# THE QUADRATIC SIEVE INTEGER FACTORIZATION ALGORITHM: THEORY AND IMPLEMENTATION

By

JUSTIN JEREMY VILORIA SANTOS

A thesis submitted in partial fulfillment of
the honors graduation requirements for the degree of
Bachelor of Science in Computer Science

UNIVERSITY OF OKLAHOMA
School of Computer Science

MAY 2020

Abstract

The inherent difficulty of the integer factorization problem is what many popular cryptographic algorithms, such as RSA, are built on top of. In this work, we discuss the process and theory behind one popular integer factorization algorithm — the quadratic sieve. Additionally, we provide a detailed Python program of this algorithm and a brief reflection on the implementation process.

# TABLE OF CONTENTS

# Chapter One

# Introduction

Integer factorization is a puzzle that has confounded mathematicians since the dawn of time. Currently, there does not exist a known polynomial-time algorithm that is able to solve this problem using a classical computer. The integer factorization record, as of May 2020, is the RSA number RSA-250, which has 250 decimal digits (829 bits). The algorithm used in this record was the number field sieve, the successor to the quadratic sieve in terms of efficiency. Still, this computation took approximately 2700 core-years using Intel Xeon Gold 6130 CPUs (Zimmerman, 2020). It is worth mentioning that a polynomial-time algorithm has been formulated for use with a quantum computer, namely Shor's algorithm (Shor, 1994); however, such a quantum computer able to use this algorithm effectively has not been built.

That being said, we will first discuss two trivial and relatively inefficient factorization methods — trial division and Fermat's method — before discussing the quadratic sieve. The main purpose in doing so is to introduce the common logic found in most factoring algorithms, and to bring focus to the increase in efficiency brought forth by quadratic sieve.

## 1.1 Trial Division

The most primitive and laborious factorization algorithm is trial division. First described in *Liber Abaci* written by Fibonacci in 1202, this method tests whether the number to be

factored (henceforth denoted by $n$) is divisible by any smaller number. It does so by dividing $n$ by every integer in the interval $[2, \sqrt{n}]$. Dividing further than $\sqrt{n}$ is unnecessary because the largest possible factor of $n$ is $\sqrt{n}$ in the case where $n$ is a perfect square of a prime number.

Here is an example of this method with $n = 8051$:

$$8051/2 = 4025.5$$

$$8051/3 = 2683.67$$

$$8051/4 = 2012.75$$

$$8051/5 = 1610.2$$

$$\vdots$$

This division continues until we reach $8051/83 = 97$, where we can finally conclude that the factorization of $8051 = 83 \cdot\ 97$.

An improvement to this method is done by only considering prime numbers as possible factor candidates. This is because there is no point in testing if $n$ is divisible by 4 if it was previously determined to be not divisible by 2. In general, if $n$ is deemed not divisible by an integer $m$, it is not necessary to test trial division for any other multiple of $m$. In doing so, we are able to drastically reduce the number of trial divisions required to reach the solution. Still, trial division has a worst-case time complexity of

$$O\left(\frac{n\sqrt{n}}{(\ln n)^2}\right) \text{ (O'Neill, 2009).}$$

## 1.2 Fermat's Method

This method, named after mathematician Pierre de Fermat, stems from the observation that $n$ can be represented as the difference of two squares $a$ and $b$, giving us the two factors $(a+b)$ and $(a-b)$:

$$n = a^2 - b^2 = (a+b)(a-b).$$

We proceed by doing an example of this method using $n = 8051$. In order to find the difference of two squares, we must incrementally try various integer values of $a$ starting at $a = \lceil \sqrt{n} \rceil$ in hopes that $a^2 - n = b^2$. Luckily for us, we do not have to stray too far from the starting value; we find that

$$\begin{aligned} n &= 8051 \\ &= 8100 - 49 \\ &= 90^2 - 7^2 \\ &= (90 - 7)(90 + 7) \\ &= 83 \cdot 97. \end{aligned}$$

However, often times we are not as fortunate in finding $a$. A drawback to this method is that it is only powerful when $a$ and $b$ are close to $\sqrt{n}$. In fact, when $a$ and $b$ are not close to $\sqrt{n}$, this method can worse than trial division.

Although still inefficient, the logic behind this method forms the basis of the quadratic sieve algorithm. The main difference between the two is that instead of attempting to find a square $b^2$ from the sequence $a^2 - n$, we try to find a subsquence of $a^2 - n$ such that the product the the elements of that subsequence is square.

