Implementation and Analysis of the Quadratic Sieve Algorithm for Integer Factorization

Sam Borremans, Aahan Mehta

May 7, 2025

1 Introduction

The quadratic sieve algorithm, developed by Carl Pomerance in the early 1980s, represents one of the most effective general-purpose factorization methods for integers in the 20-120 digit range. It builds upon earlier work by Dixon and Schroeppel, refining the approach of finding congruent squares modulo the number to be factored [1].

In this paper, we describe our implementation of the quadratic sieve algorithm, including various optimizations and parallelization techniques. We analyze both the theoretical complexity and practical performance considerations, comparing our results with theoretical predictions.

2 Mathematical Background

2.1 Factoring via Difference of Squares

The motivation behind the quadratic sieve algorithm relies on a fundamental number-theory principle: if we can find integers a and b such that:

$$a^2 \equiv b^2 \pmod{n} \tag{1}$$

where $a \not\equiv \pm b \pmod{n}$, then $\gcd(a-b, n)$ is a non-trivial factor of n.

This works because from the congruence $a^2 \equiv b^2 \pmod{n}$, we can derive that $a^2 - b^2 \equiv 0 \pmod{n}$, which means n divides $(a+b)(a-b)$. When n is composite with at least two distinct prime factors and $a \not\equiv \pm b \pmod{n}$, n cannot divide both $(a+b)$ and $(a-b)$ as else:

$$a + b \equiv a - b \pmod{n}$$
$$2a \equiv 2b \pmod{n}$$

and as $\gcd(2, n) = 1$, this would imply $a \equiv b \pmod{n}$ which is a contradiction. . Instead, one factor will share some prime factors with n while the other factor shares the remaining prime factors. Therefore, computing $\gcd(a-b,n)$ will yield a proper divisor of n that is neither 1 nor n itself.

To find such pairs (a,b), the quadratic sieve examines values of the polynomial $f(x)=x^2-n$ for many values of x near $\sqrt{n}$. If we can identify a subset of these values whose product is a perfect square, we can construct our congruence of squares.

2.2 Smooth Numbers

A positive integer is considered B-smooth if all of its prime factors are less than or equal to a bound B. Smooth numbers form the mathematical foundation of the quadratic sieve algorithm. Let $\pi(B)$ be defined as the number of primes up to B.

As shown by Pomerance, if we collect more than $\pi(B)$ B-smooth numbers, we are guaranteed to find a subset of those numbers such that their product is a perfect square [1]: for each B-smooth number m with prime factorization $m=p_1^{e_1}\cdot p_2^{e_2}\cdots p_k^{e_k}$, we can form an exponent vector $(e_1 \bmod 2, e_2 \bmod 2, \ldots, e_k \bmod 2)$. With more than $\pi(B)$ such vectors in an $\pi(B)$-dimensional vector space over $\mathbb{F}_2$, linear dependency is guaranteed. This dependency directly translates to a subset of numbers whose product has all even exponents.

2.3 The Factor Base

The factor base in the quadratic sieve consists of the primes $p\le B$ for which n is a quadratic residue modulo p. These are precisely the primes that can appear in the factorization of the values we are examining. If n were not a quadratic residue $\pmod{p}$, then the congruence $x^2\equiv n \pmod{p}$ would have no solution and p would never divide x^2-n for any x.

The quadratic character of n modulo p is determined by the Legendre symbol $\left(\frac{n}{p}\right)$, which equals 1 if n is a quadratic residue modulo p, 0 if p divides n, and -1 otherwise. Only primes with Legendre symbol 1 are included in our factor base, whereas primes with Legendre symbol 0 are directly used as factors (this happens very rarely).

For each prime p in the factor base, there exist solutions r_1 and r_2 to the congruence $x^2\equiv n \pmod{p}$. These solutions tell us where in our sieve interval the values of the polynomial $f(x)=x^2-n$ will be divisible by p, which is crucial for the efficient operation of the sieving process.

3 Preprocessing

Before we run the main quadratic sieve algorithm, we first run some basic processing.

3.1 Small Prime Factors

To check whether any small primes can factor n, we first generate all primes smaller than $\log(n)$ using the Sieve of Eratosthenes:

1. Let $L = \lfloor \log(n) \rfloor$.
2. Create a boolean array $A[2 \ldots L]$ initialized to `true`.
3. For each i from 2 to $\sqrt{L}$:
    - If $A[i]$ is `true`, mark all multiples of i greater than i as `false`.
4. After completion, all indices i for which $A[i]$ is `true` represent prime numbers less than or equal to $\log(n)$.

