Math 404 Computing Project Paper

Louis Hu, Herbert Wang, Ryan Champaigne

April 2022

1 Introduction

Often, cryptographers search for "one-way" functions–functions where the output is easy to compute for a given input, but computing the input given it's output is hard. The existence of such functions has not been proved, however certain functions have been practically used as "one-way" functions for decades and have withstood all attacks.

If one is trying to find a number n which in practice is difficult to factor, they may have a few strategies. First, they may construct n as large as possible. As n grows larger we expect n to become some amount harder to factor. Let us ask then, for a given bound on the size of n are certain numbers harder to factor then others? (this is practical as well, if you plan on working with n in some way). Here, we should note that checking n for divisibility by any arbitrary number a can be done very cheaply. Therefore, if factors of n can be guessed, or easily checked as is the case for small primes, then n can be factored rather easily. In the case that a factor of n is found this either factors n or reduces n greatly. Thus, one should choose n with factors p_i which are relatively large and difficult to guess or come across by chance. Maximizing these two principles, one could try an n which has only 2 large randomly chosen primes, p_1 and p_2. Indeed in practice, numbers constructed in this way are relatively difficult to factor and we will assume that n takes this form for the remainder of the paper.

2 Brief History of Quadratic Sieve

The Quadratic Sieve was created in 1981 by Carl Pomerance [6] as an improvement on the Linear Sieve created by Schroeppel. Both factoring systems are built on Fermat's difference of squares technique as well as Kraitchik's improvement on this technique, which we will discuss later. Pomerance greatly improved the efficiency of current algorithms introducing a sieving process, similar to the sieve of Eratosthenes, that could decompose many numbers into their small prime factors at once. For many years, the quadratic sieve remained the most efficient algorithm to factor large numbers which are otherwise difficult to factor. Here, we will give a basic overview of the Quadratic Sieve algorithm, our implementation of the algorithm, as well as an analysis of its efficiency, and the optimizations that were implemented.

3 Quadratic Sieve Procedure– Theory

The basic principle behind Quadratic Sieve is if $x \not\equiv \pm y$ but $x^2 \equiv y^2 \pmod{n}$, then n is composite and $gcd(x-y,n)$ is a nontrivial factor.

The idea of Quadratic Sieve is to find such an x and y. The methodology behind this is to take numbers $a_0, a_1, ...$, compute $a_0^2, a_1^2, ..., \pmod{n}$, then combine them in some way to get squares. Since if we have a congruence, then

$$a_{i_0}^2 a_{i_1}^2 ... a_{i_k}^2 \equiv (a_{i_0} a_{i_1} ... a_{i_k})^2$$

And thus if all conditions above hold, then we have factors of n.

3.1 Choosing a Factor Base B and Sieve Interval I

With the above in mind, we need a way to choose the a_i and to get a product of a_i's to be a square. In order to choose these values smart, assuming n is the integer we want to factor, let us define $f(x) = x^2 - n \equiv y_i \pmod{n}$. So now our problem is now redefined as

$$y_{i_1} y_{i_2} ... y_{i_k} \equiv (x_{i_1} x_{i_2} ... x_{i_k})^2 \pmod{n}$$

Now, in order to check if $y_{i_1} y_{i_2} ... y_{i_k}$ is a square, we want to check if the exponents of the prime factors of the product are even. Thus, in order to continue, we need to factor each y_i. In order to make this computationally feasible, we want to factor over a small fixed set of prime factors. We set this limit as B and all the primes less than B plus -1 will be our "factor base". We can find these primes efficiently through the Sieve of Eratosthenes When we have a number that factors into primes less than B, we say that number is B-smooth. To make y_i small, we want to select x values that are close to 0; therefore, in our procedure, we set a limit from $[\lfloor\sqrt{n}\rfloor - I, \lfloor\sqrt{n}\rfloor + I]$ where I is our sieving bound, where we consider x values only in this range and get a value close to 0. Then, now if $r = \lfloor\sqrt{n}\rfloor + k$, where $k \in \mathbb{Z}$. is in our sieving interval, then we are looking for factors p such that