# Chapter Two

# Quadratic Sieve

## 2.1 Preliminary Definitions

Some preliminary definitions must to be established before we begin the discussion. Define the Kraitchik function as

$$Q(x) = (x + \lfloor \sqrt{n} \rfloor)^2 - n.$$

Define the sieving interval as $[-M, M]$; an optimal value for $M$ will be discussed later. Using the Kraitchik function and the sieving interval, we can compute Kraitchik's sequence

$$K = (Q(x_1), Q(x_2),\ ..., Q(x_i))$$

where the values of $x_i \in \mathbb{Z}$ come from the sieving interval. Lastly, define a smoothness bound $B \in \mathbb{N}$. A numerical value for $B$ will be discussed later.

## 2.2 The General Process

The main goal of the quadratic sieve is nearly identical to the goal of Fermat's method. In order to factor $n$, quadratic sieve attempts to find integers $a$ and $b$ such that

$$a^2 \equiv b^2 \pmod{\text{n}}.$$

If the algorithm is successful in doing so, we are able to compute the two factors of $n$ by calculating $\gcd(x+y,\ n)$ and $\gcd(x-y,\ n)$ as shown in the following logic:

$$\begin{aligned} & a^2 \equiv b^2 \pmod{\text{n}} \\ \implies & a^2 - b^2 \equiv 0 \pmod{n} \\ \implies & (a-b)(a+b) \equiv 0 \pmod{n}. \end{aligned}$$

The core difference between these two methods is the path taken to find $a$ and $b$.

In quadratic sieve, We want to find a subsequence $K'$ of $K$ such that the products of the elements of that subsequence, $Q(x_{K'_1}) \cdot Q(x_{K'_2}) \cdot ... \cdot Q(x_{K'_j})$, is a perfect square. If we note that

$$Q(x) = (x + \lfloor\sqrt{n}\rfloor)^2 - n \implies Q(x) \equiv (x + \lfloor\sqrt{n}\rfloor)^2 \pmod{n},$$

we are able to restate this congruence by using the subsequence $K'$ as below:

$$\underbrace{Q(x_{K'_1}) \cdot Q(x_{K'_2}) \cdot ... \cdot Q(x_{K'_j})}_{a^2} \equiv \underbrace{(x_{K'_1} - \lfloor\sqrt{n}\rfloor \cdot\ x_{K'_2} - \lfloor\sqrt{n}\rfloor \cdot ... \cdot x_{K'_j} - \lfloor\sqrt{n}\rfloor)^2}_{b^2} \pmod{n},$$

which is exactly identical to our desired result $a^2 \equiv b^2 \pmod{\text{n}}$.

Carl Pomerance, a computer scientist and the inventor of this algorithm, gave it an estimated time complexity to factor an integer $n$ of

$$O\Big(e^{\sqrt{ln(n)\ ln(ln(n))}}\Big) \text{ (Pomerance, 1996).}$$

The algorithm itself is divided into two main sections: data collection and data processing. In the data collection portion, we generate a factor base and begin sieving to get smooth numbers. Both the factor base and sieving will be clarified later in the discussion. In the data processing section, we build a matrix out of the data collected in the previous section and perform calculations on that matrix to ultimately factor $n$.

## 2.3 Data Collection

### 2.3.1 Generating a Factor Base

Now that we know the general process that the quadratic sieve algorithm employs to factor $n$, a question arises that needs to be resolved: how exactly do we find $K'$? In other words, how do we find a subsequence of the Kraitchik sequence $K$ such that the product of the elements of the subsequence, $Q(x_{K'_1}) \cdot Q(x_{K'_2}) \cdot ... \cdot Q(x_{K'_j})$, is a perfect square? To do so, we must first find the prime factors of each element in $K$. A product is a perfect square if the sum of the exponents of matching bases in their prime factorization are all even.

For example, we want to find out if the product $29 \cdot 782 \cdot 22678$ is a perfect square without multiplying the numbers directly. First, calculate the prime factors of 29, 782, and 22678:

$$29 = 29^1$$

$$782 = 2^1 \cdot 17^1 \cdot 23^1$$

$$22678 = 2^1 \cdot 17^1 \cdot 23^1 \cdot 29^1$$

Next, match equal bases together and add their exponents to get:

$$29 \cdot 782 \cdot 22678 = 2^2 \cdot 17^2 \cdot 23^2 \cdot 29^2.$$

Since all of the exponents are even, we can conclude that the product $29 \cdot 782 \cdot 22678$ is a perfect square. This is the same approach we will use to calculate $K'$.

