Integer Factorization and RSA Encryption

Noah Zemel

February 2016

1 Introduction

Cryptography has been used for thousands of years as a means for securing a communications channel. Throughout history, all encryption algorithms utilized a private key, essentially a cipher that would allow people to both encrypt and decrypt messages.

While the complexity of these private key algorithms grew significantly during the early 20th century because of the World Wars, significant change in cryptography didn't come until the 1970s with the invention of public key cryptography. Up until this point, all encryption and decryption required that both parties involved knew a secret key. This led to a weakness in the communications systems that could be infiltrated, sometimes by brute force, but usually by more complicated backtracking algorithms.

The most popular example of breaking a cipher was when the researchers at Bletchley Park (most notably Alan Turing) were able to crack some of the private keys being transmitted each morning by the Nazi forces,. This enabled the Allied forces to decipher some important messages, revealing key enemy positions.

With the advent of computers and the power to do calculations much faster than humans can, there was a clear need for a more sophisticated encryption system—one that couldn't be cracked by intercepting or solving the private key.

In **public key cryptography**, there are two keys: a private key and a public key. The public key is used to encrypt the message, while the private key is needed to decrypt it. Only the decryptor knows the private key. If the encrypted message (called the ciphertext) is intercepted, it will be impossible to decipher without the private key.

This paper will provide an overview of the RSA cryptosystem as well as some of the necessary mathematics behind it. Furthermore, I will detail some of the factorization algorithms used to crack RSA, and will describe my work implementing these algorithms in Java with the intent of identifying which are the most efficient for factoring various sized integers.

2 Mathematical Background

In understanding how some of these public key cryptosystems function, it is necessary to give a brief overview of some pertinent ideas from number theory involving modular arithmetic.

2.1 Euclidean Algorithm

The **Euclidean algorithm** provides a fast and efficient way to solve for an inverse in modular arithmetic through the division algorithm. It is easier to simply show how this works through example. Suppose one wants to find $17^{-1} \pmod{60}$. A series of equivalences with divisors and remainders can be made, similar to long division:

$$\begin{aligned} 60 &= 3 \cdot 17 + 9 \\ 17 &= 1 \cdot 9 + 8 \\ 9 &= 1 \cdot 8 + 1 \\ 8 &= 8 \cdot 1 + 0 \end{aligned}$$

Through the division algorithm, it is deduced that the $\gcd(17, 60)$ is the last non-zero remainder (in this case, 1). Now plugging that back in to get:

$$1 \equiv 9-8 \equiv 9-(17-9\cdot 1) \equiv 2\cdot 9-17 \equiv 2(60-3\cdot 17)-17 \equiv 2\cdot 60-7\cdot 17 \pmod{60}$$

Since $60 \equiv 0 \pmod{60}$, there is left $(-7)17 \equiv 1 \pmod{60}$ which gives $17^{-1} \equiv -7 \equiv 53 \pmod{60}$, and thus the algorithm works. This algorithm is essential for algorithms discussed in Section 4, as it can be seen that the time complexity of the Euclidean algorithm is $\mathcal{O}(\ln(n))$, or linear in logarithmic time.

2.2 Fast Exponentiation

Fast exponentiation is another algorithm that helps improve efficiency when working with modular exponents. In some of our cryptosystems, it is necessary to be able to solve $M^e \mod N$, where e and N are very large numbers, usually over 150 digits. It would be incredibly computationally heavy to solve this normally, so fast exponentiation is utilized to speed this up.

Essentially, a table of the binary powers of $M \mod N$ is constructed. Let's take an arbitrary $M = 61, e = 21, N = 79$ to illustrate this. The table is as follows:

61^1	$\equiv$	61			(mod 79)
61^2	$\equiv$	3721	$\equiv$	8	(mod 79)
61^4	$\equiv$	8^2	$\equiv$	64	(mod 79)
61^8	$\equiv$	64^2	$\equiv$	67	(mod 79)
61^{16}	$\equiv$	67^2	$\equiv$	65	(mod 79)

To now find an equivalence for $M^e \pmod{N}$, substitute $61^{21} \equiv 61^{(16+4+1)} \equiv 61^{16} \cdot 61^4 \cdot 61^1 \equiv 65 \cdot 64 \cdot 61 \equiv 12 \pmod{79}$. This algorithm reduces the number of multiplications down to around $\mathcal{O}(\ln(n))$, which is polynomial in logarithmic time. This is a great improvement over the operation that would otherwise be exponential in logarithmic time (e.g. $\mathcal{O}(e^{\log n})$).

