\ \omega^{(N-1) \cdot 0} & \cdots & \omega^{(N-1) \cdot (N-1)} \end{bmatrix} \begin{bmatrix} \zeta^0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \zeta^{N-1} \end{bmatrix} x$$
Now we can move on to describing Führer’s FFT. Let $N = JK$, where $N, J, K$ are all powers of 2; by definition of a corresponding principal $N$th root of unity $\omega$, we know that $\omega^{JK} = 1$. According to [Für09], the $N$-point DFT we would like to perform is represented as a set of $K$ parallel $J$-point DFTs and a set of $J$ parallel $K$-point DFTs – the inner and outer DFTs, respectively. These inner DFTs use the principal $J$th root of unity $\omega^K$ and vice versa for the outer DFTs. Thus, the twiddle factors used have powers that even and only the scalar multiplications in between the inner and outer DFTs use odd powers of $\omega$. This recursive DFT decomposition was presented in Cooley and Tukey’s original paper, [CT65]. With even $J = 2$ or $K = 2$, we still perform a fast DFT – in fact, this is the FFT algorithm! Once more, in this case we use an $N$ that is a power of 2. Furthermore, instead of indices $j$ and $k$ ranging from 0 to $N − 1$, Führer’s FFT uses indices $j'J + j$ and $k'K + k$, where $0 ≤ j, k' ≤ J − 1$ and $0 ≤ j', k ≤ K − 1$.

The result is that for $0 ≤ j ≤ J − 1$, $0 ≤ j' ≤ K − 1$, an element of the resulting vector $y$ that is mapped to can be expressed as in figure 3.2-2, taken from [Für09], using the definition of the DFT from section 2.2. As $N$ is a power of 2, the FFTs are obtained by recursive application of this until the base case of $N = 2$. 
Figure 3.2-2. (An Element of the Result of Fürer’s FFT): Applying the definition of the DFT gives the above decomposition for an element of vector $b$, the result of the FFT, in terms of vector $a$, the input vector. The brackets indicate the correspondence to the inner / outer DFTs.

Formal pseudo code for this FFT will be provided in section 3.4. But before this, we first have to talk about the ring in which the FFT will perform its arithmetic in, as elements of input vector $x$ are actually polynomials themselves, and so the computation scheme is a bit different from what was covered in chapter 2.

3.3 Fürer’s Ring $R$

Before we delve too deep into this section, we will first give some background information on Polynomial Rings and the Chinese Remainder Theorem:

Definition 3.3-1. (A Polynomial Ring): Given a variable $x$ and a commutative ring $R$, the set of polynomials $R[x] = \{a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0 : n \geq 0, a_j \in R\}$ creates a
commutative ring with the standard addition and multiplication operations, containing $R$ as a subring. This is called a polynomial ring.

The Chinese Remainder Theorem is as follows:

**Theorem 3.3-1. (The Chinese Remainder Theorem):** Suppose that $n_1, ..., n_k \in \mathbb{Z}$ are positive and pairwise coprime. Then, given any sequence $a_1, ..., a_k \in \mathbb{Z}$, there exists an integer $x$ solving the system of simultaneous congruences:

$$x \equiv a_1 \pmod{n_1}, x \equiv a_2 \pmod{n_2}, ..., x \equiv a_k \pmod{n_k}$$

Also, all solutions $x$ of this are congruent modulo the product $N = n_1 n_2 \ldots n_k$. Thus, $\forall i \in \mathbb{N}$ such that $1 \leq i \leq k, x \equiv y \pmod{n_i} \iff x \equiv y \pmod{N}$. It is not necessary that the $n_i$’s are pairwise coprime to solve this system. A solution $x$ exists if and only if $a_i \equiv a_j (\pmod{\gcd(n_i, n_j)})$ for all $i$ and $j$. All solutions $x$ are the congruent modulo the least common multiple of $n_i$.

Basically, this theorem means that for any positive integer $n$ with a prime factorization, there is an isomorphism (an invertible structure-preserving map) between a ring and the direct product of the prime power parts.

In FIMA, according to [Für09], we consider the ring of polynomials $R[y]$ over the ring $R = \mathbb{C}[x]/(x^p + 1)$, where $P$ is a power of 2; this gives us the special properties mentioned in section 3.1. Given a primitive $2P$th root of unity $\eta \in \mathbb{C}$, such as $\eta = e^{\pi i/p}$, where $i$ is the imaginary unit, we have that $R$ is isomorphic to $\prod_{j=0}^{p-1} \mathbb{C}[x]/(x - \eta^{2j+1})$. This is due to the Chinese Remainder Theorem, which in the case of fields gives us the isomorphism as $x^p + 1 = \prod_{j=0}^{p-1} (x - \eta^{2j+1})$. Furthermore, we see that this isomorphism is also isomorphic to $\mathbb{C}^p$. Since we wish to perform Half-DFTs over the ring $R$, and it is clear from the above that polynomials over
this ring decompose into products of polynomials over the complex numbers, then \( R[y] \) is isomorphic to \( \mathbb{C}[y]^p \). Each of these \( \mathbb{C}[y] \) have a unique factorization. Then the components of \( \zeta \) that form a vector in \( \mathbb{C}^p \), \((\zeta^0, ..., \zeta^{p-1})^T\), are also principal 2Nth roots of unity in the field \( \mathbb{C} \). This means that \( (y - \zeta_k^{2j+1}) \) divides \( y^N + 1 \) and that the greatest common denominator of \( (y - \zeta_k^{2j+1}) \) and \( (y - \zeta_k^{2j+1}) \) is \( \zeta_k^{2j+1} - \zeta_k^{2j+1} \) which together are units in \( \mathbb{C} \) for all \( i \neq j \) such that \( 0 \leq i, j < N \). Because of the unique factorization in \( \mathbb{C}[y] \), it then must also follow that \( \prod_{j=0}^{N-1} (y - \zeta_k^{2j+1}) \) and each of its factors separately divide \( y^N - 1 \). By applying the Chinese Remainder Theorem again, [Für09] then shows that the following theorem is true as a consequence:

**Theorem 3.3-2. (The Half-DFT Isomorphism):** \( R[y]/(y^N + 1) \) is isomorphic to \( \prod_{j=0}^{N-1} R[y]/(y - \zeta_k^{2j+1}) \) and the Half-DFT produces this isomorphism.

An earlier result in [Für09] also shows that the following theorem is true:

**Theorem 3.3-3. (Principal Nth Root of Unity for N a Power of 2):** If \( N \) is a power of 2, and \( \omega^{N/2} = -1 \) in an arbitrary ring, then \( \omega \) is a principal \( N \)th root of unity.

We note that \( R \) contains \( x \) as a principal 2Pth root of unity, according to this theorem, as \( x^P = -1 \) in \( R \). Furthermore, as we have that \( R \) is isomorphic to \( \mathbb{C}^p \), it follows that \( \zeta \in R \) is a principal mth root of unity if and only if it is such a root of unity for every factor \( \mathbb{C}[x]/(x - \eta^{2j+1}) \) of \( R \). Clearly, \( x \mod (x - \eta^{2j+1}) \) simply results in \( \eta^{2j+1} \), and this is a principal 2Pth root of unity in \( \mathbb{C} \). So, we arrive at the following theorem:
Theorem 3.3-4. (The Variable $x$ in $R$): For $P$ that is a power of 2, $x$ is a principal $2P$th root of unity in the ring $R$.

Having $x$ as a root of unity is an excellent property of $R$ as this reduces multiplication by $x$ to cyclic shifts of the polynomial coefficients with a sign change for any wrap around – this is very efficient. Also, there are many principal $2N$th roots of unity in $R$, and we can choose arbitrary corresponding primitive roots of unity in every factor $\mathbb{C}[x]/(x - \eta^{2^j+1})$. Specifically, we want such a root of unity $\rho$ to be in $R$ with the convenient property that $\rho^{N/P} = x$. This is represented as a polynomial in $x$ – we show this by denoting $\rho$ as $\rho(x)$. [Für09] shows that the following theorem is true:

Theorem 3.3-5. (The Norm of $\rho(x)$): The $l_2$ norm of $\rho(x)$ is $\|\rho(x)\|_2 = 1$.

Because of this, it follows that the absolute value of the coefficients of $\rho(x)$ is at most 1. This helps prevent a huge numerical blowup of values when computing the FFT. So we now have a special ring $R = \mathbb{C}[x]/(x^P + 1)$ with the desired properties to perform arithmetic with in the FFT. Now, we can finally move on to discussing the FIMA algorithm properly.