To speed up computations, we want to find the prime factorization of all $Q(x_i) \in K$ over a fixed set of primes called the factor base. This is so that we can put an arbitrary limit on how much data we can collect. If a given $Q(x_i)$ does factor completely over the factor base, we add $Q(x_i)$ and $x_i + \lfloor \sqrt{n} \rfloor$ to our list of collected data. Else, if $Q(x_i)$ does not factor completely over the factor base, we continue on to $Q(x_{i+1})$ and see if that element factors completely over the factor base. The number of computations in the data processing portion of the algorithm is dependent on the amount of data we collect; so, ideally we would want to minimize this as much as possible without affecting the final results.

There are three criteria for choosing a factor base:

- The factor base should always include -1 to handle prime factorizations when $Q(x)$ is negative.
- Each prime $p$ in the factor base should be less than or equal to the smoothness bound $B$.
- All primes in the factor base must satisfy the Legendre symbol $(\frac{n}{p}) = 1$, meaning that $n$ is a quadratic residue (mod $p$).

Generally, the size of the factor base has a positive correlation with the size of $n$, meaning that the size of $B$ increases as $n$ increases. If the size of $B$ is too small, the algorithm would struggle to find numbers that factor completely over the factor base. If the size of $B$ is too large, we would be wasting valuable computation time since the same result could be found with a smaller factor base size. For small $n$, it often suffices to use trial-and-error when choosing $B$ because the number of calculations done by quadratic sieve will be relatively insignificant. However, in order to optimize efficiency in the case for a large $n$, the most optimal size of the factor base (not the smoothness bound) is approximately

$$\left(e^{\sqrt{\ln(n)\ln(\ln(n))}}\right)^{\sqrt{2}/4} \text{(Landquist, 2001).}$$

This approximation attempts to strike a balance between having an adequate size to solve the problem and minimizing computation time.

### 2.3.2 Sieving

Sieving is where the data is collected. It begins by calculating Kraitchik's function $Q(x_i)$ for all integers $x_i$ in the sieving interval $[-M, M]$ to form Kraitchik's sequence $K$. Similar to factor bases, the size of the sieving interval also has a positive correlation with the size of $n$. For small $n$, it will suffice to use trial-and-error when choosing $M$. However, when $n$ is

large, it might be too costly to use trial-and-error. If your interval is too small, the algorithm might result in no answer. Thus, the optimal size of the sieving interval has been calculated as approximately

$$\left(e^{\sqrt{\ln(n)\ln(\ln(n))}}\right)^{3\sqrt{2}/4} \text{(Landquist, 2001).}$$

Note that this is the cube of the optimal factor base size. Since the interval $[-M, M]$ is symmetric about zero, we can say that the optimal value for $M$ is half of the optimal size of the sieving interval

$$M = \frac{1}{2}\left(e^{\sqrt{\ln(n)\ln(\ln(n))}}\right)^{3\sqrt{2}/4}.$$

There are two results that can arise after we calculate a given $Q(x_i) \in K$:

- If $Q(x_i)$ *does* factor completely over the factor base, it is said to be $B$-smooth. We store the values of $Q(x_i)$ and $x_i + \lfloor\sqrt{n}\rfloor$ for further use in the data processing portion of the algorithm.

- If $Q(x_i)$ *does not* factor completely over the factor base, we throw this number away and move on to $Q(x_{i+1})$.

After all elements of $K$ have been processed, we now have a list of $Q(x_i)$ that are $B$-smooth along with a list of their respective values for $x_i + \lfloor\sqrt{n}\rfloor$, both of which represent the data that we collect in this section of the algorithm.

## 2.4 Data Processing

### 2.4.1 Building the Exponent Matrix

In the data processing part of the algorithm, the goal is to find a subsequence $K'$ of $K$ such that the product of the elements of that subsequence $Q(x_{K'_1}) \cdot Q(x_{K'_2}) \cdot \ldots \cdot Q(x_{K'_j})$ is a perfect square. Recall that a product is a perfect square if the sum of the exponents of matching bases in their prime factorization are all even. An easy way to find $K'$ is to first calculate

the prime factorization of each element $Q(x_i) \in K$. Then, we want to create an exponent matrix $E$ from each prime factorization, with each row representing the prime factorization of a given $Q(x_i)$. Since all we care about is that each exponent of matching bases is even, we can simplify calculations by working in (mod 2).