2.3 Euler's Theorem

Euler's Theorem, published by Euler in 1763, states that given relatively prime integers a and n, $a^{\phi(n)} \equiv 1 \pmod{n}$. The function $\phi(n)$ is called Euler's totient function and counts all the numbers less than or equal to n that are relatively prime to n. It is important to recognize that given a prime p, $\phi(p) = p - 1$. Similarly, given a product of two primes $N = p \cdot q$, $\phi(N) = \phi(p \cdot q) = \phi(p)\phi(q) = (p-1)(q-1)$. This theorem is at the core of RSA cryptography.

3 Overview of Modern Cryptosystems

3.1 Discrete Logarithm Problem

In the mid 1970's, Diffie and Hellman published their key exchange system which relied on the **discrete logarithm problem** (DLP) for security. In fact, all modern key-exchange cryptography can be traced back to this problem. The DLP reads: given a primitive root g of a finite field F_p, and nonzero h, find exponent x that satisfies:

$$g^x \equiv h \pmod{p}$$

Both the Diffie-Hellman key exchange and the ElGamal public key cryptosystem, a famous asymmetric cryptosystem, intrinsically rely on this equation for their security.

One of the security issues that plagues this type of symmetric key exchange is an attack called a **man-in-the-middle attack**. For practical purposes moving forward, this paper will refer to the encryptor as Bob, the decrypter as Alice, and the eavesdropper as Eve. Here's an analogy to explain this: say Eve does not know how to play chess, but claims she can play two grandmasters simultaneously. She then takes the move one of the masters uses against the other one, and vice versa (4). In a man-in-the-middle attack on a Diffie-Hellman key exchange, Eve intercepts both the keys being exchanged (A and B) and replaces them with her secretly computed value of E as $E = g^e$ and passes on her E value to both Alice and Bob instead of their transmitted A and B. In the end, both Alice and Bob end up with the same private key, so they have the appearance that their communications are secure, as they are both able to encrypt and decrypt messages, however the eavesdropper Eve is able to decipher every message being sent.

3.2 RSA Public Key Cryptosystem

Today, RSA public key encryption is used in favor of the Diffie-Hellman an ElGamal systems.

An interesting tidbit of history surrounding RSA is that, while it was invented in 1977 by the three mathematicians Rivest, Shamir, and Adleman, an earlier equivalent system was invented by the English mathematician Cocks in 1973, but was kept classified by the English government until the 90's, thus the publically available RSA encryption became the norm for those not privy to top-secret government cryptosystems. In fact, the export of such systems was considered treasonous to many governments, which made research and development of cryptography through the late 20th century (and even today) a very controlled operation, with most of the research being done under a top-secret label.

A large difference between the RSA system and Diffie-Hellman and ElGamal is the underlying equation that is being solved. The latter two systems relied on solving the congruence $a^x \equiv b \pmod{p}$ for x. In RSA encryption, the system solves for the integer d given $x^e \equiv c \pmod{N}$, where $e \cdot d \equiv 1 \pmod{\phi(N)}$. Another core difference here is that N is a composite number approximately of size 2^{1000} and is the product of two very large, privately known prime numbers (i.e. $N = p \cdot q$). The security of RSA revolves upon the difficulty of solving $e \cdot d \equiv 1 \pmod{\phi(N)}$ without the knowledge of N's prime factorization.

The RSA algorithm works through the following steps: First, Alice needs to create her public key through choosing two prime numbers p and q that are at least 150 digits. She then calculates her public key $N = p \cdot q$ and decides upon a public encryption exponent e that satisfies $\gcd(e, (p-1)(q-1)) = 1$. She then makes public her public key (N, e). For Bob to encrypt a message to Alice, he converts his message into an integer M between 1 and N, and then computes, utilizing fast exponentiation, the ciphertext $c \equiv M^e \pmod{N}$ using Alice's public key. For Alice to decrypt the message, she first calculates d (using the Euclidean algorithm) that satisfies $ed \equiv 1 \pmod{(p-1)(q-1)}$. Now Alice is ready to decrypt the message through the fact that $M = c^d \pmod{N}$.