3.4 The Faster Integer Multiplication Algorithm

We will now describe the FIMA as it is presented in [Für09]. We wish to multiply two non-negative integers modulo $2^n + 1$. To do this, the integers are encoded as polynomials, just as in SSA. However, in this case they are polynomials of $R[y]$. Then, we multiply these polynomials using the Fourier Transform. First off, we let $P = \theta(\log n)$ and round it to a power
of 2. The integers, which are in base 2 (binary), are then decomposed into large pieces that are of length $P^2/2$. Then, each of these pieces are further decomposed into smaller pieces of length $P$. Zeroes are padded into the encoding as necessary, and the larger pieces will thus be encoded into polynomials in $R$, which are coefficients of a polynomial in $y$, each of which is a polynomial in $x$. Thus, an integer $a$ turns into a list of polynomials with each element (that is a polynomial) represented as:

$$
\alpha_i = \sum_{j=0}^{P-1} a_{ij}x^j \in R \text{ with } a_{i,P-1} = a_{i,P-2} = \cdots = a_{i,P/2} = 0
$$

The following algorithm gives a function for decomposing a given binary integer $a$ into this form:

**ALGORITHM 3.4-1: Decomposition of $a$**

**Input:** Integer $a$ of length at most $n = NP^2/2$ in binary; integers $N, P$ as powers of 2.

**Output:** $a \in R^N$ (or $a \in R[y]$) encoding $a$.

1. **function** Decompose($a, N, P$)
2. \hspace{1em} For $i = 0$ to $N - 1$ do
3. \hspace{2em} For $j = 0$ to $P/2 - 1$ do
4. \hspace{3em} $a_{ij} = a \mod 2^p$
5. \hspace{3em} $a := \lfloor a/2^p \rfloor$
6. \hspace{2em} For $j = P/2$ to $P - 1$ do
7. \hspace{3em} $a_{ij} = 0$
8. \hspace{1em} $\alpha_i = (a_{i0}, a_{i1}, \ldots, a_{iP-1})^T = a_{i0} + a_{i1}x + \cdots + a_{iP-1}x^{P-1}$
9. \hspace{1em} Return $a = (\alpha_0, \ldots, \alpha_{N-1})$
Of course, we would also like the inverse of such a function to rebuild an integer. This would be used at the end of the algorithm when converting our result into an integer again. Keep in mind that this is all in base 2 binary, so to reverse this process we have to rebuild the integer by starting with zero, then finding the most significant digits and then shifting by multiplying at an appropriate power (in this case, this is clearly $2^p$), and further computing less significant digits. The following algorithm composes a vector in $R^N$ and acts as an inverse operation to Decompose:

**ALGORITHM 3.4-2: Composition of $a$**

**Input:** $a \in R^N$, integers $N, P$ as powers of 2, integer $n$.

**Output:** The integer $a$ encoded by $a$.

1. **function** $Compose(a, N, P, n)$
2. Round all components $a_{ij}$ to the nearest real integer.
3. $a = 0$
4. For $j = P - 1$ down to $P/2$ do
5. $a := a2^p + a_{N-1j}$
6. For $i = N - 1$ down to 1 do
7. For $j = P/2 - 1$ down to 0 do
8. $a := a2^p + a_{ij} + a_{i-1j+p/2}$
9. For $j = P/2 - 1$ down to 0 do
10. $a := a2^p + a_{0j}$
11. **Return** $a \mod (2^n + 1)$

Now that we have a way to represent our integers in the desired form, thanks to Decompose, we now need to worry about the FFT operation. We wish to perform Half-DFTs on each of the integers that we encode using Decompose, via a Half-FFT operation. The Half-FFT requires that we convolve each of the polynomials $a_k$ of a given decomposition $a \in R^N$ with $\zeta^k \in R$, where $\zeta$ is a principal 2Nth root of unity in $R$, with $\zeta^{N/P} = x$. This is referred to as Multiplication in $R$, and is represented with the $*$ symbol. Note that to compute $\zeta^k$, we simply put each element of the vector $\zeta$ to the $k$th power. Then, we FFT the result using a root of unity $\omega = \zeta^2$. Thus, the Half-FFT algorithm is as follows:

**ALGORITHM 3.4-3: The Half Fast Fourier Transform**

**Input:** $a = (a_0, ..., a_{N-1})^T \in R^N$, $\zeta \in R$ such that $\zeta$ is a principal 2Nth root of unity in $R$ with $\zeta^{N/P} = x$, integers $N, P$ as powers of 2.

**Output:** $b \in R^N$; the $N$-point Half-DFT of the input.
True to the traditional FFT, we can also give an algorithm for the inverse Half-FFT. To find the inverse, we must first compute the inverse of the specialized FFT; to do this we feed it an input of \( \omega^{-1} = \xi^{-2} \), dividing each element of the result by \( N \), as per the definition of the inverse FFT, to receive the inverse FFT \( b \). Then, we must convolve each element of \( b, b_k \), of the inverse FFT with powers of \( \xi \) again, but this time negative powers, to undo the transform. Thus, we arrive at the following algorithm for the inverse Half-FFT:

**Algorithm 3.4-4: The Inverse Half Fast Fourier Transform**

**Input:** \( a = (a_0, \ldots, a_{N-1})^T \in \mathbb{R}^N, \xi \in \mathbb{R} \) such that \( \xi \) is a principal \( 2N \)th root of unity in \( \mathbb{R} \), integers \( N, P \) as powers of 2.

**Output:** \( b \in \mathbb{R}^N \); the \( N \)-point inverse Half-DFT of the input.

1. function \( \text{InverseHalfFFT}(a, \xi, N, P) \)
2. \( \omega = \xi^2 \)
3. \( b = \frac{1}{N} \text{FFT}(a, \omega^{-1}, N, P) \)
4. for \( k = 0 \) to \( N - 1 \) do
5. \( b_k = b_k \ast \xi^{-k} \)
6. Return \( b = (b_0, \ldots, b_{N-1})^T \)

Next, we now have to provide an algorithm for the function call FFT in the last two algorithms. This specialized FFT will have two base cases: \( N = 1 \), in which case we simply return the input, just like in the Cooley-Tukey FFT, and \( N = 2 \), in which case the result is a two element vector in which the first element is the sum of the two elements in \( a \) and the second is the first element minus the second element of \( a \). FFTs boil down to this computation for \( N = 2 \), which is quicker than running through the loops and prevents us from having to go through another layer of recursion. We use the indices specified in section 3.2’s definition of Führer’s FFT to iteratively compute the inner FFTs first via recursion, and then iteratively Multiply in \( \mathbb{R} \) (convolve) the specified indices by \( \omega^{jk} \) and iteratively find the FFT of that via recursion. Finally,
at the end of this, we assign the results to our output \( b \). Once the outer FFTs are finished iteratively computing the elements of \( b \), we return the result. Thus, the FFT algorithm is as follows:

**ALGORITHM 3.4-5: Fürer’s Fast Fourier Transform**

Input: \( a = (a_0, \ldots, a_{N-1})^T \in R^N \), \( \omega \in R \) such that \( \zeta \) is a principal \( 2N \)th root of unity in \( R \) with \( \omega^{N/2P} = x \) and \( \omega = x^{2P/N} \) for \( N < 2P \), integers \( N, P \) as powers of 2.

Output: \( b \in R^N \); the \( N \)-point DFT of the input.

1. function \( FFT(a, \omega, N, P) \)
2. if \( N = 1 \) then Return \( a \)
3. if \( N = 2 \) then Return \( b = (a_0 + a_1, a_0 - a_1)^T \)
4. if \( N \leq 2P \) then \( J = 2 \) else \( J = 2P \)
5. \( K = N/J \)
6. for \( k = 0 \) to \( K - 1 \) do
7. for \( k' = 0 \) to \( J - 1 \) do
8. \( c_{k,k'} = a_{(k'K+k)} \)
9. \( c_k = FFT(c_k, \omega^K, J, P) \)
10. for \( j = 0 \) to \( J - 1 \) do
11. for \( k = 0 \) to \( K - 1 \) do
12. \( d_{jk} = c_{k,j} * \omega^{jk} \)
13. \( d_j = FFT(d_j, \omega^j, K, P) \)
14. for \( j' = 0 \) to \( K - 1 \) do
15. \( b_{(j'j+j)} = d_{j,j'} \)
16. Return \( b = (b_0, \ldots, b_{N-1})^T \)

After both integer decompositions have been Half-FFT’d, we perform the point-wise multiplication as in SSA, although this is Multiplication in \( R \). This is a trivial algorithm to find points along our product polynomial thanks to the Convolution Theorem. The algorithm is shown below:

**ALGORITHM 3.4-6: Component-wise Multiplication**

Input: \( a = (a_0, \ldots, a_{N-1})^T, b = (b_0, \ldots, b_{N-1})^T \in R^N \), integers \( N, P \) as powers of 2.

Output: \( c \in R^N \); the components-wise product of the two given inputs.