$$n \equiv r^2 \pmod{p}$$

Since n is a quadratic residue $\pmod{p}$, the primes we need to only consider are those that satisfy the Legendre symbol

$$\left(\frac{n}{p}\right) = 1$$

3.2 Sieving

Once we have picked factor base bound B and and interval I, we start looking at numbers x in our sieving interval and calculate y_i and check if it is B-smooth. One naive method to check if a number is B-smooth is to check if each prime in our factor base divides y_i. However, this is incredibly inefficient. A much more efficient method is a sieving method described by Pomerance. We will be working on the entire sieving interval at once.

If p is a prime factor of $f(x)$ described above, then $p|f(x+p)$. Conversely, if $x \equiv y \pmod{p}$, $f(x) \equiv f(y) \pmod{p}$. Thus for each prime p in each factor base, we solve

$$f(x) \equiv s^2 \equiv 0 \pmod{p}, x \in \mathbf{Z}_p$$

This we solve using Tonelli-Shanks Algorithm. Then we will obtain two solutions: s_{1p} and s_{2p}. Then $f(x_i)$ with x_i in the sieving interval will be divisible by p when $x_i = s_{1p} + pk$ or $x_i = s_{2p} + pk$, where $k \in \mathbb{Z}$.

Next, first we create an array beforehand, containing all the values of $f(x_i)$ for each x_i in the interval. Then for each p in our factor base greater than 2, we start at s_{1p} and s_{2p} and divide out the highest power of p for each term in the array in arithmetic progression. Once we have completed the sieving for each prime p in our factor base, we can then scan the array for any values equal to 1 or -1 (cases where -1 is in the factor base). Those array elements are $f(x_i)$'s that factor completely over the factor base (B-smooth numbers). Once we collect enough B-smooth numbers (the number of primes in the factor base plus one in order to achieve a linear dependence in the exponent vector matrix), we can continue on with the process to build the exponent vector matrix. Note that Tonelli-Shanks algorithm does not work for $p = 2$, in this case, we will do this special case separately and find the first term $f(x_r)$ in the array that is divisible by 2, and then divide out all the powers of 2 in $f(x_r)$, then continue to divide out every other term since then every other term will be divisible by 2 by construction of our interval.

3.3 Constructing Exponent Vector Matrix

If $f(x)$ is B-smooth, then we construct a vector containing the exponents $\pmod{2}$ of the primes in the factor base as described above. Then after constructing enough vectors (we will specify "enough" later), then we put all the vectors together into a matrix A. As a result, the rows of matrix A will be all the B-smooth $f(x_i)$'s and the columns will be the exponents $\pmod{2}$ of the primes in the factor base. For example, if the factor base is $\{-1, 2, 3, 7, 11\}$, and $f(x) = 2*7*11^2$, then the row corresponding to this $f(x)$ would be $(0, 1, 0, 1, 0)$.

Since we want $f_(x_i)$'s to multiply to a perfect square, then we want to find a set of rows that add up to 0 in all the entries since that is equivalent to having even exponents for all the primes in the factor base, which means that we can factor a 2 out of the exponents and thus it is a perfect square. So to generalize this problem to the matrix, given $f(x_1), f(x_2), ..., f(x_k)$ which represent the k rows of the matrix A, we want to find solutions to

$$f(x_1)e_1 + f(x_2)e_2 + ... + f(x_k)e_k \equiv 0 \pmod 2 \Rightarrow A\mathbf{e} = \mathbf{0}$$

where $\mathbf{e} = (e_1, e_2, ..., e_k)$. This is equivalent to finding the null-space of matrix A, thus, via Gaussian elimination, we can solve for e. From this knowledge, we want to construct the matrix A to have as many rows as primes in the factor base. Since we have more vectors than the dimension of the vector space (the size of the factor base), we know there must be a linear dependence. Thus, then we simply check the solution vectors to see if the corresponding products of $f(x_i)$'s and x_i yield a valid non-trivial factor of n by doing a GCD calculation through the Euclidean algorithm as described before. Since we are factoring an RSA modulus, we are finished.

