The Quadratic Sieve - An Implementation

Damian A. Ball
Brandon Morton

University of Idaho
Mathematics
Cryptography

December 15, 2010

Abstract

The Quadratic sieve is currently used as an efficient algorithm to factor composite integers. Though it's performance is second to the general number field sieve when considering integers greater than 110 digits, the quadratic sieve is much more simple to implement. In this paper we describe the general parts of the quadratic sieve as well as how we implemented the algorithm using Java. While we worked on this project, we attempted to gain a better understanding of how memory and CPU time were used by the sieve. When we profiled the application, we found methods like Gaussian elimination (GE) were effective in reducing overall computation time and by comparing one version of the program without GE to one with GE, we were quickly able to see the value of the method. Our implementation of the quadratic sieve is easily able to factor numbers 22 digits in length. We are pleased with this result.

1 Introduction

Cryptography is a critical component to digital communication. Banks, commerce, businesses, governments, and the like all have an interest in communicating over the World Wide Web (WWW). Perhaps more importantly, they all have an interest in communicating securely: with assurance that the message transmitted had not been modified, had not been read by another party, and truly came from whom the sender claims. As a result, cryptographic systems were developed to meet those goals. Historically, simple ciphers attempted to hide a message from other parties, the Caesar cipher is a basic example. Others were developed to be resilient to attacks as those attacks were developed. And with the advent

of the computer, major modifications were required to defeat growing computation capabilities. Mathematics play a central role in our ciphers as many now rely on difficult to solve problems at the core of their algorithm. The problem we focus on in this paper is the difficulty of factoring composite integers. The security of many of our important communication systems rely on the difficulty to compute the factors of a composite integer. The RSA algorithm specifically uses a composite integer for modular arithmetic. And the RSA algorithm is used in most, if not all, Secure Hyper Text Protocol (HTTPS) and Secure Socket Layer (SSL) communication. That being said, even though the factoring problem is difficult to solve, there are methods that can make the problem easier than checking a given composite integer n against all prime numbers less than $\sqrt{n}$. These methods involve some form of number sieve to - in effect - sift out the prime factors of a given number. Though this may sound simple, number sieves still run up against a computation time and space constraint that makes them unable to solve large problems in a reasonable amount of time. Thus, large composite integers are still in use and continue to protect our important communication. This paper will focus on, and act as a companion to, an implementation of the Quadratic Sieve that was written for this project.

2 Background

Carl Pomerance invented the Quadratic Sieve in 1981 and released it in his article, "Analysis and Comparison of Some Integer Factoring Algorithms." It is considered the fastest known sieve on numbers less than 110 digits and the second fastest for numbers above. The Quadratic Sieve was introduced as an improvement on Schroeppel's linear sieve. The earliest known sieve is call the Sieve of Eratosthenes and is discussed later in this paper.

3 Definitions

3.1 Variables

In this paper, n is the composite integer we want to factor using the Quadratic Sieve.

3.2 B-smooth

An integer s is considered B-smooth if s can be completely factored only using the primes less than B.

3.3 Exponent Vector

An exponent vector for an integer s and a factor base FB is the ordered list of exponents generated by factoring s over FB. Table 1 provides an example of a matrix of exponent vectors.

4 The Algorithm

The Quadratic Sieve algorithm is effectively looking for two integers $x, y \in Z/nZ$ such that:

$$\begin{aligned} x &\not\equiv \pm y \text{ (mod n)} \\ x^2 &\equiv y^2 \text{ (mod n)} \end{aligned} \tag{1}$$

If we find such integers, we are guaranteed that $gcd(n, x - y)$ will yield a factor of n. The order of the algorithm is described below:

1. Set the smoothness bound B which will be the size of our factor base.
2. Sieve to generate a list X of "Big Roots" x_i that are B-smooth.
3. Factor all $x_i \in X$ to generate the exponent vectors mod 2.
4. Solve the systems of equations to determine linear dependence (find vectors who's sum is the zero vector) using a matrix of exponent vectors from step 3.
5. Select $x^2 \equiv y^2 \bmod n$.
6. Compute $gcd(n, x - y)$ and check if the factor is non-trivial, if so, the algorithm is complete. If not, select a new x or find another linear dependency from step 4.