1. function \( ComponentwiseMultiplication(a, b, N, P) \)
2. for \( j = 0 \) to \( N - 1 \) do
3. \( c_j = a_j * b_j \)
4. Return \( c = (c_0, \ldots, c_{N-1})^T \)
Now we have all the components necessary to carry out the multiplication algorithm in similar fashion to SSA, though with a decomposition step at the beginning and a composition step at the end. This is multiplication on a modular scale, however, meaning it is fit to be called recursively from within the FIMA algorithm. This would happen during the Multiplication in \( R \) operation, which will be discussed at the end of this section. But for now, the familiar algorithm is as follows:

**Algorithm 3.4-7: Modular Integer Multiplication**

**Input:** Integer \( n \), integers \( a \) and \( b \) modulo \( 2^n + 1 \) in binary.

**Output:** The product \( d \) = \( ab \mod (2^n + 1) \)

1. \( \text{function FFT}(a, \omega, N, P) \)
2. \( \text{if } n \leq n_0 \text{ then Return } ab \mod (2^n + 1) \) [Via a simpler multiplication algorithm]
3. \( P := \log n \) rounded to the next power of 2.
4. \( N = 2n/P^2 \)
5. \( a = \text{HalfFFT}(	ext{Decompose}(a, N, P), \zeta, N, P) \)
6. \( b = \text{HalfFFT}(	ext{Decompose}(b, N, P), \zeta, N, P) \)
7. \( c = \text{ComponentwiseMultiplication}(a, b, N, P) \)
8. \( d = \text{InverseHalfFFT}(c, \zeta, N, P) \)
9. \( \text{Return Compose}(d) \)

The initial call to the FIMA algorithm then comes from the below algorithm, which in turn calls the ModularIntegerMultiplication algorithm above. This simply sets up the value for \( n \) first, such that there won’t be any warp around during execution, then begins multiplying.

**Algorithm 3.4-8: Integer Multiplication**

**Input:** Integers \( a \) and \( b \) in binary.

**Output:** The product \( d = ab \).

1. \( \text{function FFT}(a, \omega, N, P) \)
2. \( n = \text{length of } a + \text{length of } b \) rounded to the next power of 2
3. \( \text{Return ModularIntegerMultiplication}(n, a, b) \)

This is the overall structure of the FIMA algorithm as described in [Für09]; the only left to discuss is exactly how Multiplication in \( R \) is supposed to work. In [Für09], this operation is described as follows, without any explicit pseudo code: given \( a \) and \( b \) as input, as well as \( P \) as a
power of 2, if \( b \) is a power of \( x \); i.e. a monomial, then the multiplication is simply done by multiplying each coefficient of \( a \) by the single monomial coefficient and performing a cyclic shift of \( a \) by the power of this monomial and changing the sign of the coefficients on wrap around (better known as a negacyclic shift).

Otherwise, start by writing each of the two polynomials as a sum of a real and an imaginary polynomial, then compute the 4 products (as per the usual way with complex values) of these by multiplying their values at a good power of 2. This pads the space between coefficients enough so that the coefficient of the product polynomial can be recovered from the binary representation of the integer product. But essentially, we recursively multiply these elements by truncating the coefficient values such that the accuracy of the algorithm is not compromised (bounds are given on the required number of digits for this in the paper) and then calling IntegerMultiplication to multiply each of these components as integers encoded as described. This concludes the FIMA algorithm – the pseudo code shown in this section is also provided in one place in Appendix B for convenience.
Chapter 4

Java Implementations of the Algorithms

4.1 Classes and Implementation Details

Now that we have described how these algorithms work in detail and provided pseudo code for both of them, we can begin to discuss the details of actually implementing these algorithms in practice. In both SSA and FIMA, we require an abstraction of complex numbers and operations performed on them. This includes addition / subtraction, multiplication / division, and inversion, just to name a few. In Java, we can provide an object-oriented abstraction of complex numbers that is simply an object containing two doubles that represent $a$ and $b$ in the complex expression $a + bi$. The aforementioned operations on these are simple to implement: to subtract or add, we simply subtract or add the component doubles together; to multiply, each component is distributively multiplied to form 4 products – 2 of which are real numbers and 2 of which are imaginary, and adding the real and imaginary distributions together. Division is a bit harder – we have to multiply by the denominator’s complex conjugate. Thus:

$$\frac{a + bi}{c + di} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{ac + bd + bci + adi}{c^2 + b^2} = \frac{ac + bd}{c^2 + d^2} + \frac{bc + ad}{c^2 + d^2}i$$

As $a, b, c, d$ are doubles, these two components are trivial to compute. We can find a similar formula to compute the reciprocal (multiplicative inverse):

$$(a + bi)^{-1} = \frac{1}{a + bi} = \frac{a - bi}{(a + bi)(a - bi)} = \frac{a - bi}{a^2 + b^2} = \frac{a}{a^2 + b^2} + \frac{b}{a^2 + b^2}i$$

The Java code for the Complex class is shown in Appendix C, which is then used in both SSA and FIMA. There are multiple ways to instantiate a Complex object, for convenience:
explicitly providing both real and imaginary values, just the real (imaginary is set to be 0), or no inputs to represent the zero complex value $0 + 0i$. Additionally, since both algorithms are performed using binary numbers, we also need an abstraction to encapsulate and facilitate data type information in both binary and decimal. The class IntegerOperand does this, forming an IntegerOperand object that contains the string binary representation of the integer in binary, an array of type int that lists the binary digits in the integer starting from the most significant digit, a BigInteger object that represents an arbitrarily large integer value, and the binary digit length for convenience. Large integers such as these are most likely to be formed using automatic random generation or via some kind of algorithm, so IntegerOperand objects are formed from either a string representation of the binary number or by the int array. The IntegerOperand class Java code is shown in Appendix D.

Both the SSA and FIMA algorithms have their own Java classes within which they are implemented, named SchonhageStrassen and FasterIntegerMultiplication, respectively. Both of these classes employ the Complex and IntegerOperand classes. As an object, SchonhageStrassen and FasterIntegerMultiplication accept two IntegerOperand objects representing the two integers to multiply and then multiplies them by calling the SchonhageStrassenMultiplication and FurerMultiplication methods. The two Java implementations are in Appendix E and Appendix F, respectively.

4.2 The Schönhage-Strassen Implementation

The implementation of the SSA is rather straightforward. The int array data type in the IntegerOperand objects provided as integer to multiply already represent polynomial vectors for our integers. We simply perform a Cooley-Tukey FFT on them directly. The SchonhageStrassenMultiplication method essentially encapsulates the Fast Convolution
Algorithm (not as a method call) and performs the bit carries in the SSA provided in section 2.4 (Algorithm 2.4-1). Before applying the FFT algorithm, as the size of the result has to be a power of 2 for the FFT algorithm to work, we keep multiplying int $nfft$ by 2 until it surpasses the sum of the lengths of each integer in binary. The FFT method then pads the int array of the binary integer with zeroes at the end until it is of length $nfft$, then applying the Cooley-Tukey FFT proper as described in section 2.2. In the process, the resulting values are converted into Complex objects and the remainder of the FFT computation uses the Complex class to represent complex numbers.

In addition to this, the CooleyTukeyFFT method computes the Cooley-Tukey FFT slightly differently than described in section 2.2. The odd partition elements are multiplied by the twiddle factors as expected in a loop first. Then, another loop assigns the first half of the list of the transformed elements (indexed by $k$) to be the sum of the $k$th even and odd elements and it assigns the second half of the elements (indexed by $k + \text{half}N$, where $\text{half}N = N/2$) to be the difference between the $k$th even and odd elements that were partitioned. The reasoning behind this is as follows: since the Java implementation uses $\omega = e^{-2\pi i/N}$, we know that $\omega^k = (e^{-2\pi i/N})^k = e^{-2\pi ik/N}$. As such, if we examine the even and odd partition elements and the relationship between the elements of the DFT, we see that any element $\hat{x}_k$ of the resulting DFT follows the relation:

$$
\hat{x}_k = \begin{cases} 
\hat{x}_{e,k} + e^{-2\pi ik/N}\hat{x}_{o,k} & (0 \leq k < N/2) \\
\hat{x}_{e,k-N/2} + e^{-2\pi ik/N}\hat{x}_{o,k-N/2} & (N/2 \leq k < N)
\end{cases}
$$

This is due to the periodicity of the DFT, that states that $\hat{x}_{e,k+N/2} = \hat{x}_{e,k}$ and that $\hat{x}_{o,k+N/2} = \hat{x}_{o,k}$, giving us the above relation. Also, the twiddle factors can be rewritten as follows:

$$
e^{-2\pi i(k+N/2)/N} = e^{-2\pi ik/N - \pi i} = e^{-\pi i}e^{-2\pi ik/N} = -e^{-2\pi ik/N}$$
This conclusion comes directly from the Euler Formula, which gives that $e^{-\pi i} = -1$. This lets us cut the number of twiddle factor computations in half since for $0 \leq k < N/2$:

$$\hat{x}_k = \hat{x}_{e,k} + e^{-2\pi ik/N} \hat{x}_{o,k}$$

$$\hat{x}_{k+N/2} = \hat{x}_{e,k} - e^{-2\pi ik/N} \hat{x}_{o,k}$$

So, employing the computation strategy that is used in the Java implementation reduces the number of twiddle factor computations. Note also, that to compute twiddle factors, we use Euler’s Formula to write $\omega^k = e^{-2\pi ik/N} = \cos(-2\pi k/N) + \sin(-2\pi k/N)i$, so the exponentiation is trivially computed by trigonometric functions instead of fast exponentiation, which could be costly for large $k$. In the case of requiring an inverse FFT, the CooleyTukeyFFT method accepts a boolean value to specify whether or not to multiply the twiddle factor exponent values by $-1$. Before reconstructing the integer at the end of the SSA, we have to round the result to the nearest real integer – this is done by the method RoundToRealInt, which rounds the real component of the complex class to the closest integer.