3.4 Gaussian Elimination

One critical step to the Quadratic Sieve procedure is to find the solution vector to the system described above. There are several ways to do this, Gaussian elimination being the most common method. The standard run time of Gaussian elimination is $O(B^3)$, but since the matrix we are looking at is sparse and mod2, there are two better algorithms out there we could have applied: block Wiedemann and block Lanczos. The first one is block Wiedemann algorithm [8]. This run time is approximately $O(B(\omega + Bln(B)ln(ln(B)))$, where B is the number of primes in the factor base and ω is approximately the number of field operations required to multiply the matrix to a vector. Since the matrix is sparse, the idea is that ω is low, which will decrease the run time significantly compared to the naive standard Gaussian Elimination. However, the bottleneck in the run time is often the sieving part [6]. Thus, we chose to go with this modified Gaussian Elimination algorithm for our algorithm [ç$_a rachchige_1 991$]. We found later empirically that the algorithm was effective and was never the bottleneck.

4 Quadratic Sieve Procedure– Algorithm Implementation

Based on the theory above, we decided to go through and implement this version of the Quadratic Sieve in Python as described below. While modifications to this algorithm can be used to factor arbitrary n, as described above, we will assume the number n we are factoring is a product of two large primes p and q.

4.1 Input and Output

In practice, numbers of this form are used in the RSA cryptosystem which we will further describe later. The input for our QS function will be a large integer

n, a bound B as described above in the theory, and a sieve interval I, also described above.

4.2 Choice of B and I

We choose our B according to the paper our professor gave us [6].

$$B = exp((1/2 + o(1))(lognloglogn)^{1/2})$$

In the paper, they prove how for a sieve interval $I = n^{1/2+o(1)}$, this was the optimal bound, and the number of steps to do the sieving to find the requisite number of B-smooth polynomial values would be $exp((1+o(1))(lognloglogn)^{1/2})$ [2]. We chose $o(1) = .08$, multiplied by 1.05, and added a constant of 4000.

Because the I values were too large based on the bounds of the aforementioned paper, we chose to use the bound of another paper by Eric Landquist [4].

$$I = (exp((lognloglogn)^{1/2}))^{3\sqrt{2}/4}$$

For I, we multiplied by .000027 and added a constant of 60000.

4.3 Factor Base Generation

To generate all the primes for our factor base, we will use the Sieve of Eratosthenes. This algorithm time complexity is O(B(log log B)), so it is very efficient. In addition, we chose primes for which n is a quadratic residue mod p. This is explained above in the procedure. We can easily check this through a legendre function and seeing if it returns 1. This check considerably speeds up the sieving process by reducing number of total primes that need to be checked.

4.4 Sieving

To find the requisite number of B-smooth numbers (described in the theory section), we first create a "sieve-array", a Python list of $x^2 - N$ of x values from $sqrtn - I$ to $sqrtn + I$. These values should be close to n, so they should have a higher likelihood of being $B - smooth$. We then will loop through all the primes in the factor base, and generate residues r_1, r_2 using Tonelli-Shanks. Say we are looking at prime p currently in the loop, we then loop through every pth index starting from $(r_1 - \sqrt{n} + I) \pmod{p}$ to end of sieve-array, and dividing out all the powers of p. We also need to go in the negative direction and so we go through every pth index starting from $(r_1 - \sqrt{n} + I) \pmod{p} + I$ to 0, and again divide out all the powers of p. We repeat this process with r_2. Since Tonelli-Shanks algorithm cannot be applied to $p = 2$, we do this separately. We check for the first entry in the sieve-array divisible by 2. Once we find that index, we then start from there, divide out all the powers of two, and go to the index after the next index. We will continue going through every other term in the sieve-array and dividing out all powers of two.