5 Implementation

For our implementation we used a paper written by Eric Landquist on the Quadratic Sieve for reference. As a result, Landquist's paper is cited throughout this section.

5.1 Helper Functions

A set of functions are critical to the Quadratic Sieve algorithm but are not explicitly part of the algorithm. In some cases, there exist other - sometimes faster - algorithms that achieve the same results, but we chose to implement functions described in both our course book [2] and in Landquist's paper [1].

5.1.1 Sieve of Eratosthenes

The Sieve of Eratosthenes is used to generate the factor base for the Quadratic Sieve. In order to generate the factor base primes, the following algorithm is used:

1. Generate a list m of integers from 2 to k
2. Let $p = 2$
3. Remove all multiples of p from m
4. Select the next number $\widehat{p}$ in m after p and set $p = \widehat{p}$
5. Loop over steps 3 and 4 until $p^2 > n$
6. m is now the list of primes from 2 to n

This algorithm is known to be slow for large values of k; however, the primes used in the factor base are considered small.

5.1.2 Factor Base Size

The factor base is the ordered list L of small primes over which candidates for squares x_i are factored. As previously mentioned, "Big Roots" x_i are selected because they are B-smooth; meaning they factor over the factor base. Landquist [1] suggests that the optimum choice for the factor base size B is:

$$B = \left(e^{\sqrt{ln(n)ln(ln(n))}}\right)^{\sqrt{2}/4} \tag{2}$$

For the purposes of our implementation, we us a slightly modified $\widehat{B}$ as a Java `int` with roundoff error from approximating n:

$$\widehat{B} = \lfloor B \rfloor \tag{3}$$

5.1.3 Sieve Interval

The sieve interval is used when creating "Big Roots" as a bound on how far from $\lfloor \sqrt{n} \rfloor$ the algorithm will check for candidate roots x_i. The "Big Root" function is given by equation 6. A rough approximation of the sieving interval M is also given by Landquist [1] as:

$$M = B^3 \tag{4}$$

As with equation 3, we use a modified $\widehat{M}$:

$$\widehat{M} = \widehat{B}^3 \tag{5}$$

This bound is traditionally implemented as $\left[-\widehat{M}, \widehat{M}\right]$. However, for the purposes of this project, we restricted the left side of the bound to 1 s.t. our bound is $\left[1, \widehat{M}\right]$.

5.1.4 Legendre Symbol

The Legendre symbol is used in the Quadratic sieve to select a set of prime numbers for the factor base for which n is a quadratic residue mod p.

5.1.5 getFactorBase(k)

The getFactorBase(k) function returns an ordered list FB of k prime numbers that have a Legendre symbol of 1. That is to say, n is a quadratic residue mod p, where $p \in FB$. We elected to exclude -1 from our factor base. Landquist's paper [1] was not clear about the value of -1 and it appeared to only confuse the math.

5.1.6 Heron's Method

Heron's method, aka the "Babylonian method", is a method for computing square roots. The method starts with a positive guess value and iteratively approaches the actual root to a certain precision. Since Java, the language used to program this algorithm, does not include a square root function for the `BigInteger` class, we had to write the function ourselves. We chose Heron's method to compute the square root of a `BigInteger`. The function is titled FloorRoot because we take the floor of the result to maintain the integer property of `BigInteger`. Below is the code snippet for the square root function:

```
private static BigInteger FloorRoot(BigInteger x)
{
    BigInteger one = BigInteger.valueOf(1);
    BigInteger two = BigInteger.valueOf(2);
    long intermediateGuess = (long)Math.sqrt(x.floatValue());
    BigInteger guess = BigInteger.valueOf(intermediateGuess);

    boolean guessBelowGoal;
    boolean guessPlusOneAboveGoal;
    do
    {
        guess = guess.add(x.divide(guess)).divide(two);

        guessBelowGoal = guess.pow(2).compareTo(x) <= 0;
        guessPlusOneAboveGoal = guess.add(one).pow(2).compareTo(x) > 0;
    } while (!(guessBelowGoal && guessPlusOneAboveGoal));

    return guess;
}
```