An example of building an exponent matrix is shown here. Let $K = (19343, 114376, 225998)$. We want to find the subsequence $K'$. First, we must calculate the prime factorization of each element in $K$:

$$19343 = 2^0 \cdot 17^0 \cdot 23^1 \cdot 29^2$$

$$114376 = 2^3 \cdot 17^1 \cdot 23^0 \cdot 29^2$$

$$225998 = 2^1 \cdot 17^3 \cdot 23^1 \cdot 29^0.$$

From this, we can form the exponent matrix $E$:

$$\begin{bmatrix} 0 & 0 & 1 & 2 \\ 3 & 1 & 0 & 2 \\ 1 & 3 & 1 & 0 \end{bmatrix} \equiv \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \text{ (mod 2)}.$$

### 2.4.2 Processing the Exponent Matrix

Now, we would like find the subsequence $K'$ such that the product of its elements are prime. Using the exponent matrix $E$, we can easily find the product $Q(x_{K'_1}) \cdot Q(x_{K'_2}) \cdot ... \cdot Q(x_{K'_j})$ by solving the matrix equation

$$S \cdot E = \vec{0} \text{ (mod 2)}$$

for the vector $S$, where each element in $S$ corresponds to a row in $E$ representing the prime factorization of a given $Q(x_i)$. In that sense, the vector $S$ is a binary vector where a value of 1 means that the corresponding $Q(x_i)$ is an element of $K'$.

Using the matrix from the previous example, we want to solve the matrix equation

$$S \cdot \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \pmod 2.$$

The solution is $S = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$, implying that Row 0 + Row 1 + Row 2 $= \vec{0} \pmod 2$, with Row 0 corresponding to 19343, Row 1 corresponding to 114376, and Row 2 corresponding to 225998. Thus, we know that the subsequence $K' = (19343, 114376, 225998) = K$; both $K$ and $K'$ are the same, although this is rarely the case in practical applications. This also means that the product $19343 \cdot 114376 \cdot 225998$ is a perfect square, and if there were any doubts we can always multiply directly to check our results:

$$19343 \cdot 114376 \cdot 225998 = 22360508^2.$$

### 2.4.3 Factoring $n$

From both the data collection and data processing portions of the algorithm, we have calculated the subsequence $K'$, and we also have their corresponding values of $x_{K'_j} - \lfloor \sqrt{n} \rfloor$. Thus, we have all the necessary elements to form the congruence

$$\underbrace{Q(x_{K'_1}) \cdot Q(x_{K'_2}) \cdot \ldots \cdot Q(x_{K'_j})}_{a^2} \equiv \underbrace{(x_{K'_1} - \lfloor \sqrt{n} \rfloor \cdot \ x_{K'_2} - \lfloor \sqrt{n} \rfloor \cdot \ldots \cdot x_{K'_j} - \lfloor \sqrt{n} \rfloor)^2}_{b^2} \pmod n,$$

which was our primary goal from the very beginning.

Now that we have this, it is a simple matter of computing $\gcd(x-y,\ n)$ and $\gcd(x+y,\ n)$ to find the two factors of $n$. However, there is a little caveat in this: there is approximately a 50% chance that you find a trivial factor, i.e. $n$ or 1. If this happens, just choose another subsequence $K'$ of $K$ whose product is a square. If another $K'$ does not exist, try adjusting the smoothness bound $B$ or the sieving interval $[-M, M]$ and repeating the algorithm.

## 2.5 A Full Example

The inherent complexity of this algorithm necessitates a full example from start to finish. Say that we want to factor $n = 87463$ using the quadratic sieve.

The first step in this process is to generate the factor base. Recall that we always include -1 in the factor base, every prime $p$ in the factor base must be less than or equal to the smoothness bound $B$, and the Legendre symbol $(\frac{n}{p})$ must equal 1. Since $n$ is relatively small, we can just choose an arbitrary smoothness bound to be $B = 37$. If $n$ were larger, we would use the approximation of the most optimal factor base size from the previous section. Below is the table of Legendre symbol calculations for every prime less than or equal to our set smoothness bound of 37:

| $p$ | 2 | 3 | 5 | 7 | 11 | 13 | 17 | 19 | 23 | 29 | 31 | 37 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Legendre symbol | 1 | 1 | -1 | -1 | -1 | 1 | 1 | 1 | -1 | 1 | -1 | -1 |