After going through all the primes, we can go through the sieve-array and check if sieve-array[i] equals 1 or -1; if so, then we can add it to a list of B-smooth numbers. We also want to keep track of the original x values, which will allow for smoother solving for factor later on. Once we have the requisite B-smooth numbers, then we can break and return.

4.5 Further Speedups

Much of our speed optimizations were accomplished using a speed up to the actual language we coded in. Since Python is a high level language, it runs much slower compared to a lower level C++ or even java. Thus, we chose to run the sieving and finding B-smooth numbers in Python 3.9.7. Then, we ran matrix construction, Gaussian elimination, and factor calculation in PyPy 3.9. In order to connect the two interpreters into one runtime, we used the pickl library to write the factor list and B-smooth numbers found by Python 3.9.7 to a file, which was then read by the PyPy program. We did not run our whole program in PyPy because PyPy performs slowly to construct arrays and perform calculations to compute the B-smooth numbers.

Other helper functions were also rewritten. Since PyPy 3.9 does not support the math module isqrt, the C-version from Mark Dickinson was used [1].

Lastly, we optimized our bounds considering our code's advantages and disadvantages.

5 Performance

For performance, we tested with the first 3 numbers in the coding assignment, $n_1 = 16921456439215439701$, $n_2 = 46839566299936919234246726809$, $n_3 = 6172835808641975203638304919691358469663$. We also constructed 3 more test numbers: $n_4 = 112244567799988876543321$, $n_5 = 2895899311081231989935403230973937$, $n_6 = 1270371939018099737770439595405348523$.

n	n$_1$	n$_2$	n$_3$	n$_4$	n$_5$	n$_6$
Sieving Run Time	0.018	1.289	242.2	0.190	7.429	31.23
GE Run Time	0.020	0.756	201.2	0.230	6.429	26.53
Total Run Time	.041	2.32	471.8	0.489	14.89	63.74

To constructing our function to generate B and I, $F(N) = B, I$, against the optimal B and I values that we found for our program, we performed a regression. As illustrated by the figures.

Figure 1: Graph of a regression performed on a, b on $B(N) = exp((1/2 + o(1))(lognloglogn)^{1/2})$. x-axis: ln(N) vs y-axis: B values.

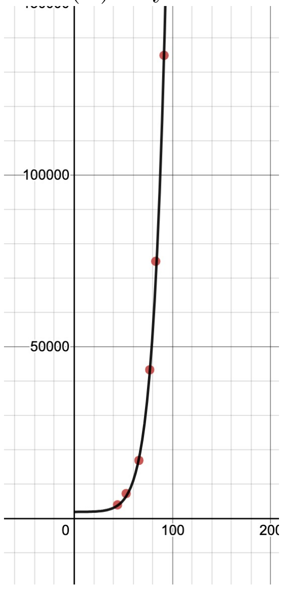
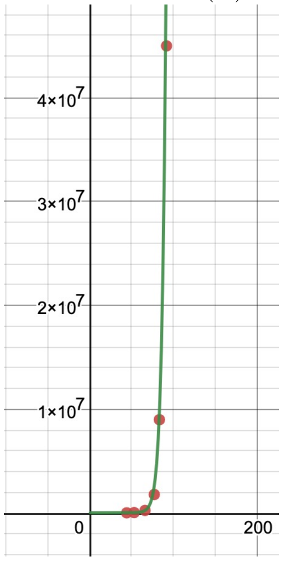


Figure 2: Graph of a regression of a, b on $I(N) = a(exp((lognloglogn)^{1/2}))^{3\sqrt{2}/4} + b$. x-axis: ln(N) vs y-axis: I values



6 Discussion

We were able to effectively factor the first 3 primes listed in the assignment and could most likely factor the fourth with enough time. With more time in the future however, we would like to have implemented the Multiple Polynomial Quadratic Sieve [7]. The use of multiple polynomials gives a better chance of factorization, and requires a shorter sieve interval. The use of multiple polynomials also allows for better applications of parallel processing.