### 4.3 The Faster Integer Multiplication Implementation

The FIMA implementation largely follows the pseudo code, except that we never recursively call ModularIntegerMultiplication as in [Für09]. The reason for this will be described later in this section, but first we mention some other implementation details. Again, we use arrays of Complex values to represent polynomials as vectors of complex coefficients, making them elements of the ring $R$. These arrays are then placed inside an array of Complex arrays to create an element of $R^N$. We utilize the Complex class for all operations except for the decomposition and composition procedures. For decomposition, the vector representations of polynomials using the Complex class are formed and assigned to the appropriate spot in the array of Complex
arrays. We directly manipulate the binary string to alter the given binary integer as operations such as taking a binary number and computing the modulo with $2^p$ is simply a matter of obtaining the last $P$ binary digits (which is trivial with strings) and dividing by $2^P$ and taking the floor of this is simply obtaining all but the last $P$ binary digits (which is again, trivial with strings). For composition, we instead rebuild the binary string directly with the help of the BigInteger class for operations. Additionally, rounding to the next largest power of 2 is efficiently done through bitwise OR operations and shifting appropriately.

Computing $\zeta$ and powers of it is similar to computing twiddle factors in the SSA implementation, and it is represented as a Complex array – we simply do the same thing we did for twiddle factors for each element. It should be noted that $\zeta$ has the value $e^{i\pi(2k+1)/N}$ at $e^{i\pi(2k+1)/P}$ for $k = 0, \ldots, P - 1$. So for our intents and purpose, $\zeta$ (called zeta in the implementation) is an array of Complex evaluations of $e^{i\pi(2k+1)/N}$ for $k$, starting with $k = 0$, converted using the Euler Formula into complex numbers.

Finally, the reason the Java implementation does not recursively call ModularIntegerMultiplication when performing Multiplication in $R$ is due to the fact that this operation is done by convolving using the SSA implementation functions. We sacrifice some efficiency for simplicity by doing this, but the running time should only be affected by a small constant factor. An optimized version of FIMA would probably employ the recursive strategy described by [För09], or perhaps some of the other strategies discussed in the paper. To convolve the polynomial when performing Multiplication in $R$, we can use the Fast Convolution Algorithm described in section 2.3. In the case of $b$ being a monomial, we perform the negacyclic shift described earlier, to maintain efficiency. This is essentially the modulus operation with polynomials, where the coefficient are also multiplied by the monomial’s coefficient.
5.1 Runtime Comparisons

Table 5.1-1 shows run time measurements in milliseconds of the SSA and the FIMA implementations for increasing sizes of \( n \). The run time trials consisted of generating two random, \( n \) digit binary strings and applying both algorithms to multiply them. The column “FIMA-SSA” denotes the difference in run time between the two algorithms. The two columns under “% of run time” indicate the percent of the total time spent on each algorithm during the trial, so that one can compare the difference in effort between the two algorithms. This table seems to not bode well for FIMA, as the percent of overall effort seems to be more and more dominated by FIMA as the size of \( n \) increases.

<table>
<thead>
<tr>
<th>( n )</th>
<th>SSA</th>
<th>FIMA</th>
<th>FIMA-SSA</th>
<th>SSA %</th>
<th>FIMA %</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
<td>663</td>
<td>1915</td>
<td>1252</td>
<td>25.72</td>
<td>74.28</td>
</tr>
<tr>
<td>100000</td>
<td>1911</td>
<td>2263</td>
<td>352</td>
<td>45.78</td>
<td>54.22</td>
</tr>
<tr>
<td>150000</td>
<td>4670</td>
<td>5215</td>
<td>545</td>
<td>47.24</td>
<td>52.76</td>
</tr>
<tr>
<td>200000</td>
<td>4881</td>
<td>6175</td>
<td>1294</td>
<td>44.15</td>
<td>55.85</td>
</tr>
<tr>
<td>300000</td>
<td>11552</td>
<td>14560</td>
<td>3008</td>
<td>44.24</td>
<td>55.76</td>
</tr>
<tr>
<td>400000</td>
<td>10070</td>
<td>16111</td>
<td>6041</td>
<td>38.46</td>
<td>61.54</td>
</tr>
<tr>
<td>500000</td>
<td>11566</td>
<td>17715</td>
<td>6149</td>
<td>39.50</td>
<td>60.50</td>
</tr>
<tr>
<td>1000000</td>
<td>28287</td>
<td>65255</td>
<td>36968</td>
<td>30.24</td>
<td>69.76</td>
</tr>
</tbody>
</table>
One should note, however, that if the line assigning the IntegerOperand formed from the product binary entries to the output IntegerOperand variable AB were to be commented out, this greatly decreases the run time of the SSA. Reconstructing a decimal form and binary string from the result takes some considerable time – this was omitted from the trials so that the computation time stands out. It stands to reason, then, that the slowness of the FIMA implementation may stem from constructing the resulting product.

Indeed, table 5.1-2. seems to confirm this fact. This table breaks down the time spent on the most crucial methods in the FIMA in the above trials, in both milliseconds and percent of total run time. We see that Compose takes up a huge portion of the computation time. For \( n = 1,000,000 \), it consumes over 50% of the run time. Not only that, but the second most intensive procedure in the algorithm is Decompose, which demonstrates the cost of simply preparing the input for the algorithm. The last column shows what the run time looks like without including Compose and Decompose, and we find that this time is actually shorter than the SSA implementation in almost all cases. Not only that, but these values grow slower in comparison to the SSA.

<table>
<thead>
<tr>
<th>Trial Details</th>
<th>Decompose</th>
<th>Compose</th>
<th>HalfFFT</th>
<th>InverseHalfFFT</th>
<th>CompMult</th>
<th>w/o (De)compose</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>Time (ms)</td>
<td>ms</td>
<td>%</td>
<td>ms</td>
<td>%</td>
<td>ms</td>
</tr>
<tr>
<td>50000</td>
<td>1915</td>
<td>412</td>
<td>21.5</td>
<td>239</td>
<td>12.48</td>
<td>810</td>
</tr>
<tr>
<td>100000</td>
<td>2263</td>
<td>523</td>
<td>23.1</td>
<td>521</td>
<td>23.02</td>
<td>844</td>
</tr>
<tr>
<td>150000</td>
<td>5215</td>
<td>710</td>
<td>13.6</td>
<td>2045</td>
<td>39.21</td>
<td>1632</td>
</tr>
<tr>
<td>200000</td>
<td>6175</td>
<td>1058</td>
<td>17.1</td>
<td>2710</td>
<td>43.89</td>
<td>1582</td>
</tr>
<tr>
<td>300000</td>
<td>14560</td>
<td>1877</td>
<td>12.9</td>
<td>7622</td>
<td>52.35</td>
<td>3276</td>
</tr>
<tr>
<td>400000</td>
<td>16111</td>
<td>3011</td>
<td>18.7</td>
<td>7844</td>
<td>48.69</td>
<td>3280</td>
</tr>
<tr>
<td>500000</td>
<td>17715</td>
<td>4506</td>
<td>25.4</td>
<td>8005</td>
<td>45.19</td>
<td>3290</td>
</tr>
<tr>
<td>1000000</td>
<td>65255</td>
<td>21196</td>
<td>32.5</td>
<td>33133</td>
<td>50.77</td>
<td>7003</td>
</tr>
</tbody>
</table>
It is important to note that the implementation of the SSA is somewhat optimized, but still rather novice compared to what is used in convention (e.g. in cryptography). In reality, an iterative version of the FFT (though somewhat more difficult to implement) would be used to use less recursive calls. Also, the twiddle factors are generally pre-computed – but in these implementations the twiddle factor computation time takes a negligible amount of time (for large \( n \) such as 500,000, less than 1% of the runtime). The base used is only 2; a bigger base would reduce the memory requirements of computation as well, as explained in [GKZ07]. There are also other forms of optimization that can drastically improve the running time.