The Sieve of Eratosthenes runs in $\mathcal{O}(L \log\log L)$ time, where $L = \log(n)$. Therefore, the total time complexity of this step is: $\mathcal{O}(\log(n) \log\log\log(n))$ which is negligible compared to the overall complexity of the quadratic sieve.
Once we have the list of primes, we check each one to see if it divides n. This requires up to $\mathcal{O}(\log n)$ trial divisions, each costing $\mathcal{O}(\log n)$ time (for division of big integers), giving an overall cost of $\mathcal{O}(\log^2 n)$ for this divisibility checking step. Again, this is a small cost relative to the main sieve.

3.2 Miller-Rabin Test

After removing small primes, we can check whether the remaining number n (divided by its small prime factors) is prime. This can be conducted using the Miller-Rabin Test, which is a probabilistic primality test.

Let n be the odd integer we want to test for primality, with $n > 2$. We repeat the test k times.

1. Write $n - 1 = 2^s \cdot d$ where d is odd.
2. Repeat the following k times:
    (a) Pick a random integer a in the range $[2, n-2]$.
    (b) Compute $x = a^d \mod n$.
    (c) If $x = 1$ or $x = n - 1$, continue to the next iteration.
    (d) Otherwise, repeat up to $s - 1$ times:
        - Compute $x = x^2 \mod n$.
        - If $x = n - 1$, break and continue to the next iteration.
    (e) If none of the squarings resulted in $n - 1$, then n is definitely composite.

3. If all k rounds pass, then n is probably prime.

It has been proven that for any odd composite number n, at most $1/4$ of the bases a in the valid range incorrectly suggest that n is prime [2]. This means the probability that a random base fails to detect compositeness is at most $\frac{1}{4}$. As such, to reduce the chance of error, we repeat the test for k independent random bases. If n is composite, the chance it passes all k tests is at most:

$$\left(\frac{1}{4}\right)^k$$

In our implementation, we use $k = 20$ rounds, which makes the probability that a composite number is falsely classified as prime no more than:

$$\left(\frac{1}{4}\right)^{20} \approx 9 \times 10^{-13}$$

It is thus unlikely that the program exits because it believes n is prime even if it is not. However, if the program does exit even though the user is certain the number n is prime, a flag in the configuration file can be edited to skip this Miller-Rabin test.

Finally, we can check if the number is a perfect square. If not, we can proceed with the quadratic sieve algorithm.

4 The Quadratic Sieve Algorithm

The quadratic sieve works by finding many values of x such that $f(x) = x^2 - n$ is B-smooth. Once enough such relations are found, linear algebra over $\mathbb{F}_2$ is used to identify a subset of these values whose product is a perfect square, leading to a congruence of squares modulo n.

4.1 Overview of the Algorithm

The Quadratic sieve algorithm for factoring large integers n is as follows:

Algorithm 1 Quadratic Sieve Algorithm

```
1: Choose a smoothness bound $B$
2: Generate a factor base of primes $p \leq B$ where $n$ is a quadratic residue modulo $p$
3: Sieve to find values of $x$ where $x^2 - n$ is $B$-smooth
4: Form a matrix of exponent vectors (mod 2) for these smooth relations
5: Use Gaussian elimination to find linear dependencies in the matrix
6: Use these dependencies to construct congruent squares modulo $n$
7: Compute $\gcd(a - b, n)$ to find a non-trivial factor
```

4.2 Sieving Process

The sieving process is the heart of the quadratic sieve algorithm and is responsible for efficiently finding the B-smooth values of $f(x) = x^2 - n$. Rather than testing each value of $f(x)$ individually through trial division, we employ a sieve-based approach similar to the Sieve of Eratosthenes.

The key insight is that if p is a prime in our factor base and r is a solution to $x^2 \equiv n \pmod{p}$, then p divides $f(x)$ whenever $x \equiv r \pmod{p}$. This allows us to identify positions in our sieve interval where divisibility by p occurs in a systematic way.

For each prime p in the factor base:

1. Solve the congruence $x^2 \equiv n \pmod{p}$ to find solutions r_1 and r_2 using the Tonelli-Shanks algorithm (discussed in detail in the implementation section).
2. For each solution r, mark every position x in the sieve array where $x \equiv r \pmod{p}$.
3. At each marked position, subtract $\log(p)$ from the logarithmic approximation of $|f(x)|$.

The logarithmic sieving approach is based on the observation that if m has prime factorization $m = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$, then $\log(m) = e_1 \log(p_1) + e_2 \log(p_2) + \cdots + e_k \log(p_k)$. Thus, by initializing our sieve array with $\log(|f(x)|)$ and subtracting $\log(p)$ each time we find a divisibility by p, the positions with values close to zero after sieving are likely to correspond to B-smooth values of $f(x)$.

