Implementing an Optimized Quadratic Sieve Algorithm for the Factoring of Large Integers

Keith Cressman, Aakash Kothapally, Colin Zhu

April 25, 2023

1 Introduction

For an extremely large number, finding its factorization is a challenging computational task that can often serve a valuable role in cryptography. For instance, the security of the RSA algorithm depends on the difficulty of factorizing extremely large numbers. The simplest possible solution for factoring a number n, often referred to as trial division, is to check divisibility against all primes up to $\sqrt{n}$, but this becomes an exceptionally slow process as n grows. The quadratic sieve addresses this issue of computational scale when factoring large numbers. In this paper, we present our implementation of the quadratic sieve, which includes additional optimizations to the traditional algorithm. We also provide the results of running our algorithm on various 20 to 49 digit integers. Additionally, we discuss the value of our optimizations in speeding up the algorithm's runtime and ensuring an accurate result.

2 Quadratic Sieve Overview

The main idea for the quadratic sieve comes from the basic principle [5].

Basic Principle. *Let n be an integer and suppose there exists integers x and y with $x^2 \equiv y^2$ mod n but $x \not\equiv \pm y$ mod n. Then n is composite. Moreover, $gcd(x - y, n)$ gives a nontrivial factor of n.*

The backing for the basic principle comes from the notion that given we have x and y that satisfies the above equations, then $x^2 - y^2$, which can be rewritten as $(x + y)(x - y)$, will be divisible by n without either $x + y$ or $x - y$ being divisible by n. This ensures that we can use $x - y$ to identify a nontrivial

factor contained of n via the gcd function.

The immediate question is how to find the right values for x and y. A naive method would be to evaluate x^2 mod n for $x = \lceil \sqrt{n} \rceil$, $\lceil \sqrt{n} \rceil + 1$, $\lceil \sqrt{n} \rceil + 2$, and so on, stopping once we get a perfect square for some value (assuming we find one, then for some $y \in \mathbb{Z}$, we can say $x^2 \equiv y^2$ mod n, while $x \neq \pm y$ mod n, satisfying the conditions of the basic principle). But it is really rare to find the right y^2 by simply iterating over the possible x values. Instead, we can consider the multiplication of multiple x values. As an example, assume x_1 and x_2 are potential values of x. If for $x_1^2 \equiv y_1$ mod n and $x_2^2 \equiv y_2$ mod n, we find that $y_1 y_2$ mod n is a perfect square, then $x^2 = (x_1 x_2)^2$ and $y^2 = y_1 y_2$ satisfy the conditions of the basic principle. It will be much quicker to find the x value by seeing when multiplied values work, since it avoids having to iterate one by one until we get to that x value.

This resolves some issues, but it leaves two key questions unanswered. How do we know when we will have enough values such that we can multiply some particular subset of the values and yield the right x value? And given we know we have enough values, how do we know which subset to pick?

We can answer the first question by using the concept of smoothness. We say a number is B-smooth if all of its prime factors are $\leq B$. We can notice that if a number is not B-smooth, given B is set high enough, it becomes rather unlikely that we can use it to form our multiplication (if we say this number has some factor $p > B$, we would need another term to also contain the factor p in order for it to be a perfect square, but multiples of p will be far apart given the size of p). Conversely, we can notice that if a number is B-smooth, it could potentially be useful for yielding a perfect square. In fact, we can rely on the following lemma for even stronger guidance [2].

Lemma. *If m_1, m_2, ..., m_k are positive B-smooth integers, and $k > \pi(B)$ (where $\pi(B)$ denotes the number of primes in the interval [1, B]), then some non-empty subsequence of (m_i) has product a square.*

So for a certain B, if we know that $\pi(B)$ primes exist from 1 to B, we just need to select at least $\pi(B) + 1$ numbers in a set X such that for all $x_i \in X$, x_i^2 mod n is B-smooth. Then subset of these numbers in X will multiply to form a perfect square.

Now, we can address the second question, how to select the right subset, by using the concept of Gaussian elimination. We know that ultimately all of the numbers in X are made up of prime factors, so to determine when a subset will product a square, we just need to check when their product yields prime

factors are raised to even powers, or exponents of 0 mod 2. So we can form vectors based on the exponents of each number's prime factors (so if we set B as 5, then the prime factors can only be 2, 3, or 5, and given $x_i = 2_1^e * 3_2^e * 5_3^e$, the vector is (e_1, e_2, e_3)). Then, we form a matrix of these values mod 2. If by adding a set of rows together, they all sum to 0 mod 2, then the numbers associated with each of the rows serves as the correct subset. But this is equivalent to finding a set of linearly dependent row vectors via Gaussian elimination [2]. So we can simply perform Gaussian elimination to identify the correct numbers.