If we take all of the primes $p$ with Legendre symbols of 1 and add -1, we get our factor base:

$$\{-1, 2, 3, 13, 17, 19, 29\}.$$

Next, we begin sieving to find all $B$-smooth numbers. Again, since $n$ is small we can choose an arbitrary sieving interval of [-30, 30]; if $n$ were large, we would benefit from using the optimal sieving interval size discussed in the previous section. To start the sieve, we calculate the Kraitchik function $Q(x_i) = (x_i + \lfloor\sqrt{n}\rfloor)^2 - n$ for each integer $x_i$ in the sieving interval and see if $Q(x_i)$ factors completely over the factor base. If it does, we keep track of the values of $Q(x_i)$ and $x_i + \lfloor\sqrt{n}\rfloor$. For the sake of brevity, $Q(x_i)$ calculations are not shown for every $x_i$ in [-30, 30]. Only the ones that factor completely over the factor base and their corresponding values of $x_i + \lfloor\sqrt{n}\rfloor$ are shown in the table below:

| $Q(x_i)$ | $x_i + \lfloor \sqrt{n} \rfloor$ |
|---|---|
| -17238 | 265 |
| -10179 | 278 |
| 153 | 296 |
| 1938 | 299 |
| 6786 | 307 |
| 12393 | 316 |

This table represents the data that we collect in this portion of the algorithm. With this data, we can continue to the data processing.

The first step in data processing is building the exponent matrix. To do so, we must calculate the prime factorization for each $Q(x_i)$ that we stored in the table above, which is shown here:

$$-17238 = -1^1 \cdot 2^1 \cdot 3^1 \cdot 13^2 \cdot 17^1 \cdot 19^0 \cdot 29^0$$

$$-10179 = -1^1 \cdot 2^0 \cdot 3^3 \cdot 13^1 \cdot 17^0 \cdot 19^0 \cdot 29^1$$

$$153 = -1^0 \cdot 2^0 \cdot 3^2 \cdot 13^0 \cdot 17^1 \cdot 19^0 \cdot 29^0$$

$$1938 = -1^0 \cdot 2^1 \cdot 3^1 \cdot 13^0 \cdot 17^1 \cdot 19^1 \cdot 29^0$$

$$6786 = -1^0 \cdot 2^1 \cdot 3^2 \cdot 13^1 \cdot 17^0 \cdot 19^0 \cdot 29^1$$

$$12393 = -1^0 \cdot 2^0 \cdot 3^6 \cdot 13^0 \cdot 17^1 \cdot 19^0 \cdot 29^0$$

From these prime factorizations, we can formulate the resulting exponent matrix in modulo 2.

| $Q(x_i)$ | -1 | 2 | 3 | 13 | 17 | 19 | 29 |
|---|---|---|---|---|---|---|---|
| -17238 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| -10179 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 153 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1938 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 6786 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 12393 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

We use a table to clarify which $Q(x_i)$ each row represents and also which base in the factor base each column represents.

Now that we have our exponent matrix $E$, we can process this to find $K'$ whose elements form a product $Q(x_{K'_1}) \cdot Q(x_{K'_2}) \cdot ... \cdot Q(x_{K'_j})$ that is even. We do this by solving the matrix equation $S \cdot E = \vec{0} \pmod 2$ for the vector $S$. We have that

$$S \cdot \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} = \vec{0} \pmod 2 \implies S = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

The vector $S$ implies that the sum of row 0, row 1, row 2, and row 4 is $\vec{0}$ modulo 2. We cross-reference the table above to check each row's corresponding $Q(x_i)$ value and conclude that $-17238 \cdot -10179 \cdot 153 \cdot 6786$ is a perfect square.

Now, with all of this information, we are able to completely factor $n$. We get the congruence

$$a^2 \equiv b^2 \pmod n$$

$$Q(x_{K'_1}) \cdot Q(x_{K'_2}) \cdot ... \cdot Q(x_{K'_j}) \equiv (x_{K'_1} - \lfloor \sqrt{n} \rfloor \cdot \ x_{K'_2} - \lfloor \sqrt{n} \rfloor \cdot ... \cdot x_{K'_j} - \lfloor \sqrt{n} \rfloor)^2 \pmod n$$

$$-17238 \cdot -10179 \cdot 153 \cdot 6786 \equiv (265 \cdot 278 \cdot 296 \cdot 307)^2 \pmod{87463}.$$

Therefore,

$$a = \sqrt{-17238 \cdot -10179 \cdot 153 \cdot 6786}$$

$$b = 265 \cdot 278 \cdot 296 \cdot 307.$$

Recall that once we get $a$ and $b$, we can get the factorization of $n$ by calculating $\gcd(x-y,\ n)$ and $\gcd(x+y,\ n)$. By doing so, we get factors

$$\gcd(a-b,\ n) = 149$$

$$\gcd(a+b,\ n) = 587.$$

Thus, the problem is solved, and we conclude that $n = 87463 = 149 \cdot 587$.