5 Implementation Details

Our implementation of the quadratic sieve algorithm incorporates several optimizations to improve performance. The code is written in C++ using the GMP library for arbitrary precision arithmetic and OpenMP for parallelization.

5.1 Smoothness Bound Selection

The choice of smoothness bound B significantly impacts the algorithm's performance. Too small a value makes smooth numbers rare, while too large a value creates excessive work in the linear algebra phase. Following theoretical analysis, we select B using the formula:

$$B = \exp\left((1/2 + O(1))(\log n \log\log n)^{1/2}\right)$$

This formula is derived from optimizing the expected running time by balancing the cost of finding smooth numbers and the cost of the linear algebra step. After experimenting with different values ourselves using the program we wrote, we chose 0.05 as the value for $O(1)$, but this can be edited in the config file. However, in future versions, it could be interesting to determine the value for this constant based on the input number instead of a constant.

5.2 Sieve Interval Selection and Management

The sieve interval is a critical parameter that determines the range of x values we examine at once. The choice of sieve interval affects both the efficiency of finding smooth relations and memory consumption. In our implementation, the sieve interval starts with a default value defined in the configuration file (SIEVE_INTERVAL) and can be adaptively increased during execution.

5.2.1 Initial Sieve Interval

We begin with a moderate sieve interval size to balance memory usage and the probability of finding smooth relations. The starting point for our sieve is set at $\lceil\sqrt{n}\rceil$, as values of $f(x) = x^2 - n$ are minimized in absolute value near $\sqrt{n}$, making them more likely to be smooth.

The initial sieve interval is a configurable parameter that can be adjusted based on the available memory and the size of the input number. For our experiments, we typically used around 10^5, which is the largest number for which we can find smooth numbers in a fast time.

5.2.2 Adaptive Sieve Interval Expansion

Our implementation includes an adaptive mechanism to increase the sieve interval when the rate of finding smooth relations is too low. If after several attempts (specifically, every 5 attempts in our implementation), the number of collected relations is less than half the size of the factor base, we multiply the sieve interval by a factor of 10, up to a maximum specified by MAX_SIEVE_INTERVAL.

This adaptive approach ensures that for smaller numbers, we avoid excessive memory usage and finding unnecessary amounts of smooth numbers by starting with a moderate interval. For larger numbers where we need more smooth numbers to find dependencies, we progressively increase the sieve interval to find more relations. The program can thus adapt to different factorization problems without requiring manual tuning

5.2.3 Parallelized Sieve Processing

To efficiently process the sieve interval, we parallelize multiple aspects of the sieving process using OpenMP(the initialization of the sieve array with logarithmic values of $|x^2 - n|$, the processing of each prime in the factor base to mark positions divisible by that prime and the verification of potentially smooth candidates through trial division).

The sieve array itself is represented as a vector of double-precision floating-point values, where each position corresponds to an x value in the current interval. We initialize this array with the logarithms of $|f(x)| = |x^2 - n|$ for each x.

The actual sieving operation is then performed by subtracting $\log(p)$ at positions where p divides $f(x)$. This is also parallelized across the primes in the factor base.

After sieving, we identify positions where the sieve array values are close to zero (specifically, less than 0.1 in our implementation to account for floating-point errors). These positions are likely to correspond to smooth values and are collected as candidates for verification through trial division.

5.3 Parallelized Sieving

We parallelize the sieving process using OpenMP, accelerating the most time-consuming part of the algorithm. Each thread processes a portion of the sieve interval independently, and results are combined afterwards. This approach scales effectively with the number of available processor cores and provides speedup on modern multi-core systems.

5.4 Tonelli-Shanks Algorithm

The Tonelli-Shanks algorithm is an efficient method for computing square roots modulo a prime p. In the context of the quadratic sieve, this algorithm is essential because we need to solve the congruence $x^2 \equiv n \pmod{p}$ for each prime p in our factor base to determine where p divides $f(x) = x^2 - n$ in our sieve interval.

The algorithm works as follows:

1. Express $p - 1 = Q \cdot 2^S$ where Q is odd
2. Find a quadratic non-residue z modulo p (any number with Legendre symbol -1)
3. Initialize $c \equiv z^Q \pmod{p}$, $R \equiv n^{(Q+1)/2} \pmod{p}$, $t \equiv n^Q \pmod{p}$, and $M = S$
4. While $t \neq 1$:
   (a) Find the smallest i such that $t^{2^i} \equiv 1 \pmod{p}$
   (b) Set $b \equiv c^{2^{M-i-1}} \pmod{p}$
   (c) Update $R \equiv Rb \pmod{p}$, $t \equiv tb^2 \pmod{p}$, $c \equiv b^2 \pmod{p}$, and $M = i$
5. Return R and $p - R$ as the two square roots of n modulo p

Without this algorithm, we would need to use brute force to find the roots, which would require $\mathcal{O}(p)$ operations per prime. The Tonelli-Shanks algorithm reduces this to $\mathcal{O}(\log^2 p)$ operations, making it feasible to work with much larger primes in our factor base.