3.2.1 Why RSA works

At first, it looks like magic that Alice is able to convert the ciphertext back into the original message while no eavesdroppers can. Let's look at the arithmetic involved: After Bob sends Alice the ciphertext $c \equiv M^e \pmod{N}$, she computes:

$$c^d \equiv (M^e)^d \equiv M^{de} \equiv M^{1+k(\phi(N))} \equiv M \cdot (M^{\phi(N)})^k \equiv M \cdot 1^k \pmod{N}$$

Two important substitutions in the above equivalence occur when, first, she is able to substitute $de \equiv 1 + k(\phi(N))$ for some $k \in Z$, as this was established during the original calculation for d ($de \equiv 1 \mod \phi(N)$). The second substitution involves using Euler's theorem to generate the equivalence $M^{\phi(N)} \equiv 1 \pmod{N}$, given $\gcd(M, N) = 1$. It is assumed that M and N are relatively prime, as $N = p \cdot q$ where both p and q are quite large, thus the chance of M sharing a common factor with N is approximately $\frac{1}{p}$, which is negligible. Modern RSA systems will have a catch mechanism that ensures the integerized message M does not share a common factor with N. Therefore, this substitution holds.

The security of this system involves the fact that there is no known method to calculate $\phi(N)$ without the knowledge of N's prime factorization.

4 Factorization Algorithms

My work focuses on cracking RSA public key cryptography through factoring N into p and q through a variety of factorization algorithms that I have spent much time implementing in Java.

In particular, I compare the efficiencies of these algorithms to one another and determining which algorithms are more efficient given a variable size modular field (N). My implementation involves a unique combination of number theory, linear algebra, and computer science. Specifically, I have looked at the quadratic sieve method, the number field sieve, the Pollard rho algorithm, Shanks' square form factorization, and as a control group, trial division.

4.1 Trial Division

Trial division is the easiest of the algorithms to understand, however is the least efficient, as its run-time is exponential (as opposed to the faster, sub-exponential run-times of the other algorithms). It can be thought of as brute force factorization, as it is the most basic algorithm. The algorithm simply iterates through a list of primes $p_1, p_2, \ldots, p_k$ and tries to divide N by p_i. If it divides without a remainder, the algorithm has successfully factored N. If not, i is incremented and the algorithm repeats. The run time of trial division is exponential in logarithmic time by recognizing that it is necessary to try all primes up to $\sqrt{N}$ in the worst-case scenario. In Big-O notation, given $N^{\frac{1}{2}}$ tries, the time complexity is: $\mathcal{O}(e^{\frac{1}{2}\cdot\ln(N)})$. All other algorithms implemented should run faster than trial division, especially for $N \approx 2^{1000}$.

4.2 The Quadratic Sieve

The majority of research done on factorization algorithms has been with the quadratic sieve, a rather complicated and incredibly useful algorithm that functions in sub-exponential time, on the order of $\mathcal{O}(e^{\sqrt{\ln(n)\ln\ln(n)}})$.

4.2.1 Difference of Squares Factorization

At the heart of the quadratic sieve is difference of squares factorization (recall from algebra that 899 is easily factored via: $899 = 900 - 1 = 30^2 - 1^2 = (30-1)(30+1) = (29)(31)$). This same logic is used to factor N into some $a^2 - b^2$, which also works the same under modular arithmetic. The algorithm starts with some $b = \lceil\sqrt{N}\rceil$, and then increases b by 1 and reduces $b^2 \pmod{N}$ until it equals a perfect square a^2. It is now possible to factor N through the congruence $a^2 \equiv b^2 \pmod{N}$. This is much easier starting with $b \approx \sqrt{N}$ as

$b^2 \pmod N$ will be a small number. This idea leads to the three-step process known as the quadratic sieve.

The first step is called **relation building**, wherein a large table of integers $a_1, a_2, \ldots, a_k$ is created such that the quantity $c_i \equiv a_i^2 \pmod N$ factors as a product of small primes.

The next step is called the **linear algebra elimination step**, where we take a product of various $c_{i1}, c_{i2}, \ldots, c_{is}$ generated in the relation building so that all of the corresponding c_i's factors are exponentiated to an even power, such that $c_{i1}, c_{i2}, \ldots, c_{is} = b^2$ is a perfect square. In practice, this is computed through solving for the basis vectors in a massive, sparse matrix of the prime powers modulo 2.