We can list the basic steps of the quadratic sieve in order. First, we select B. Then, we add values of x^2 mod n to our set that are B-smooth until we get at least $\pi(B) + 1$ values (the ideal number needed is discussed later in this paper). Then, we use Gaussian elimination to find the right subset of values that has product a square. If we use the set of numbers x_1, x_2, ..., x_k, where the product of $y_1 = x_1^2$ mod n, $y_2 = x_2^2$ mod n, ..., $y_k = x_k^2$ mod n is a square, then in terms of the basic principle, we have $x = x_1 x_2 \dots x_k$ and $y^2 = y_2 y_2 \dots y_k$. We can find y by performing modular arithmetic. Then, if $x \neq \pm y$ mod n, we can identify gcd(x-y, n) as a factor of n. If $x \neq \pm y$ mod n, we cannot identify a non-trivial factor, and we go back to look for another subset of B-smooth values.

Now, we have provided a full understanding of the tools used by the quadratic sieve method. A more thorough discussion of how we implement the quadratic sieve (with additional optimizations) will be provided in the algorithm design section.

3 Algorithm Design

Our algorithm uses many of the core tools discussed in the previous section, but there are certain details that do not have clear inputs, such as the specification of the number B and a method for examining various numbers for B-smoothness. We will describe our algorithm and specifically note when we are making specific choices beyond the scope of the previous section.

First, we check if n is prime by running the Miller Rabin primality test with ten different values of a. A prime number will never be identified by Miller-Rabin as composite, but a composite number could potentially be declared a prime. For a single value of a, the test fails with probability 10^{-5} [2]. Thus, with ten repeated trials, the probability that a composite number would go

undetected is less than 10^{-50}. Once we are confident that n is composite, the quadratic sieve can begin.

3.1 Determining the Factor Base

The first step in the quadratic sieve algorithm is to choose a value for B. We relied on an estimate declaring that when we optimize for speed in factoring n, we should select $B = e^{(\frac{1}{2}+\epsilon)(\log n \log\log n)^{1/2}}$ [2]. The choice for ϵ is open for interpretation, but for our purposes, we found that setting it to just 0.005 yielded sufficient results for factoring numbers with 20 to 49 digits. We will have a deeper discussion regarding our choice in ϵ in the results section.

The next step is to find the factor base (the primes below B that we use as factors for prospective x values). One additional optimization we can take is the use of the Legendre symbol. The Legendre symbol as a function of n and prime p is 1 if n is a quadratic residue modulo p, meaning that n is congruent to some perfect square mod p, and it is also not divisible by p. This ensures we avoid the problem of needing a multiple of p in any other term used in the multiplication to get the final x value. We also have the advantage of utilizing Euler's criterion, which states that the Legendre symbol of $(\frac{a}{p}) \equiv a^{\frac{p-1}{2}}$ mod p. Thus we limit our factor base not only to primes, but specifically to the primes that give the required quadratic residue.

With a factor base containing $\pi(B)$ primes, at least $\pi(B) + 1$ primes are needed to guarantee that we will be able to find $x^2 \equiv y^2 \ mod \ n$ for some x and y. However, it is quite possible that this will involve $x \equiv \pm y \ mod \ n$, thus resulting in only a trivial factor being found. To make it more probable that a nontrivial factor will be found, more B-smooth numbers should be found. There is a tradeoff here: the more B-smooth numbers we need to find, the longer it will take, but the more likely it is that we will find a nontrivial factor. Considering all of the numbers tested here, we found that $\pi(B) + 20 + ln(B)$ was a good number of smooth numbers to find. Admittedly, this slowed things down significantly for values of n below around 10^{20}, but was important in guaranteeing a solution for the larger values of n.

3.2 Finding B-Smooth Numbers

Now that we have our prime factors for our factor base, we have to determine the B-smooth numbers. For values of n less than around 10^{15}, it is efficient to simply do a brute force search through all numbers x, starting at $\lceil\sqrt{n}\rceil$,

checking if all the factors of x^2 *mod* n are in the factor base. For larger values of n, the following, more efficient approach is used.