# Chapter Three

# Implementation

As part of my honors research requirement, I was tasked to implement the quadratic sieve algorithm. This implementation is hosted on GitHub, and the instruction on how to use it is found in the README file. In this chapter, I will briefly reflect on my program by discussing noteworthy implementation details and mentioning some future improvements.

This implementation of the quadratic sieve algorithm was written in Python. The rationale that caused me to choose this language was a combination of my familiarity with Python and its outright simplicity. The algorithm itself was already complex, so I knew that choosing a more verbose programming language, such as Java, might cause complications.

There was one positive that I did not take into account when choosing Python though: Python can deal with arbitrarily large integer values without having to explicitly set variable types. This is desirable because the algorithm does computation with extremely large numbers, especially when $n$ is large. However, this problem was not entirely eliminated as Python still has an upper limit to their numbers, which is the IEEE 754 64-bit double maximum value or approximately $1.8 \times 10^{306}$. That means that any calculations involving numbers greater than $1.8 \times 10^{306}$ would cause an integer overflow. Python's way of dealing with this is automatically transforming said numbers to the 'inf' type, which ultimately leads to unintended results.

My workaround for this is using decimal module in Python's standard library which

allows for numbers larger than the 64-bit double maximum value. The only caveat to this is that we must set a precision for our numbers. If the precision is not sufficient, the least significant digits of our calculations will be truncated. If we are over-precise, we would waste precious memory. Thus, a careful balance must be set. For testing purposes, I set this precision value to be 1000, meaning that calculations will be precise up to 1000 digits long. This was sufficient enough for my tests, however, if one was to test extremely large numbers ($> 150$ bits), it may not be adequate.

Another issue that I had to solve was that probability that the algorithm resulted in the undesirable trivial solution seemed exceptionally high in early versions of my implementation. The problem was linked to the amount of data I collected. In the early versions, I limited this amount to equal the size of the factor base, which generates a square exponent matrix in the data processing portion of the algorithm. This was the likely source of the problem. To combat this while still allowing data processing to run efficiently, I instead increased this bound to 1.1x the size of the factor base. This ultimately lead to more data being collected, which resulted in a higher chance that a non-trivial result would be found.

There are also some differences in my implementation versus the algorithm as prescribed in Chapter 2. This was due to the different resources I had access to during the development of this program. Much of this implementation stems from "A Tale of Two Sieves" by Carl Pomerance, the original inventor of the quadratic sieve. On the other hand, a majority of the details in Chapter 2 came from Eric Landquist's "The Quadratic Sieve Factoring Algorithm", which describes methods that provide efficiency improvements when compared to the methods in the original paper by Pomerance. The main difference between my implementation and the algorithm described in Chapter 2 is that my sieving interval was $[0, M]$ instead of $[-M, M]$. This change completely eliminates negative Kraitchik functions within my implementation, thus I also did not need -1 in my factor base. That being said, I also did adopt one feature from Landquist into my program, that being the Legendre symbol condition when generating factor bases. Overall, the core logic of the algorithm stayed intact, and

I was still able to arrive at the correct result with a 100% success rate, albeit slightly less efficiently.

# Chapter Four

# Conclusion

Integer factorization is a difficult problem to solve, and is an ongoing research topic among mathematicians and cryptographers alike. We have discussed such an algorithm aimed at solving this problem: the quadratic sieve. We explained the methods and processes that the algorithm employs, and used it to factor a non-trivial integer as a practical example of its use. Additionally, a Python implementation was provided along with a short reflection on the implementation process.

# REFERENCES


Landquist, Eric (2001). *The Quadratic Sieve Factoring Algorithm*. Paper. Charlottesville VA: University of Virginia.

O'Neill, Melissa E. (2009). *The Genuine Sieve of Eratosthenes*. Paper. Claremont CA: Harvey Mudd College.

Pomerance, Carl (1996). "A Tale of Two Sieves". In: *Notices of the American Mathematical Society* 43.12, pp. 1473–85.

Shor, Peter W. (1994). "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134.

Zimmerman, Paul (2020). *Factorization of RSA-250*. URL: https://lists.gforge.inria.fr/pipermail/cado-nfs-discuss/2020-February/001166.html.