The final step is **GCD computation**. In this step, the algorithm takes some $a = a_{i1}, a_{i2}, \ldots, a_{is}$ and computes the GCD $d = \gcd(N, a - b)$. It is relatively likely that d is a nontrivial factor of N. If it is not, different relations must be chosen that satisfy the even exponent condition. If, after exhausting all possible combinations of relations, there is still no non-trivial gcd, then it is necessary to increase the size of the sieve and try again. A more in-depth example of implementation will follow at the end of this section.

4.2.2 Smooth Numbers

The quadratic sieve takes the ideas from factorization via difference of squares a step further by using a property related to **smooth numbers**. An integer n is B-smooth if all of its prime factors are less than or equal to B (1). For example, 2, 3, 4, 6, 8, and 9 are 3-smooth while 5, 7, 10 and 11 are not.

In evaluating the efficiency of the quadratic sieve, it is important to understand the function $\psi(X, B)$, which counts the number of B-smooth numbers between 1 and X. Its growth as X and B grow to very large numbers "needs to increase in just the right way" (1).

An important theorem detailing this growth is called the **Canfield, Erdős, and Pomerance Theorem**, which reads: fix a number $0 < \epsilon < 1$ and let X and B increase together while satisfying: $(\ln X)^\epsilon < \ln B < (\ln X)^{1-\epsilon}$. In moving forward, we denote $u = \frac{\ln X}{\ln B}$. Now there is the unique property that the number of B-smooth numbers less than X satisfies $\psi(X, B) = X \cdot u^{-u(1+o(1))}$ where $o(1)$ is little-o notation meaning as X tends to infinity, $o(1)$ goes to 0. In looking at which values of B and X are needed for the algorithm to work, we arrive at the equation: $L(X) = e^{\sqrt{(\ln X)(\ln \ln X)}}$. By looking at the growth table of $L(X)$ with respect to X (1), it is seen that $L(X)$ follows a pattern of sub-exponential growth. It turns out that in order to have a reasonable chance of factoring N, it is necessary to choose $B \approx L(N)^{1/\sqrt{2}}$ (1). In order for the linear elimination step to be successful, it follows that there must be at least $\pi(B)$ B-smooth numbers, where $\pi(B)$ is the number of primes up to B. This much seems obvious, and for all practical applications of the quadratic sieve where it is factoring integers on the order of 1024 bits (approximately 300 digits), this will usually be met by the nature of taking many relations.

Finding B-smooth numbers $\pmod N$ is crucial to be able to solve the table of relations, as it needs numbers with small factors in order to piece two or more of them together successfully to have the factors of their product all be raised to an even exponent. If the factors are large, this task becomes very difficult. A core question is how do we appropriately find numbers $a > \sqrt(N)$ such that $a^2 \pmod N$ is B-smooth? The quadratic sieve will sieve a list of different a values by prime powers up to B, and from that it is possible to construct relations from every entry that has been sieved. There is some pertinent logic that surrounds how to reduce these numbers that will be addressed in the example.

4.2.3 Quadratic Residue

In order to speed up the efficiency of the quadratic sieve algorithm, the implementation utilizes a facet of modular arithmetic known as quadratic residues. If an integer a cannot be equal to a square modulo prime p, then the quadratic sieve should know to skip that integer in the sieve. Given these conditions, a is referred to as a **quadratic nonresidue** modulo p. If a is a square modulo p (there exists $c \in Z$ such that $c^2 \equiv a \pmod p$), a is referred to as a **quadratic residue** modulo p. Now the question presents itself of how do we tell if a is a quadradic residue (QR) or a quadratic nonresidue (NR)? In order to solve this, one must implement what is known as **quadratic reciprocity** using Legendre symbols and a pertinent theorem. The **Legendre symbol** of a modulo p is denoted as $\left(\frac{a}{p}\right)$, and via various operations, will reduce to either 1 if a is a QR, -1 if a is a NR, or 0 if p divides a. The following three operations allow one to reduce the Legendre symbol to determine if a is a QR or NR modulo p:

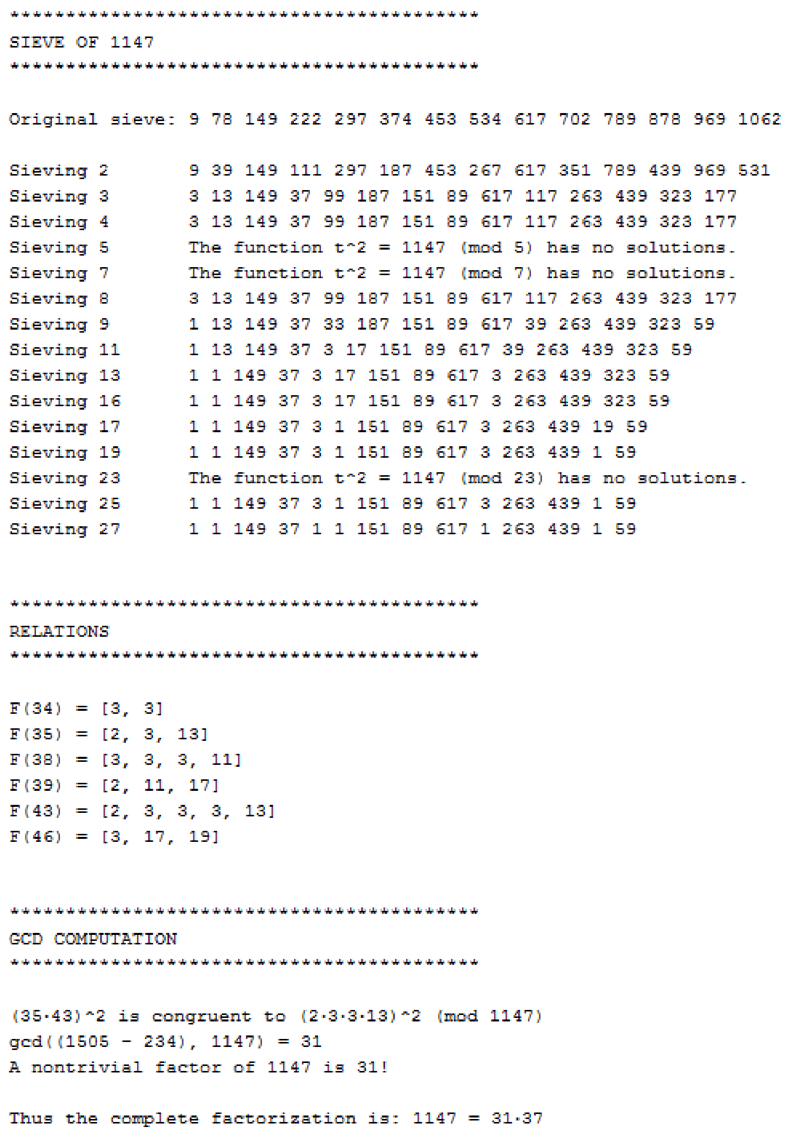
1) $\left(\frac{-1}{p}\right) = \begin{cases} 1 & \text{if } p \equiv 1 \pmod 4 \\ -1 & \text{if } p \equiv 3 \pmod 4 \end{cases}$

2) $\left(\frac{2}{p}\right) = \begin{cases} 1 & \text{if } p \equiv 1 \text{ or } 7 \pmod 8 \\ -1 & \text{if } p \equiv 3 \text{ or } 5 \pmod 8 \end{cases}$

3) $\left(\frac{p}{q}\right) = \begin{cases} \left(\frac{q}{p}\right) & \text{if } p \equiv 1 \pmod 4 \text{ or } q \equiv 1 \pmod 4 \\ -\left(\frac{q}{p}\right) & \text{if } p \equiv 3 \pmod 4 \text{ and } q \equiv 3 \pmod 4 \end{cases}$