First, we utilize the Tonelli-Shanks Algorithm to find solutions to $x^2 - n \equiv 0$ *mod* p for each p in the factor base [3]. We used this algorithm because after a review of the relevant literature, it appears the Tonelli-Shanks algorithm is a relatively common method of computing square roots modulo a prime number that tends to be among the fastest over a wide range of primes [4]. Then, starting at $x = \lceil \sqrt{n} \rceil$, initialize a register with the value of $x^2 - n$ for many values of x greater than this starting value. For large enough values of n, it is guaranteed that $x^2 - n$ will be equivalent to x^2 *mod* n for all values of x that can be tested in a reasonable amount of time, incrementing by one each time. Thus, if any of these values have all prime factors within the factor base, they will be of use. Next, for each prime p in the factor base, the Tonelli-Shanks Algorithm gives two solutions to $x^2 - n \equiv 0$ *mod* p. So, for each solution x, we then divide the registers corresponding to x and divide by p until they are no longer divisible by p. At the end of this process, after iterating through all primes in the factor base, the registers with a value of 1 remaining indicate that the original value was a product of primes in the factor base, giving a number x such that x^2 *mod* n is a product of primes in the factor base.

A few side notes about the process above: 1) For any solution to the Tonelli-Shanks Algorithm, the solution plus any multiple of p is also a solution. Some limit must be placed on the number of solutions considered at one time, as there an infinite number of potential solutions. As such, our algorithm only looks at a fixed length number of possibilities for x at a time, moving on to the next set of possibilities after all B-smooth values have been discovered in the previous set. The first idea for this length was to use 10 * the largest prime in the factor base. Another idea was to exploit the computer's cache size and choose an array such that all possibilities could be stored in the cache at once. Now, consider a few data structure sizes. The laptop used by one of the authors (Keith) has around 8MB of cache memory. Each element in the array stores, for some x, the number $x^2 - n$, which is on the order of n. So, for n up to 37448430805296159090191815103305542055009260219 47 (the largest number tested) it stores a number somewhere from 0 to 2^{162}; in Python, this takes up to 32 bytes to store. Assuming 32 bytes per entry in the array, and a cache size of $8MB$, we get an array of length $250,000$, or rather something smaller like $100,000$ since the computer has lots of other things to store in the cache, too. In practice, however, this cache-optimized approach was no faster than just using 10 * the largest prime in the factor base: a length of around one million elements for the largest n.

As another side note, for large values of n, it is less efficient to initialize each register as $x^2 - n$ and divide by all the prime factors. It is more efficient to initialize the register as $log(x^2 - n)$ and subtract $log(p)$ once, even if p divides $x^2 - n$ several times. Clearly, the register will not go to zero if it has non-distinct prime factors. Thus, if the register gets below the log of 1 plus the largest prime in the factor base, we check manually if it is B-smooth by brute force. In practice, this approach (subtracting logs instead of dividing integers) sped up the process of finding B-smooth numbers by about 50%. However, perhaps due to some oversight in the implementation, it did not always result in nontrivial factors being found, whereas the other approach always did. For instance, the 49 digit number 3744843080529615909019181510330554205500926021947 could not be factored using the implementation with subtracting logs, but could be factored the other way, although the entire process took about 50% longer. A 50% speed up is significant, but was deemed not worthwhile, considering the downsides.

3.3 Using the B-Smooth Numbers to Factor n

The next step is to determine which subset of these numbers $x_i \in X$ can be multiplied together such that them squared mod n is a perfect square. As explained in the previous section, this is motivated by ensuring any prime factor that appears in the multiplication appears an even number of times. Thus, we need to form a matrix where each row represents a specific number's exponents for each factor in the factor base. When we find a combination of rows that summed together yield even exponents for all factors, it might be a possible product for a perfect square. Again, as explained in the previous section, this process can be done by setting the matrix as mod 2, then identifying a set of linearly dependent row vectors via Gaussian elimination (since we would want each exponent among all rows to sum to 0 mod 2 in this case).

So we actually build the matrix using our factor base and our set of B-smooth numbers. We made sure to store the factors for each of the B-smooth numbers, so this is simply a process of placing the factor counts into the correct entries. We then compute the residue of the entire matrix mod 2 at the very end.

Next, we perform Gaussian elimination on this matrix. We use the typical process of Gaussian elimination, performing the necessary elementary operations (additions, multiplications, swaps) to place the matrix in row echelon form. At this point, we can use that to know the correct vectors that form the left nullspace.

This left nullspace allows us to find x and y such that $x^2 \equiv y^2$ mod n, since as stated before when we have our exponents sum to be 0 mod 2, we have a perfect square to use.