Table 1: Example Sieving Matrix

x	2	3	5	7	factorization	exponent vector
15	0	1	1	0	$2^0 3^1 5^1 7^0$	$\{0, 1, 1, 0\}$
20	0	0	1	0	$2^2 3^0 5^1 7^0$	$\{0, 0, 1, 0\}$
21	0	1	0	1	$2^0 3^1 5^0 7^1$	$\{0, 1, 0, 1\}$

5.1.7 getBigRoot(k)

The getBigRoot(k) function takes an integer k that is inside the sieving interval $\left[-\widehat{M}, \widehat{M}\right]$ and returns a "Big Root" result from the following function:

$$getBigRoot\,(k) = \left(k + \left\lfloor \sqrt{n} \right\rfloor\right)^2 \pmod{\text{n}} \tag{6}$$

5.2 Sieving Matrix

The sieving matrix is a matrix $A[i, j]$ whose j values are the powers of the primes in the factor base that divide into a "Big Root" x_i mod 2. For example, let our factor base $FB = \{2, 3, 5, 7\}$ and our list of "Big Roots" $X = \{15, 20, 21\}$. The sieving matrix for FB, X is shown in table 1. The x, factorization, and exponent vector values are added for explanation and do not exist in the traditional sieving matrix.

5.3 Gaussian Elimination

When solving a matrix as a system of equations, there are many different algorithms available for use. One method that is time efficient is Gaussian elimination. Gaussian elimination is covered in detail in a Linear Algebra course and as such, we will not explain it in detail. In essence, the algorithm reduces the matrix into row echelon form using elementary row operations and then further reduces it into reduced row echelon form. When the matrix is in it's final form, a solution to the system - if it exists - will be apparent. We implemented Gaussian elimination as the method for solving the sieving matrix for our program.

5.4 Room for improvement

As previously mentioned, there are ways in which individual parts of this program could be swapped for greater memory and speed efficiency. Using the Block Wiedemann sparse matrix during the Gaussian elimination step was one such swap was mentioned in Landquist's paper [1]. However, due to time constrains, we did not implement this method.

6 In Action

When running our application we could factor a 22 digit composite integer to it's non-trivial prime factors in roughly 15 seconds. A simple factoring method for 10 digits took twice as long and appeared to increase exponentially in time complexity with every digit.

7 Conclusion

The Quadratic Sieve is a simple and viable solution to use when factoring composite integers. It's functional parts are small, interchangeable, and can be enhanced. Other papers we read indicated and gave examples of how this algorithm could be distributed across multiple cores and computers. If such distribution was time efficient, factoring large composite integers will become much easier. That being said, as we mentioned in the introduction to this paper, there are still space and time constraints on the Quadratic Sieve and other sieves that will prevent large scale attacks on cryptographic systems that rely on large composite integers.

8 Java Application: QuadraticSieve

8.1 How to use/run in Ubuntu

To use this application you will need to have the Java Run-time Environment (JRE) or the Java Development Kit (JDK) installed. You can install the JRE with the following command:

```
sudo apt-get install openjdk-6-jre
```

Once you have the JRE or JDK installed, you can navigate to the application's jar file and run the application with:

```
java -jar QuadraticSieve.jar n
```

Where n is the number you would like to factor.

8.2 Verification

Here are some test results:

```
java -jar QuadraticSieve.jar 15
The factors of N are:
3
5
```

```
java -jar QuadraticSieve.jar 335752297121
The factors of N are:
548657
611953

java -jar QuadraticSieve.jar 335752297127
The factors of N are:
36019
9321533
```

References

[1] Eric Landquist. The quadratic sieve factoring algorithm, 2001.

[2] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography: with Coding Theory*. Pearson: Prentice Hall, second edition, 2006.