5.5 Gaussian Elimination and Dependency Finding

Once sufficient smooth relations are found, we construct a matrix where each row represents the exponent vector of a smooth relation modulo 2. Gaussian elimination is then used to find linear dependencies in this matrix. This process is also parallelized as most operations (checking for 0 rows, and eliminating 1's from non pivot columns) on rows are independent.

The implementation for the Gaussian Elimination function works in modulo 2, tracking dependencies in a transformation matrix T while operating on a matrix M. Working over $\mathbb{F}_2$ allows us to use efficient bit operations (XOR for addition, AND for multiplication), which significantly improves performance compared to general-purpose linear algebra routines.

Our implementation includes several optimizations:

- Parallelization using OpenMP to exploit multi-core processors
- Tracking processed rows and columns to avoid redundant work
- Using a sparse representation initially to handle large matrices efficiently
- Optimized bit operations for $\mathbb{F}_2$ arithmetic

The algorithm returns a set of dependency vectors, each representing a subset of relations whose product is a perfect square. These dependencies are then used to construct congruent squares modulo n, potentially yielding factors of n.

6 Performance Analysis

The theoretical complexity of the quadratic sieve is:

$$\exp\left((1+o(1))(\log n \log\log n)^{1/2}\right)$$

This expression represents the number of bit operations required to factor n. The derivation involves analyzing the cost of finding B-smooth numbers and the cost of the linear algebra step.

In practice, the algorithm's performance is influenced by several factors, including the size of the input number, the choice of smoothness bound B the size of the sieve interval, the efficiency of the linear algebra implementation and hardware characteristics (processor cores, cache sizes, memory bandwidth). Our experiments show that slightly deoptimizing B on the low side can improve performance by reducing the size of the matrix in the linear algebra step, particularly when memory constraints are significant.

7 Experimental Results

After experimentation with several large composite numbers, we have produced a list of runtimes for several large numbers.

N	$\exp\left((\log n \log\log n)^{1/2}\right)$	Runtime (s)
15123969271373	26,285.20	0.026
86024948691518227	155,113.37	0.179
1701495534744665061 7	423,203.62	0.062
1918101644538405952783	994,886.38	0.220
333332000000333334666667	2,432,537.27	0.294
594925356096913903030500 7159999	35,172,885.16	5.141
3192590766338586965010359379 8951381	125,016,354.66	38.419
104926970397646886106311393667715507	148,326,614.71	33.977
83893153674119714337771753095846831 42183	714,012,950.71	152.361
631832938285080690492549937063909265742121 79729	5,662,176,856.17	623.127

As shown by the graph below, the empirical runtime of our program aligns with the theoretical complexity of the quadratic sieve.

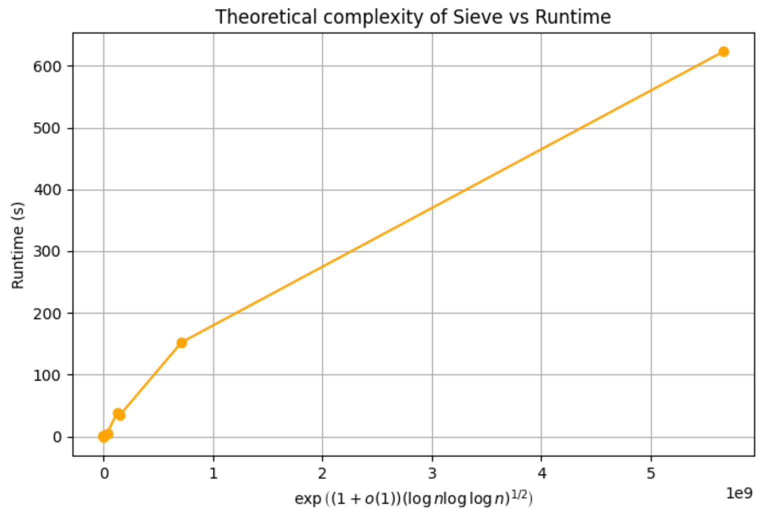

Figure 1: Empirical runtime (s) against theoretical complexity as n grows. The runtime grows linearly against our heuristic, which is what we would expect.

As shown by the graph, the empirical runtime has a near linear relationship with the runtime complexity detailed by Pomerance's paper. As such, our program aligns with the theoretical complexity of the quadratic sieve.

References

[1] Carl Pomerance. Smooth numbers and the quadratic sieve. In *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography*, volume 44, pages 69–81. Cambridge University Press, 2008.

[2] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.