4.2.4 Example

To demonstrate the quadratic sieve in action, I have included a screenshot of the output from the Java program I wrote implementing the quadratic sieve using a very small value of N in order to demonstrate certain aspects of the algorithm, as shown in Figure 1. For this example, I chose $N = 1147 = 31 \cdot 37$, and for this sieve to work, it is necessary to calculate smooth numbers up to $B = 27$, which means the sieve will factor prime powers up until 27. While the program normally approximates B by using the formula $B \approx L(N)^{\frac{1}{\sqrt{2}}}$ as described in section 4.2.2, these approximations tend to fall apart when working with small numbers, thus trial and error was required to find a successful value of B.



```
*******************************************
SIEVE OF 1147
*******************************************

Original sieve: 9 78 149 222 297 374 453 534 617 702 789 878 969 1062

Sieving 2       9 39 149 111 297 187 453 267 617 351 789 439 969 531
Sieving 3       3 13 149 37 99 187 151 89 617 117 263 439 323 177
Sieving 4       3 13 149 37 99 187 151 89 617 117 263 439 323 177
Sieving 5       The function t^2 = 1147 (mod 5) has no solutions.
Sieving 7       The function t^2 = 1147 (mod 7) has no solutions.
Sieving 8       3 13 149 37 99 187 151 89 617 117 263 439 323 177
Sieving 9       1 13 149 37 33 187 151 89 617 39 263 439 323 59
Sieving 11      1 13 149 37 3 17 151 89 617 39 263 439 323 59
Sieving 13      1 1 149 37 3 17 151 89 617 3 263 439 323 59
Sieving 16      1 1 149 37 3 17 151 89 617 3 263 439 323 59
Sieving 17      1 1 149 37 3 1 151 89 617 3 263 439 19 59
Sieving 19      1 1 149 37 3 1 151 89 617 3 263 439 1 59
Sieving 23      The function t^2 = 1147 (mod 23) has no solutions.
Sieving 25      1 1 149 37 3 1 151 89 617 3 263 439 1 59
Sieving 27      1 1 149 37 1 1 151 89 617 1 263 439 1 59


*******************************************
RELATIONS
*******************************************

F(34) = [3, 3]
F(35) = [2, 3, 13]
F(38) = [3, 3, 3, 11]
F(39) = [2, 11, 17]
F(43) = [2, 3, 3, 3, 13]
F(46) = [3, 17, 19]


*******************************************
GCD COMPUTATION
*******************************************

(35·43)^2 is congruent to (2·3·3·13)^2 (mod 1147)
gcd((1505 - 234), 1147) = 31
A nontrivial factor of 1147 is 31!

Thus the complete factorization is: 1147 = 31·37
```

Figure 1: The Quadratic Sieve

In the walkthrough of the output, the first item of interest is the original sieve values. These numbers are genereated through the function $F(T) \equiv T^2 \pmod{N}$. The first value of T is the square root ceiling of N, denoted $\lceil\sqrt{N}\rceil$, as $\lceil\sqrt{N}\rceil^2 \pmod{N}$ will be close to 0, making the prime factors of the value $\pmod{N}$ small, and thus easier to find relations. Starting with $F(34) \equiv 9$, and then $F(35) \equiv 78$, and so forth, the sieve is capped once $T^2 > 2N$.

This can be seen in the output that the last number in the sieve is $F(47)$ as $48^2 = 2307 > 2 * 1147$. When all the original values have been gathered, the sieve can begin.

As the algorithm sieves through the list, it will divide certain numbers in the list by increasing prime powers p up to B, starting with 2. For each iteration of the sieve (each time p is increased), the algorithm determines which entries satisfy $t^2 \equiv 1147 \pmod{p}$ and divides those entries by the prime factor of p. The sieve starts with the first prime power, 2, and every entry in the sieve that satisfies $t^2 \equiv 1147 \equiv 1 \pmod{2}$ is divied by 2, which is every other entry starting with $F(35)$. Next, the sieve moves on to sieving 3. There are two solutions to the equivalence $t^2 \equiv 1147 \equiv 1 \pmod{3}$, where $t = 1$ and $t = 2$, thus 3 is sieved twice from the list. The first reduction begins with $F(34)$ as $34 \equiv 1 \pmod{3}$ and the second begins with $F(35)$ as $35 \equiv 2 \pmod{3}$. For each of these reductions, every third entry in the sieve is divided by 3 since the algorithm is operating currently in modulo 3. The sieve continues down the list of prime powers. The program runs a calculation before each iteration of the sieve to verify that there is at least one solution to $t^2 \equiv 1147 \pmod{p}$ by seeing if 1147 is a quadratic residue modulo p. In the output, 5, 7, and 23 all have no solutions, and thus are not sieved. Once the end of the sieve is reached ($p \geq B$), we look at every entry that has been reduced to 1 and build relations from the prime factors of the corresponding $F(T)$ values (the original entries in the sieve).

The next step the program calculates is determining which of the primes factors in the relations table can be combined to create a perfect square (i.e. which relations can be combined so that the combined prime factors all appear to an even exponent?). In my program, this is done by solving the nullspace of a matrix with all the prime factors for basis vectors in order to generate solutions, which is not shown in the output. In this example, there is only one solution which is obtained by multiplying $F(35)$ and $F(43)$, as shown in the last part of the output ($F(35) \cdot F(43) \equiv (2 \cdot 3 \cdot 3 \cdot 13)^2 \pmod{1147}$).