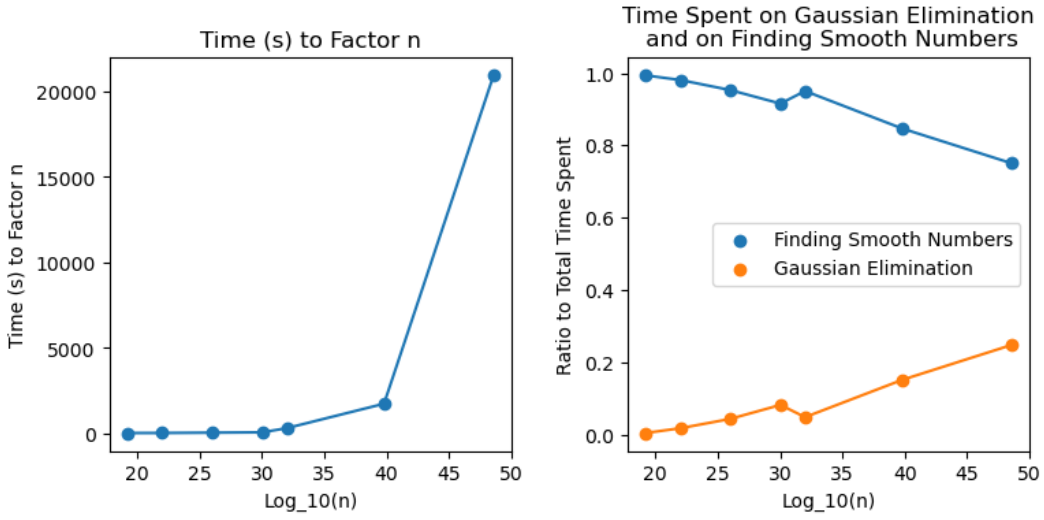
Finally, we can use our subset of numbers to potentially factor n. No matter what, we know $x^2 \equiv y^2$ mod n. But as stated in the Basic Principle, it is necessary that $x \not\equiv \pm y$ mod n in order for a nontrivial factor to be found. If $x \equiv \pm y$ mod n, then our factors would just be 1 and n, which isn't very useful. In this case, we go back to finding a new set of linearly dependent vectors via Gaussian elimination, then solving for x and y, and checking again.

Eventually, when $x \not\equiv \pm y$ mod n, then we will get some nontrivial factor by specifically taking gcd(x-y, n). We can obtain the other factor(s) by dividing n by gcd(x-y, n) (although we also know that it would be gcd(x+y, n)). This factorization of n satisfies the purpose of the quadratic sieve.

4 Results

4.1 Times for Factoring N

Plotted below are the runtimes for correctly factoring integers of varying sizes. In each case, two primes with similar magnitude were randomly chosen and then multiplied together to find n. The largest n successfully tested had 49 digits, and took just under six hours to factor.



Most of the runtime is taken up by finding B-smooth numbers. With that

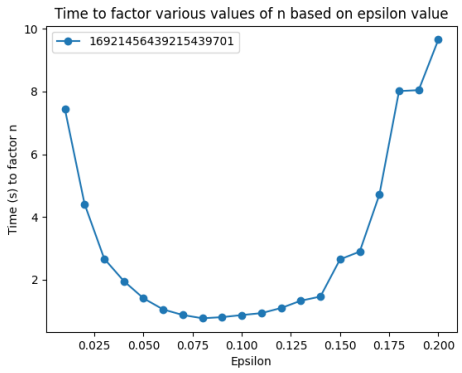
being said, for large values of n, the Gaussian Elimination step also takes a significant amount of time.

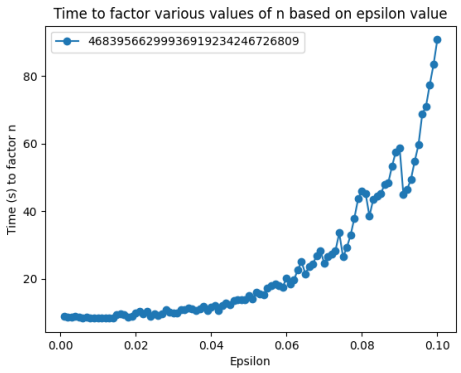
One of the parameters in the algorithm is ϵ, which controls the size of the factor base. The upper bound B is chosen as $B = e^{(\frac{1}{2}+\epsilon(\log n \log\log n)^{\frac{1}{2}}}$. One consideration here is speed, but also correctness. For values of ϵ very close to 0, the algorithm rarely ends up finding nontrivial factors for many values of n. When ϵ is too small, the congruences $x^2 \equiv y^2 \ mod \ n$ tended to all have $x \equiv y \ mod \ n$, and so only trivial factors were found. By increasing the size of the factor base, these trivial factors are found with lower frequency, allowing for the nontrivial factors to be discovered. However, this comes at a cost. The number of B smooth numbers found is at least the number of the primes in the factor base, so more computation needs to be done to find more B smooth numbers. It also results in a larger matrix for Gaussian Elimination, which adds to the runtime. For the largest numbers tested, up to the 49 digit number, $\epsilon = 0.005$ had a good combination of efficiency and correctness. Our algorithm recursively retries again with ϵ multiplied by ten if it fails to find a nontrivial factor with the old ϵ.