Another obvious improvement is to translate this program into a lower level language such as C or Java. Since Python is an interpreted language, it amplifies the number of instructions required in order to perform a line of code. Thus if we switched to C, then we could get much higher speeds in terms of run time. It would also allow for more precise calculations with Big Integer library which is necessary for Multiple Polynomial Quadratic Sieve.

7 Applications to RSA

RSA is a public-key cryptosystem developed by Ron Rivest, Adi Shamir and Leonard Adleman in 1977. In public-key cryptosystems, users generate a public encryption key which can be used by anyone to encrypt their messages, which only the the creator of the public key can decrypt using the private decryption key. In RSA, message receivers must generate three values e,d and n, such that $(m^e)^d \equiv m \pmod{n}$ holds for all messages, with $m < n$, and with n equal to the product of two large randomly chosen primes, p and q. The pair (n, e) is released as a public key, while d, p, and q, are kept secret. The sender must calculate their message to the eth power mod n and send this value to the receiver. The receiver recovers the message by raising the value to the dth power mod n which is equal to m. The power of RSA relies on the practical impossibility of factoring n. If an attacker can factor n, they can easily generate the decryption key d from the public encryption key e by solving the equation $(ed \equiv 1 \pmod{()}p - 1)(q - 1)$. Therefore, if you can factor n you can also calculate d, and recover the message. Factoring algorithms such as the one presented in this paper are important for a few reasons; First, it is important to work on creating the best possible factoring algorithm to guarantee that cryptosystems are not vulnerable. That is, if they can withstand the attacks from the best factoring algorithms researchers currently have, they likely can withstand attacks from anybody. Second, mathematicians can give their best estimates of where factoring algorithms will be in the future to inform how secure cryptosytems will need to become. Lastly, factoring is itself a topic of study that has useful implications in number theory and other branches of mathematics. While the algorithms are practical in nature, they require strong mathematical theory to inform their creation.

8 Conclusion

In addition to the practical cryptographic applications, RSA spurred development of other public key cryptosystems as well as put pressure on creating better factorization algorithms. While the Quadratic Sieve is still the fastest algorithm for numbers of the form $n = pq$ for $n < 10^{100}$, other factoring algorithms have been developed that surpass the Quadratic Sieve for larger numbers. Namely, the General Number Field sieve has proved to be the best factoring algorithm for very large n. Additionally, the study of elliptic curves has produced both the Elliptic-Curve cryptosystem which can provide a similar security to RSA but with considerably smaller keys, and elliptic-curve factorization which is the fastest known algorithm for factoring randomly chosen large n which may not have exclusively large prime factors [5]. Optimization of all of these factorization methods as well as development of new public key cryptosystems remain a large open problem in cryptography.

9 Acknowledgments

We would like to thank our professor Dr. Samit Dasgupta.

References

[1] Mark Dickinson. isqrt.lean. `https://github.com/mdickinson/snippets/blob/master/proofs/isqrt/src/isqrt.lean`, 2019.

[2] RNDr Marian Kechlibar. The quadratic sieve - introduction to theory with regard to implementation issues. page 47.

[3] Çetin K. Koç and Sarath N. Arachchige. A fast algorithm for gaussian elimination over gf(2) and its implementation on the gapp. *Journal of Parallel and Distributed Computing*, 13(1):118–122, 1991.

[4] Eric Landquist. The quadratic sieve factoring algorithm. page 12.

[5] Carl Pomerance. A tale of two sieves. In Arthur Benjamin and Ezra Brown, editors, *Biscuits of Number Theory*, pages 85–104. American Mathematical Society.

[6] Carl Pomerance. Smooth numbers and the quadratic sieve. In *in [Buhler and Stevenhagen 2007]. Citations in this document: §5.* University Press, 2005.

[7] Robert D. Silverman. The multiple polynomial quadratic sieve. *Mathematics of Computation*, 48(177):329–339, 1987.

[8] D. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, 1986.