Notice, however, that twiddle factors are not pre-computed in the FIMA implementation, nor is recursion avoided in the FFT, and neither is a different base used. Thus, we cannot attribute the improved performance of the FIMA (without including Compose and Decompose) to that. In addition to this, the Decompose and Compose operations are very naïve implementations. An optimized implementation would find better (but more complicated or complex) ways of performing Compose. Splitting up the integers in the Decompose stage could also be improved, but Decompose causes less of a run time hit than Compose, so simply providing an optimized Compose operation may make the FIMA implementation much faster in the long run. It is clear from the growth rate that the SSA run time would also grow much more quickly than the rest of the FIMA run-time. So, optimizing Compose and Decompose is key to making the FIMA algorithm better than the SSA algorithm. This is obvious from how much Compose dominates the run time for large values of \( n \), and how the growth of its run time is faster than every other procedure. If we look at table 5.1-3, we find that there is a decreasing correlation between the percentage of the total run time spent in the main computation tasks (MultiplyInR and FFT) throughout the execution of the algorithm, and the size of \( n \). This means that in the long run, the algorithm scales well. Since Compose / Decompose do not use either of these operations and
5.2 Implementation Analysis

The main advantage of the SSA is its simplicity and how short it is. Just over 150 lines of code give impressive results, even when the implementation is rather naïve. The code itself is simple to understand and not very convoluted. In contrast, however, it is without a doubt that the FIMA is much more difficult to implement and more complicated; especially the full implementation proposed in [Für09]. The SSA algorithm presented in this thesis required only a few days of effort to get a working implementation. It took much longer to program the simplified form of the FIMA algorithm, and it is still not working properly as presented here. The code is still under development at the time of submission of this thesis to the Schreyer College; however, the program is complete enough to test run time speed and get accurate predictions for how fast a working implementation would run.

<table>
<thead>
<tr>
<th>Trial Details</th>
<th>Time spent on (ms) / percent of total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Total time (ms)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>50000</td>
<td>1915</td>
</tr>
<tr>
<td>100000</td>
<td>2263</td>
</tr>
<tr>
<td>150000</td>
<td>5215</td>
</tr>
<tr>
<td>200000</td>
<td>6175</td>
</tr>
<tr>
<td>300000</td>
<td>14560</td>
</tr>
<tr>
<td>400000</td>
<td>16111</td>
</tr>
<tr>
<td>500000</td>
<td>17715</td>
</tr>
<tr>
<td>100000</td>
<td>65255</td>
</tr>
</tbody>
</table>
A worthwhile implementation of the FIMA would require an optimized Compose algorithm, which uses a less computation-intensive approach to reconstructing the polynomial from the coefficients obtained at the last step. It is also important to note that the original form of the FIMA algorithm uses recursion at the MultiplyInR step, which allows it to keep a low complexity at very large values of \( n \). As a base case for this recursion, a certain length greater than or equal to 16 is used as a lower bound for when a different algorithm (probably Karatsuba) would finish the multiplication process. In the FIMA implementation presented here, we do not do this and instead use the Convolution Theorem to multiply the polynomials – this is the slower alternative.

Considering that the naïve version of the FIMA (not counting Compose / Decompose) can actually be faster than a somewhat naïve implementation of SSA for relatively small values of \( n \) bodes well. The issue is making the Compose procedure execute faster. Then, FIMA will run faster than SSA and will also grow slower for increasing \( n \). In Compose, the implementation of FIMA uses Java’s BigInteger class for the multiplication and addition operations. There are certainly better ways to multiply by powers of 2, which alone can save some time, for example.

5.3 Conclusions

The FIMA has potential to be faster than SSA in an actual implementation, not just asymptotically. It is very implementation sensitive, however. This is especially true of the Compose operation – optimized implementations should strive to make Compose as fast as possible. Considering that Compose and Decompose take up the most time in the FIMA, it would make sense to focus optimizations there. If Compose and Decompose took less time, it seems that FIMA would be faster at calculating the product of two large integers – especially for really big values of \( n \), possibly in the millions.
It seems that the choice between using SSA and FIMA comes down to the size of $n$ that one plans to use. It is difficult to implement an optimized FIMA – this requires a lot of careful programming and optimization. However, SSA is much shorter and much easier to implement, so unless one plans to utilize the massive speed difference for astronomical values of $n$ in FIMA, the difference in implementation difficulty might be a deal breaker for using it; then SSA would be more preferable.

In conclusion, the FIMA algorithm does indeed possess the potential to be faster than SSA, something which becomes more and more evident for increasing values of $n$. Future attempts at implementations should focus on optimizing the Compose / Decompose operation, as otherwise the speed advantages of FIMA will not be realized. Implementing the algorithm with larger bases would be very difficult as well, but may be necessary in practice to utilize less memory and take advantage of constant-time operations on floating point / integer values. At around $n = 2,000,000$, both algorithms ran into memory usage issues for a standard Java program. Improvements on these issues are important toward practical implementations of FIMA.
Chapter 6

Appendices and References

In this section, on the following pages, are any appendices – such as Java code, pseudo code, and so on – mentioned in the text and a list of references referred to throughout this thesis. Please refer to the table of contents for the page numbers of any specific appendix or for the reference section. Note that any code is written in the NetBeans IDE 7.2.1.

It is also important to understand the FIMA code in appendix F is still under construction, and the algorithm does not accurately compute integer products yet. All methods have been unit tested to have their desired functionality, except for Half-FFT and Inverse-Half-FFT (and thus, by extension the FFT algorithm). A simple example that shows why these do not work properly would be to try computing the Half-FFT of some input and then computing the Inverse-Half-FFT on the result – we should get the same thing we started with (within some small tolerance of error) as these operations are inverses of each other. However, we do not; this issue is currently being investigated.

Contact either Dr. FÜRER or myself to get more up to date / better functioning code as it is developed.
Appendix A

The Schönhage-Strassen Algorithm Pseudo Code

ALGORITHM 2.4-1: The Schönhage-Strassen Algorithm

**Input:** Integers \(x \in \mathbb{Z}_b^n\) and \(y \in \mathbb{Z}_b^m\)

**Output:** Integer \(z = xy \in \mathbb{Z}_b^{2n}\)

1. Perform the Fast Convolution Algorithm on \(x\) and \(y\) to obtain \(\text{convolution} = x \ast y\)
2. \(c := 0\)
3. **For** \(i = 0\) **to** \(2n - 2\) **do**
   4. \(z_i = (\text{convolution}_i + c) \mod b\)
   5. \(c = \lfloor(\text{convolution}_i + c) / b \rfloor\)
6. \(z_{2n-1} = c\)

ALGORITHM 2.3-1: The Fast Convolution Algorithm

**Input:** \(x, y \in \mathbb{C}^n\)

**Output:** \(z = x \ast y \in \mathbb{C}^m\)

1. \(k := \lfloor \log(2n - 1) \rfloor\)
2. \(m := 2^k\)
3. \(x_{\text{pad}} := (x_0, ..., x_{n-1}, 0, ..., 0) \in \mathbb{C}^m\)
4. \(y_{\text{pad}} := (y_0, ..., y_{n-1}, 0, ..., 0) \in \mathbb{C}^m\)
5. \(\hat{x} := \text{CT}_\text{FFT}(k, x_{\text{pad}})\)
6. \(\hat{y} := \text{CT}_\text{FFT}(k, y_{\text{pad}})\)
7. **For** \(i = 0\) **to** \(m - 1\) **do**
   8. \(\hat{z}_i = \hat{x}_i \hat{y}_i\)
9. \(z = \text{CT}_\text{IFFT}(k, \hat{z})\)

ALGORITHM 2.2-1: The Cooley-Tukey Fast Fourier Transform

**Input:** \(n = 2^k\) and \(x \in \mathbb{C}^n\)

**Output:** \(\hat{x} \in \mathbb{C}^n\), the FFT of \(x\)

1. **function** CT-FFT(k, x)
2. **if** \(k = 0\) **then** \(\hat{x} = x\)
3. Partition $x$ into $x_e, x_o \in \mathbb{C}^n$
4. $\hat{x}_e = \text{CT-FFT}(k - 1, x_e)$ [by recursion]
5. $\hat{x}_o = \text{CT-FFT}(k - 1, x_o)$ [by recursion]
6. $\omega := e^{2\pi i/n}$
7. for $i = 0$ to $n - 1$ do
8. $\hat{x}_i = \hat{x}_{e,i \mod n/2} + \omega^i \hat{x}_{o,i \mod n/2}$

ALGORITHM 2.2-2: The Inverse Cooley-Tukey Fast Fourier Transform

Input: $n = 2^k$ and $x \in \mathbb{C}^n$
Output: $\tilde{x} \in \mathbb{C}^n$, the IFFT of $x$

1. function CT_IFFT($k, x$)
2. $\tilde{x} := CT\_FFT(k, x, \text{true})$
3. for $i = 0$ to $n - 1$ do
4. $\tilde{x}_i := \tilde{x}_i / n$
Appendix B

Fürer's Faster Integer Multiplication Algorithm Pseudo Code

ALGORITHM 3.4-8: Integer Multiplication
Input: Integers \( a \) and \( b \) in binary.
Output: The product \( d = ab \).
1. function \( FFT(a, \omega, N, P) \)
2. \( n = \text{length of } a + \text{length of } b \text{ rounded to the next power of 2} \)
3. Return \( \text{ModularIntegerMultiplication}(n, a, b) \)