.

4.2 ϵ Parameter Tuning

As alluded to earlier, we spent a lot of time considering various values of ϵ to optimize runtime while ensuring we could always factor N. Ultimately, we recognized that it is impossible to select a single value for ϵ that factors N at the quickest pace for all numbers, as can be seen in our plotted results.





For numbers around 10^{20} or lower, such as the number on the left, larger values

of ϵ around $\epsilon = 0.1$ result in the fastest running times. However, for larger values of n, such as the number on the right, smaller values of ϵ are needed to keep the Gaussian Elimination from taking extremely long due to the size of the matrix. We see that a value around $\epsilon = 0.005$ is best for our larger numbers (it seems to be close to optimal for this number, and it also works well with larger numbers). This idea that ϵ should decrease as we increase the size of N works well with the intention of ϵ in the quadratic sieve, as in the original paper it is referred to as $o(1)$, meant to signify a function that tends towards 0 with larger and larger inputs [2]. Via experimentation, we are obtaining the result implied by the theoretical derivation of the quadratic sieve.

5 Potential Improvements

One improvement would be to convert the code to run in C instead of Python. Since C is compiled instead of interpreted, it runs faster. The speed-up factor in going from Python to C is expected to be around 10x to 100x.

We also recognize the limitations of our results for ϵ parameter tuning. Although we recognize that smaller values of ϵ are better for larger numbers, and via experimentation, we observed that an $\epsilon = 0.005$ worked best for a particularly large number with 49 digits, it could be possible to derive a more intuitive understanding of ϵ. Perhaps with more experimentation, or a stronger mathematical intuition in constructing the formula for the upper bound of B, we could do a better job at selecting the right ϵ to quickly factorize a large number.

In all cases tested here (up to a 49 digit number), finding smooth numbers always took longer than the Gaussian Elimination step. However, for larger numbers, it is possible that the Gaussian Elimination would take longer and become the main bottleneck. The matrix used in the quadratic sieve has $\pi(B)$ columns, and slightly more rows than columns, where $\pi(B)$ is the number of primes less than or equal to B. Approximating the matrix as being square, the simple Gaussian elimination algorithm has runtime complexity of $O(\pi(B)^3)$. However, faster implementations exist which would reduce the runtime complexity to only $O(\pi(B)^2)$ [1].

6 Conclusion

Our implementation of the quadratic sieve is successful at factoring numbers up to the 49 digit number provided in the project description. We also considered various optimizations to improve the sieve's runtime. First, we provide an initial check if a number is prime, using the strongly-accurate Miller-Rabin primality test to provide a quick result for potential primes. Also, we implement the Tonelli-Shanks algorithm to quickly find the square root of a number mod a prime. Finally, we considered various values for the ϵ parameter in the algorithm to help it return a value at the optimal time.

However, we are aware that our implementation could face greater problems as the numbers scale up. This implementation of the quadratic sieve has proven successful in factoring numbers up to 49 digits. The largest number tested was 3744843080529615909019181510330554205500926021947, requiring just under six hours to factor. Given the trend in runtime, numbers with a few more digits could probably be factored within a few days of computation. While these results are strong, they are nowhere near fast enough to attack a very advanced form of RSA, the security of which relies on the difficulty of factoring a key with hundreds of digits.

7 References

[1] Cetin K Koç and Sarath N Arachchige. "A Fast Algorithm for Gaussian Elimination over GF (2) and its Implementation on the GAPP". In: *Journal of Parallel and Distributed Computing* 13.1 (1991), pp. 118–122.

[2] Carl Pomerance. "Smooth numbers and the quadratic sieve". In: (Jan. 2008).

[3] Daniel Shanks. "Five number-theoretic algorithms". In: *Proceedings of the Second Manitoba Conference on Numerical Mathematics (Winnipeg), 1973*. 1973.

[4] Gonzalo Tornaría. "Square roots modulo p". In: *LATIN 2002: Theoretical Informatics: 5th Latin American Symposium Cancun, Mexico, April 3–6, 2002 Proceedings 5*. Springer. 2002, pp. 430–434.

[5] W. Trappe and L.C. Washington. *Introduction to Cryptography: With Coding Theory*. Pearson Prentice Hall, 2006. ISBN: 9780131862395.