The final step of the algorithm computes the GCD of the difference of squares $(35 \cdot 43 - 2 \cdot 3 \cdot 3 \cdot 13)$ and N, and if that result is non-trivial, then it is a factor of N. If it is trivial, the algorithm returns to the solution vectors of the null-space matrix and chooses another one. If still no non-trivial GCD calculations present themselves, it is necessary to increase B and re-do the entire sieve. Here, the implementation has computed a non-trivial factor of 1147 as 31, and the other factor is easily obtained by dividing 1147 by 31.

4.2.5 The Number Field Sieve

The general number field sieve (GNFS) is the fastest factorization algorithm for integers larger than 2^{350}, or approximately 100 decimal digits. It follows a similar method to the quadratic sieve, but uses an algebraic number field to reduce the time complexity to $\mathcal{O}\left(e^{(c+o(1)) \cdot \ln(N)^{\frac{1}{3}} \ln \ln(N)^{\frac{2}{3}}}\right)$ where $c \approx 1.526$ (9). When looking at numbers of the form $a^e \pm b$, the number field is determined by a, e, and b (6). Implementing and fully understanding the number field sieve is graduate-level work that requires significant programming expertise and incredibly expensive computers to achieve the type of efficiency first described by Lenstra, *et al.* in their 1989 paper.(9; 11)

4.3 Pollard's ρ Method

Pollard's ρ factorization algorithm is much less complicated than the quadratic sieve; however, it does not function as quickly for factoring realistic sizes of N in an actual RSA system. There are further shortcomings that make the ρ and the similar $\rho - 1$ and $\rho + 1$ algorithms somewhat obsolete for cracking RSA systems. The algorithm is so named because of the shape the "search" makes in looping through integers (it looks like the greek letter ρ). The method works as follows: given some N to be factored as in the other algorithms, it starts by setting $a = 2$. Next, it loops $j = 2, 3, 4, \ldots$ until it reaches a specified bound. Inside the loop, the algorithm sets $a = a^j \pmod{N}$ and compute $d = \gcd(a - 1, N)$. If $1 < d < N$, it has successfully factored N with nontrivial factor d. If $d = 1$ or $d = N$, it has not, and it increments j and tries again. This method in many ways is similar to brute force factorization, but with a significant efficiency advantage. Pollard's ρ method works best when one of the factors of N is significantly smaller than the other, and becomes very slow when both factors of N are of similar size. For this reason, this algorithm is not the preferred method of factoring N values used in actual RSA implementations. Furthermore, the time complexity of this algorithm is exponential in logarithmic time (approximately $\mathcal{O}\left(e^{\frac{1}{4}\ln(N)}\right)$), as opposed to the faster subexponential methods. Finally, modern RSA systems defend against ρ factorization by making sure that neither $(p - 1)$ nor $(q - 1)$ factor into small primes, making this factorization algorithm somewhat irrelevant.(1; 5; 6)

4.4 Shanks' Square Form Factorization

Shanks' Square Form Factorizations (SquFoF) utilizes the difference of squares factorization described in section 4.1.1. This is the fastest algorithm for factoring integers between 2^{30} and 2^{60} (12). The algorithm works through computing quadratic forms of different integers under the given modulus N to be factored. For a more detailed description of the algorithm, Gower and Wagstaff's paper (12) is very comprehensive. SquFoF operates in exponential time complexity under logarithmic operations, given as $\mathcal{O}\left(e^{\frac{1}{4}\ln(N)}\right)$.

5 Analysis of Algorithms

The initial design of this project involved my successful implementation and comparison of these various factorization algorithms. In practice, my computer science skills have hindered the efficiency of the more complex algorithms (quadratic sieve and SquFoF), while favoring the simpler ones (Pollard ρ, trial division). As such, I have resorted to publically available code in order to successfully compare these algorithms. Tilman Neumann has compiled a program that implements all of these algorithms, among others (7). He writes of his implementation that trial division is the most efficient for $N < 2^{27}$, 32-bit SquFoF for $2^{28} < N < 2^{42}$, 64-bit SquFoF for $2^{43} < N < 2^{68}$, and the self-initializing variant of the quadratic sieve for $2^{69} < N$. "[The quadratic sieve] is currently the algorithm of choice for "hard" composites with about 20 to 120 digits. [...] For larger numbers, the number field sieve moves to the front, but this "viability border" between the quadratic sieve and the number field sieve is not very well defined, and shifts as new computer architectures come on line and when new variations of the underlying methods are developed." (10) The general number field sieve is the most efficient algorithm given $2^{350} < N$ (1).