ALGORITHM 3.4-7: Modular Integer Multiplication
Input: Integer \( n \), integers \( a \) and \( b \) modulo \( 2^n + 1 \) in binary.
Output: The product \( d = ab \mod (2^n + 1) \)
1. function \( FFT(a, \omega, N, P) \)
2. if \( n \leq n0 \) then Return \( ab \mod (2^n + 1) \) [Via a simpler multiplication algorithm]
3. \( P := \lfloor \lg n \rfloor \text{ rounded to the next power of 2.} \)
4. \( N = 2n/P^2 \)
5. \( a = \text{HalfFFT(Decompose}(a, N, P), \zeta, N, P) \)
6. \( b = \text{HalfFFT(Decompose}(b, N, P), \zeta, N, P) \)
7. \( c = \text{ComponentwiseMultiplication}(a, b, N, P) \)
8. \( d = \text{InverseHalfFFT}(c, \zeta, N, P) \)
9. Return \( \text{Compose}(d) \)

ALGORITHM 3.4-1: Decomposition of \( a \)
Input: Integer \( a \) of length at most \( n = NP^2/2 \) in binary; integers \( N, P \) as powers of 2.
Output: \( a \in R^N \) (or \( a \in R[y] \)) encoding \( a \).
1. function \( \text{Decompose}(a, N, P) \)
2. For \( i = 0 \) to \( N - 1 \) do
3. For \( j = 0 \) to \( P/2 - 1 \) do
4. \( a_{ij} = a \mod 2^p \)
5. \( a := \lfloor a/2^p \rfloor \)
6. For \( j = P/2 \) to \( P - 1 \) do
ALGORITHM 3.4-2: Composition of \( a \)

**Input:** \( a \in \mathbb{R}^N \), integers \( N, P \) as powers of 2, integer \( n \).

**Output:** The integer \( a \) encoded by \( a \).

1. function \( \text{Compose}(a, N, P, n) \)
2. Round all components \( a_{ij} \) to the nearest real integer.
3. \( a = 0 \)
4. For \( j = P - 1 \) down to \( P/2 \) do
5. \( a := a2^P + a_{N-1j} \)
6. For \( i = N - 1 \) down to 1 do
7. For \( j = P/2 - 1 \) down to 0 do
8. \( a := a2^P + a_{ij} + a_{i-1j+p/2} \)
9. For \( j = P/2 - 1 \) down to 0 do
10. \( a := a2^P + a_{0j} \)
11. Return \( a \mod (2^n + 1) \)

ALGORITHM 3.4-3: The Half Fast Fourier Transform

**Input:** \( a = (a_0, ..., a_{N-1})^T \in \mathbb{R}^N \), \( \zeta \in \mathbb{R} \) such that \( \zeta \) is a principal \( 2N \)th root of unity in \( \mathbb{R} \) with \( \zeta^{N/P} = x \), integers \( N, P \) as powers of 2.

**Output:** \( b \in \mathbb{R}^N \); the \( N \)-point Half-DFT of the input.

1. function \( \text{HalfFFT}(a, \zeta, N, P) \)
2. For \( k = 0 \) to \( N - 1 \) do
3. \( \omega = \zeta^2 \)
4. \( \omega = \zeta^2 \)
5. Return \( FFT(a, \omega, N, P) \)

ALGORITHM 3.4-4: The Inverse Half Fast Fourier Transform

**Input:** \( a = (a_0, ..., a_{N-1})^T \in \mathbb{R}^N \), \( \zeta \in \mathbb{R} \) such that \( \zeta \) is a principal \( 2N \)th root of unity in \( \mathbb{R} \), integers \( N, P \) as powers of 2.

**Output:** \( b \in \mathbb{R}^N \); the \( N \)-point inverse Half-DFT of the input.

1. function \( \text{InverseHalfFFT}(a, \zeta, N, P) \)
2. \( \omega = \zeta^2 \)
3. \[ b = \frac{1}{N} \text{FFT}(a, \omega^{-1}, N, P) \]

4. For \( k = 0 \) to \( N - 1 \) do
5. \[ b_k = b_k \cdot \zeta^{-k} \]
6. Return \( b = (b_0, \ldots, b_{N-1})^T \)

**ALGORITHM 3.4-5: Fürer’s Fast Fourier Transform**

**Input:** \( a = (a_0, \ldots, a_{N-1})^T \in R^N, \omega \in R \) such that \( \zeta \) is a principal 2\( N \)th root of unity in \( R \) with \( \omega^{N/2P} = x \) and \( \omega = x^{2P/N} \) for \( N < 2P \), integers \( N, P \) as powers of 2.

**Output:** \( b \in R^N \); the \( N \)-point DFT of the input.

1. function \( \text{FFT}(a, \omega, N, P) \)
2. if \( N = 1 \) then Return \( a \)
3. if \( N = 2 \) then Return \( b = (a_0 + a_1, a_0 - a_1)^T \)
4. if \( N \leq 2P \) then \( J = 2 \) else \( J = 2P \)
5. \( K = N/J \)
6. for \( k = 0 \) to \( K - 1 \) do
7. for \( k' = 0 \) to \( J - 1 \) do
8. \( c_{k,k'} = a_{(k'K+k)} \)
9. \( c_k = \text{FFT}(c_k, \omega^K, J, P) \)
10. for \( j = 0 \) to \( J - 1 \) do
11. for \( k = 0 \) to \( K - 1 \) do
12. \( d_{j,k} = c_{k,j} \cdot \omega^{jk} \)
13. \( d_j = \text{FFT}(d_j, \omega^J, K, P) \)
14. for \( j' = 0 \) to \( K - 1 \) do
15. \( b_{(j'J+j)} = d_{j,j'} \)
16. Return \( b = (b_0, \ldots, b_{N-1})^T \)

**ALGORITHM 3.4-6: Component-wise Multiplication**

**Input:** \( a = (a_0, \ldots, a_{N-1})^T, b = (b_0, \ldots, b_{N-1})^T \in R^N \), integers \( N, P \) as powers of 2.

**Output:** \( c \in R^N \); the components-wise product of the two given inputs.

1. function \( \text{ComponentwiseMultiplication}(a, b, N, P) \)
2. for \( j = 0 \) to \( N - 1 \) do
3. \( c_j = a_j \cdot b_j \)
4. Return \( c = (c_0, \ldots, c_{N-1})^T \)
package MiscellaneousClasses;

public class Complex {
    public double a, b;

    public Complex() {
        a = 0;
        b = 0;
    }

    public Complex(double real) {
        a = real;
        b = 0;
    }

    public Complex(double real, double imaginary) {
        a = real;
        b = imaginary;
    }

    public static Complex add(Complex x, Complex y) {
        return new Complex(x.a + y.a, x.b + y.b);
    }

    public static Complex subtract(Complex x, Complex y) {
        return new Complex(x.a - y.a, x.b - y.b);
    }

    public static Complex multiply(Complex x, Complex y) {
        return new Complex(x.a * y.a - x.b * y.b, x.a * y.b + x.b * y.a);
    }

    public static Complex divide(Complex x, Complex y) {
        double denominator = y.a * y.a + y.b * y.b;
        if (denominator != 0) {
            return new Complex((x.a * y.a + x.b * y.b) / denominator,
                               (x.b * y.a - x.a * y.b) / denominator);
        } else {
            System.err.println("Error: division of x by y = 0 + 0i does not exist!");
        }
        return x;
    }

    public static Complex complexConjugate(Complex x) {
        return new Complex(x.a, -1 * x.b);
    }
}
public static Complex inverse(Complex x) {
    Complex conjugate = complexConjugate(x);
    double denominator = x.a * x.a + x.b * x.b;
    
    if(denominator != 0) {
        return(new Complex(conjugate.a / denominator, conjugate.b / denominator));
    }
    else {
        System.err.println("Error: inverse of x = 0 + 0i does not exist!");
    }
    return x;
}

global static double magnitude(Complex x) {
    return(Math.pow(x.a * x.a + x.b * x.b, 1 / 2.0));
}
package FastIntegerMultiplication;
import java.math.BigInteger;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class IntegerOperand
{
    public String stringBinaryValue;
    public int[] arrayBinaryValue;
    public BigInteger integerValue;
    public int length;

    public IntegerOperand(String value)
    {
        value = value.replace("-", "");
        if(isBinary(value))
        {
            stringBinaryValue = value;
            arrayBinaryValue = stringToIntArray(value);
        }
        else
        {
            stringBinaryValue = "0";
            arrayBinaryValue = new int[1];
            arrayBinaryValue[0] = 0;
        }
        integerValue = new BigInteger(stringBinaryValue, 2);
        length = stringBinaryValue.length();
    }
    public IntegerOperand(int value[])
    {
        String strValue = "";
        for(int i = 0; i < value.length; i++)
        {
            strValue = strValue.concat(String.valueOf(value[i]));
        }
        stringBinaryValue = strValue;
        arrayBinaryValue = value;
        length = value.length;
        integerValue = new BigInteger(stringBinaryValue, 2);
    }
    public boolean isBinary(String value)
    {
        Matcher matcher = Pattern.compile("^\s*[01]+\s*$").matcher(value);
        if(matcher.find() && matcher.start() == 0 && matcher.end() == value.length())
        {
            return true;
        }
        return false;
    }
    public int[] stringToIntArray(String value)
    {  
    
    }
}
char[] string = value.toCharArray();
int[] array = new int[string.length];

for(int i = 0; i < array.length; i++)
{
    if(string[i] == '1')
    {
        array[i] = 1;
    }
    else
    {
        array[i] = 0;
    }
}

return array;
package Schonhage_Strassen_Algorithm;

import FastIntegerMultiplication.IntegerOperand;
import MiscellaneousClasses.Complex;

public class SchonhageStrassen
{
    public IntegerOperand x, y, xy;
    public int n;
    public static double PI = 3.1415926535897932384626;

    public SchonhageStrassen(IntegerOperand a, IntegerOperand b)
    {
        if(a.length == b.length)
        {
            x = a;
            y = b;
            n = a.length;
        }
        else
        {
            System.err.println("Error: Lengths of integer operands do not match.");
        }
    }

    public void SchonhageStrassenMultiplication()
    {
        int nfft = 2, c = 0, temp = 0;
        Complex[] fftX, fftY, fftZ;
        int[] Z, z;

        while(nfft < x.length + y.length)
        {
            nfft *= 2;
        }

        fftX = FFT(x.arrayBinaryValue, x.length, nfft);
        fftY = FFT(y.arrayBinaryValue, y.length, nfft);
        fftZ = ComponentwiseMultiplication(fftX, fftY, nfft);
        Z = RoundToRealInt(InverseFFT(fftZ, nfft, x.length + y.length - 1));
        z = new int[Z.length];

        for(int i = 0; i < z.length - 1; i++)
        {
            c += Z[i];
            z[z.length - i - 1] = c % 2;
            c = c / 2;
        }

        z[0] = c % 2;
        xy = new IntegerOperand(z);
    }

    public static Complex[] FFT(int[] x, int N, int nfft)
    {
        Complex[] list = new Complex[nfft];
        for(int i = 0; i < N; i++)
        {
            list[i] = new Complex((double)x[N - i - 1]);
        }
    }
}
for(int i = N; i < nfft; i++)
{
    list[i] = new Complex();
}
return(CooleyTukeyFFT(list, nfft, false));
}

public static Complex[] CooleyTukeyFFT(Complex[] x, int N, boolean inversePower)
{
    int halfN = N / 2;
    Complex[] xEven, xOdd, fftX;
    if(N == 1)
    {
        return x;
    }
    xEven = new Complex[halfN];
    xOdd = new Complex[halfN];
    fftX = new Complex[N];
    for(int k = 0; k < halfN; k++)
    {
        xEven[k] = x[2 * k];
        xOdd[k] = x[2 * k + 1];
    }
    xEven = CooleyTukeyFFT(xEven, halfN, inversePower);
    xOdd = CooleyTukeyFFT(xOdd, halfN, inversePower);
    for(int k = 0; k < halfN; k++)
    {
        xOdd[k] = Complex.multiply(ComputeOmegaPower(k, N, inversePower), xOdd[k]);
    }
    for(int k = 0; k < halfN; k++)
    {
        fftX[k] = Complex.add(xEven[k], xOdd[k]);
        fftX[k + halfN] = Complex.subtract(xEven[k], xOdd[k]);
    }
    return fftX;
}

public static Complex[] ComponentwiseMultiplication(Complex[] A, Complex[] B, int nfft)
{
    Complex[] C = new Complex[nfft];
    for(int i = 0; i < nfft; i++)
    {
        C[i] = Complex.multiply(A[i], B[i]);
    }
    return C;
}

public static Complex[] InverseFFT(Complex[] x, int nfft, int length)
{
    Complex inverseFactor = new Complex(1 / (double)nfft);
    Complex[] ifft = CooleyTukeyFFT(x, nfft, true);
    for(int i = 0; i < length; i++)
    {
        ifft[i] = Complex.multiply(inverseFactor, ifft[i]);
    }
    return ifft;
return ifft;
}

public static int[] RoundToRealInt(Complex[] values)
{
    int[] rounded = new int[values.length];
    for(int i = 0; i < values.length; i++)
    {
        rounded[i] = (int)Math.floor(0.5 + Math.abs(values[i].a));
    }
    return rounded;
}

public static Complex ComputeOmegaPower(int k, int N, boolean inverseOmega)
{
    double x = -2 * PI * k / N;
    if(inverseOmega)
    {
        x *= -1;
    }
    return(new Complex(Math.cos(x), Math.sin(x)));
}
Furer’s FasterIntegerMultiplication Java Implementation Class

```java
package Furers_Algorithm;
import FastIntegerMultiplication.IntegerOperand;
import MiscellaneousClasses.Complex;
import Schonhage_Strassen_Algorithm.SchonhageStrassen;
import java.math.BigInteger;

public class FasterIntegerMultiplication {
    public IntegerOperand A, B, AB;
    public int n0;

    public FasterIntegerMultiplication(IntegerOperand Avalue, IntegerOperand Bvalue, int n) {
        A = Avalue;
        B = Bvalue;
        if(n0 >= 16) {
            n0 = n;
        } else {
            n0 = 16;
        }
    }

    public void FurersMultiplicationAlgorithm() {
        Complex[][] a, b, c, d;
        Complex[] zeta;
        int n = round(A.length + B.length), P, N;
        String result;
        if(n <= n0) {
            SchonhageStrassen ss = new SchonhageStrassen(A, B);
            ss.SchonhageStrassenMultiplication();
            AB = ss.xy;
            return;
        }
        P = (int)round(lg(n));
        N = (int)(2 * n / (P * P));
        zeta = ComputeZeta(N, P);
        a = HalfFFT(Decompose(A, N, P), zeta, N, P);
        b = HalfFFT(Decompose(B, N, P), zeta, N, P);
        c = ComponentwiseMultiplication(a, b, N, P);
        d = InverseHalfFFT(c, zeta, N, P);
        result = Compose(d, N, P).toString(2);
        AB = new IntegerOperand(result);
    }
}
```

```java
public Complex[][] Decompose(IntegerOperand a, int N, int P) {
    Complex coefficients[][] = new Complex[N][P];
    String aBinary = A.stringBinaryValue;
    if(a.length > N * P * P / 2)
```
{ return null; }

for(int i = 0; i < N; i++)
{
    for(int j = 0; j < P / 2; j++)
    {
        coefficients[i][j] = new Complex((new IntegerOperand( aBinary.substring((aBinary.length() - P < 0) ? 0 : aBinary.length() - P))).intValue.intValue());
        aBinary = aBinary.substring(0, (aBinary.length() - P < 0) ? 0 : aBinary.length() - P);
    }
    for(int j = P / 2; j < P; j++)
    {
        coefficients[i][j] = new Complex(0);
    }
}

return coefficients;

public BigInteger Compose(Complex[][] a, int N, int P)
{
    BigInteger PshiftValue, result = BigInteger.ZERO;
    String Pshift = "1";
    int[][] aRound = new int[a.length][a[1].length];
    for(int i = 0; i < P; i++)
    {
        Pshift += "0";
    }
    for(int i = 0; i < a.length; i++)
    {
        for(int j = 0; j < a[i].length; j++)
        {
            aRound[i][j] = (int)Math.floor(0.5 + Math.abs(a[i][j].a));
        }
    }
    PshiftValue = new BigInteger(Pshift, 2);
    for(int j = P - 1; j >= P / 2; j--)
    {
        result = result.multiply(PshiftValue).add(new BigInteger(String.valueOf(aRound[N - 1][j])));
    }
    for(int i = N - 1; i >= 1; i--)
    {
        for(int j = P / 2 - 1; j >= 0; j--)
        {
            result = result.multiply(PshiftValue).add(new BigInteger(String.valueOf(aRound[i][j]))).add(new BigInteger(String.valueOf(aRound[i - 1][j + P / 2])));
        }
    }
    for(int j = P / 2 - 1; j >= 0; j--)
    {
        result = result.multiply(PshiftValue).add(new BigInteger(String.valueOf(aRound[0][j])));
    }
    return(result.mod((new BigInteger("2")).pow(n).add(BigInteger.ONE)));
public Complex[][] HalfFFT(Complex[][] a, Complex[] zeta, int N, int P)
{
    Complex[][] aWeighted = new Complex[N][];
    Complex[] omega = ExponentiateVector(zeta, 2);
    for(int k = 0; k < N; k++)
    {
        aWeighted[k] = MultiplyInR(a[k], ExponentiateVector(zeta, k), P, true);
    }
    return(FFT(aWeighted, omega, N, P));
}

public Complex[][] InverseHalfFFT(Complex[][] a, Complex[] zeta, int N, int P)
{
    Complex[][] aIHFFT;
    Complex[] omega = ExponentiateVector(zeta, 2);
    aIHFFT = ScaleInRN(FFT(a, InversePowerOfVector(omega), N, P),
        new Complex(1 / (double)N));
    for(int k = 0; k < N; k++)
    {
        aIHFFT[k] = MultiplyInR(aIHFFT[k],
            ExponentiateVector(InversePowerOfVector(zeta), k), P, true);
    }
    return aIHFFT;
}