Next, we will look at the time complexity ($\mathcal{O}$-values) of each of the examined algorithms (1; 8):

Algorithm	Time Complexity (Big-$\mathcal{O}$)	Type
Trial Division	$\mathcal{O}\left(e^{\frac{1}{2}\ln(N)}\right)$	exponential
Pollard ρ	$\mathcal{O}\left(e^{\frac{1}{4}\ln(N)}\right)$	exponential
SquFoF	$\mathcal{O}\left(e^{\frac{1}{4}\ln(N)}\right)$	exponential
Quadratic Sieve	$\mathcal{O}\left(e^{(1+o(1))\cdot\sqrt{\ln(N)\ln\ln(N)}}\right)$	subexponential
Number Field Sieve	$\mathcal{O}\left(e^{\left(\sqrt[3]{\frac{64}{9}}+o(1)\right)\cdot\ln(N)^{\frac{1}{3}}\ln\ln(N)^{\frac{2}{3}}}\right)$	subexponential

More research should be done in analyzing the borders of efficiency between these various algorithms, ideally working with very experienced computer programmers and powerful, reliable machines.

6 Conclusion

Modern RSA cryptosystems are, generally speaking, impossible to crack given the current hardware and factorization algorithms used. As larger and larger numbers are factored, RSA systems will adapt to use even larger keys. Currently, private keys are usually 2048 bits, or 617 decimal digits. The largest numbers factored so far have been on the order of 200 decimal digits. There was even a competition called the RSA Factoring Challenge that awarded money for the successful factorization of larger and larger numbers from a set of massive composite integers labelled the RSA numbers that promoted the research of new factorization algorithms and the fine-tuning of existing ones.

So far, we have yet to attain factorization in linear time under logarithmic operations, with the best algorithms operating in subexponential time. Given the pace of new technological breakthroughs, it will come, sooner or later, that our archaic computational systems based on transistors will limit the seemingly boundless increase in efficiency in these algorithms. One such linear time algorithm has been discovered for quantum computers (computers that function not with bits, but with quantum bits (qubits)—physical particles that can have "spin"). This algorithm was created in 1994 and is called Shor's Algorithm. Given the infancy of quantum computing, the largest number this algorithm has thus been able to factor has only been 56153. But all great things start in small places, and with many more years of technological discovery to come, it is possible that this algorithm can usurp the general number field sieve as the fastest factorization algorithm.

References

[1] Jeffery Hoffstein, Jill Pipher, and Joseph H. Silverman. *An Introduction to Mathematical Cryptography.* Springer, New York, New York, 2014.

[2] Carl Pomerance. *A Tale of Two Sieves.* American Mathematical Society, 43(12): 1473–1485, 1996.

[3] Willie K. Harrison. *Physical-Layer Security: Practical Aspects of Channel Coding and Cryptography* (Doctoral Dissertation). School of Electrical and Computer Engineering, Georgia Institute of Technology, 2012.

[4] *Man-in-the-middle attack.* Wikipedia. `http://en.wikipedia.org/wiki/Man-in-the-middle_attack`.

[5] J. M. Pollard. *Theorems on factorization and primality testing.* Mathematical Proceedings of the Cambridge Philosophical Society, 76: 521-528, 1974.

[6] Richard P. Brent. *Primality Testing and Integer Factorisation.* `http://maths-people.anu.edu.au/~brent/pd/rpb120.pdf`.

[7] Tilman Neumann. *PSIQS.* `http://www.tilman-neumann.de/`.

[8] Kostas Bimpikis and Ragesh Jaiswal. *Modern Factoring Algorithms.* University of California, San Diego.

[9] A.K. Lenstra, H.W. Lenstra, Jr., M.S. Manasse, J.M. Pollard. *The Number Field Sieve.* December, 1989.

[10] Carl Pomerance. *Smooth Numbers and the Quadratic Sieve.* Algorithmic Number Theory, Vol. 44, 2008.

[11] Carl Pomerance. The Number Field Sieve. Proceedings of Symposia in Applied Mathematics, Vol. 48, 1994.

[12] Jason E. Gower and Samuel S. Wagstaff, Jr. *Square Form Factorization.* American Mathematical Society, 1997.