public Complex[][] FFT(Complex[][] a, Complex[] omega, int N, int P)
{
    Complex[][] b = new Complex[N][P];
    Complex[][][] c, d;
    int J, K;
    if(N == 1)
    {
        return a;
    }
    else if(N == 2)
    {
        b[0] = AddPolynomials(a[0], a[1]);
        b[1] = SubtractPolynomials(a[0], a[1]);
        return b;
    }
    if(N <= 2 * P)
    {
        J = 2;
    }
    else
    {
        J = 2 * P;
    }
    K = N / J;
    c = new Complex[K][J][];
    d = new Complex[J][K][];
    for(int k = 0; k < K; k++)
    {
        for(int kprime = 0; kprime < J; kprime++)
        {
            c[k][kprime] = a[kprime * K + k];
        }
        }
\[
c[k] = \text{FFT}(c[k], \text{ExponentiateVector}(\omega, K), J, P);
\]
\[
\text{for}(\text{int } j = 0; j < J; j++)
\{
    \text{for}(\text{int } k = 0; k < K; k++)
    \{
        d[j][k] = \text{MultiplyInR}(c[k][j],
            \text{ExponentiateVector}(\omega, j \times k), P, \text{true});
    \}
    d[j] = \text{FFT}(d[j], \text{ExponentiateVector}(\omega, J), K, P);
    \text{for}(\text{int } jprime = 0; jprime < K; jprime++)
    \{
        b[jprime \times J + j] = d[j][jprime];
    \}
\}
\]
return b;
\]
\]
public Complex[][] ComponentwiseMultiplication(Complex [][] a,
    Complex[][] b, int N, int P)
{
    Complex[][] c = new Complex[N][];
    \text{for}(\text{int } j = 0; j < N; j++)
    \{
        c[j] = \text{MultiplyInR}(a[j], b[j], P, \text{false});
    \}
return c;
\]
public Complex[] MultiplyInR(Complex[] alpha, Complex[] beta, int P,
    boolean oneSided)
{
    boolean shift = true;
    int index = -1;
    \text{for}(\text{int } i = 0; i < \text{beta.length}; i++)
    \{
        if(\text{beta}[i].a != 0 || \text{beta}[i].b != 0)
        \{
            if(index == -1)
            \{
                index = i;
            \}
            else if(index != -1)
            \{
                shift = false;
                break;
            \}
        \}
    \}
if(shift)
    \{
        Complex[] product = new Complex[alpha.length + index],
            result = new Complex[P];
        \text{for}(\text{int } i = 0; i < \text{alpha.length}; i++)
        \{
            product[i + index] = Complex.multiply(alpha[i], beta[index]);
        \}
for(int i = P; i < product.length; i++)
    product[i - P] = Complex.multiply(new Complex(-1.0),
          product[P]);
}

for(int i = 0; i < P; i++)
    result[i] = product[i];

return result;
}

else

    Complex[] fftAlpha, fftBeta, fftProduct;
    int nfft = 2;

    while(nfft < alpha.length + beta.length)
    {
        nfft *= 2;
    }

    fftAlpha = SchonhageStrassen.
        CooleyTukeyFFT(PadWithZeros(alpha, nfft), nfft, false);

    if(!oneSided)
    {
        fftBeta = SchonhageStrassen.
            CooleyTukeyFFT(PadWithZeros(beta, nfft), nfft, false);
    }
    else
    {
        fftBeta = PadWithZeros(beta, nfft);
    }

    fftProduct = SchonhageStrassen.
        ComponentwiseMultiplication(fftAlpha, fftBeta, nfft);

    return(PolynomialMod(SchonhageStrassen.InverseFFT(fftProduct, nfft,
    nfft - 1), P));
}

public Complex[] ComputeZeta(int N, int P)
{
    Complex[] zeta = new Complex[P];
    double x;

    for(int k = 0; k < zeta.length; k++)
    {
        x = SchonhageStrassen.PI * (2 * k + 1) / N;
        zeta[k] = new Complex(Math.cos(x), Math.sin(x));
    }

    return zeta;
}

public Complex[] ExponentiateVector(Complex[] vector, int power)
{
    Complex[] result = new Complex[vector.length];

    for(int i = 0; i < vector.length; i++)
    {
        result[i] = Complex.fastExponentiation(vector[i], power);
    }

    return result;
}
public Complex[] InversePowerOfVector(Complex[] vector) {
    Complex[] result = new Complex[vector.length];
    for(int i = 0; i < vector.length; i++) {
        result[i] = Complex.inverse(vector[i]);
    }
    return result;
}

public Complex[][] ScaleInRN(Complex[][] toScale, Complex scalar) {
    Complex[][] scaled = new Complex[toScale.length][toScale[1].length];
    for(int i = 0; i < toScale.length; i++) {
        for(int j = 0; j < toScale[i].length; j++) {
            scaled[i][j] = Complex.multiply(toScale[i][j], scalar);
        }
    }
    return scaled;
}

public Complex[] PolynomialMod(Complex[] polynomial, int P) {
    if(polynomial.length >= P + 1) {
        Complex[] p1 = new Complex[P],
        p2 = new Complex[polynomial.length - P];
        for(int i = 0; i < p1.length; i++) {
            p1[i] = polynomial[i];
        }
        for(int i = 0; i < p2.length; i++) {
            if(P + i < polynomial.length) {
                p2[i] = Complex.multiply(new Complex(-1.0),
                polynomial[P + i]);
            } else {
                p2[i] = new Complex();
            }
        }
        return(AddPolynomials(p1, p2));
    } else {
        return(PadWithZeros(polynomial, P));
    }
}

public Complex[] AddPolynomials(Complex[] alpha, Complex[] beta) {
    int n = Math.min(alpha.length, beta.length);
    Complex[] gamma = new Complex[n];
    for(int i = 0; i < n; i++) {
        gamma[i] = Complex.add(alpha[i], beta[i]);
    }
}
public Complex[] SubtractPolynomials(Complex[] alpha, Complex[] beta) {
    int n = Math.min(alpha.length, beta.length);
    Complex[] gamma = new Complex[n];
    for(int i = 0; i < n; i++)
    {
        gamma[i] = Complex.subtract(alpha[i], beta[i]);
    }
    return gamma;
}

public Complex[] PadWithZeros(Complex[] vector, int length) {
    Complex[] padded = new Complex[length];
    for(int i = 0; i < vector.length; i++)
    {
        padded[i] = vector[i];
    }
    for(int i = vector.length; i < length; i++)
    {
        padded[i] = new Complex();
    }
    return padded;
}

public int round(int n) {
    n = --n;
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;
    n |= n >> 16;
    n++;
    return n;
}

public int lg(int n) {
    long i = 1;
    int counter = 0;
    while(i < n)
    {
        i *= 2;
        counter++;
    }
    return(counter - 1);
}
REFERENCES


ACADEMIC VITA

Peter Sutor Jr.
2 Mason Street, Mount Pocono, PA 18344
pws5164@psu.edu

Education

Bachelor of Science in Computer Science and Mathematics (Systems Analysis) from Pennsylvania State University, expected May 2014. Schreyer Honors College Scholar.

Honors and Awards

Awards:
- President’s Freshmen Award (2011)
- Dean’s List for every undergraduate semester (2010-2014)

Scholarships:
- John Woloschuk Memorial Scholarship (2011)
- Tau Beta Pi Record Scholarship (2013 / 2014)

Grants:
- Internal Commonwealth Grant (2010)
- NSA POISSON Grant (2014)

Association Memberships/Activities

Honors Society Memberships:
- Penn State Chapter of Honors Engineering Society Tau Beta Pi (2012)
- Penn State Chapter of National Honors Society of Leadership and Success (2012)
- National Honors Society Phi Kappa Phi (2012)

Activities:
- Participation in the Geometrics and Physics (GAP XII) conference in Sanya, China (2014).

Professional Experience

Penn State University Park, State College PA. August 2012 – Present

Mathematics, Computer Science, and Economics Peer Tutor
- Peer tutor for all mathematics classes up to and including Differential Equations.
- Peer tutor for introductory Computer Science courses.
- Peer tutor for introductory Economics courses.

Chemistry Department Software Developer for New Interactive Website
- Ongoing project oriented around designing a sophisticated scientific calculator tailored for Chemistry students, programmed in Java (project supervisor: Dr. Pshemak Maslak).
- Worked on designing layout, fixing and improving numerical algorithms and creating new ones.
Keystone Automotive, Inc., Exeter PA  
*Software Developer Intern (returning intern)* June 2013 – August 2013

- Designed and developed a system to simplify and handle generating CSV file reports dynamically for customers using SQL and Visual Basic using .NET.
- Developed an application to adjust settings for this system to easily customize how the CSV reports look and are created.

*Web Developer Intern*

May 2012 – August 2012

- Created online utilities for administrating purposes in Visual Basic using .NET.
- Developed and programmed an efficient Smart Search algorithm for the search engine on the company’s distributing webpage using only SQL.

Penn State Worthington-Scranton, Dunmore PA.  

- Tutored Students in all Mathematics classes up to Three Dimensional Calculus.
- Tutored Students in general Physics classes.

Publications and Papers

Smarkusky, Debra L., Toman, Sharon A., Sutor, Peter, and Hunt, Christopher.  