A Computational Approach to the Quadratic Sieve

WILL DENTON, AIDEN CROWE, and MAXWELL LIN, Duke University, USA

The quadratic sieve is a factoring algorithm that uses the properties of exponentiation over $(\mathbb{Z}/n\mathbb{Z})$ to discover the factors of a composite number n. The basic principle states that if $x^2 \equiv y^2 \pmod{n}$ and $x \not\equiv \pm y \pmod{n}$, then n must be composite and $\gcd(x - y, n)$ is a nontrivial factor of n (i.e., not 1 or n). The quadratic sieve algorithm uses a factor base for some bound B, consisting of all primes p such that $p \leq B$. Then, through trial and error, this factor base can be used to find an x, y that satisfies the above principle for a given composite number.

1 INTRODUCTION TO THE QUADRATIC SIEVE

The quadratic sieve algorithm [5] consists of four main steps. In implementation, many of these steps are broken down further, but the underlying structure is outlined below.

(1) Choose a smoothness bound B. This will be the upper bound on primes considered during the quadratic sieve algorithm. The factor base is the set of primes $\{ p \in \mathbb{Z} \mid p \leq B \}$ with size $\pi(B)$.

(2) Choose a random integer x and calculate $x^2 \pmod{n}$. If $x^2 \pmod{n}$ is B-smooth, meaning that $x^2 \equiv \prod p_i^{q_i}$ where all p_i are in the factor base, add x to a list.

(3) Once this list has $\pi(B) + 1$ elements, use linear algebra to find a product of the list elements such that all exponents of the prime factors are even. Call this product a^2.

(4) Finally, define b so that $b^2 = \prod p_i^{q_i}$ for all primes of each x_i used to construct a^2. If $a \not\equiv b \pmod{n}$, n is composite and $\gcd(a - b, n)$ is a factor of n. Otherwise, the process should be repeated for a different pair of a, b.

2 IMPLEMENTATION OF THE QUADRATIC SIEVE

2.1 Choosing the Bound B and the Sieve Size S

In addition to n, the number to be factored, we must choose two other parameters to initialize the quadratic sieve: B, the smoothness bound as defined above, and S, the size of the sieve we want to create. Both of these quantities scale with n, where higher values of B and S would mean more computations, but more rows in our matrix for the linear algebra step. If B and S are set too low, we might not be able to find $\pi(B) + 1$ elements for our matrix, in which case we would not be able to factor the number. Therefore, to optimize for efficiency, we set B and S by hand for each number we factor, following a process of trial and error.

Ideally, both of these parameters could be dynamically adjusted to optimize performance, or we could have created an equation to output these parameters based on the size of n, but that would have required additional experimentation. Instead, we chose to focus on the computing aspects of our program.

2.2 Finding Primes Less Than B

To find primes less than some bound B we can use a method called the Sieve of Eratosthenes. In order to create a list a primes we start with a list of all numbers less than B. Next, we pick the

Authors' address: Will Denton; Aiden Crowe; Maxwell Lin, Duke University, Durham, North Carolina, USA.

smallest prime, 2, and we mark 2 as prime and all numbers that are multiples of 2 as not prime $(4, 6, 8, \dots)$. We then pick the next smallest number that is prime, 3, and repeat the process. A small optimization we implemented is that we only need to check primes less than $\sqrt{B}$ as all primes greater than $\sqrt{B}$ will be marked as primes already. Additionally, all composite numbers c such that $c > \sqrt{B}$ must have a prime factorization that includes at least one prime $p \leq \sqrt{B}$.

2.3 Selecting the Factor Base

To improve on the quadratic sieve as outlined in Section 1, we implemented an optimization to find B-smooth factors faster than trial division. This optimization, which is often known as "sieving," is described throughout the rest of this section, and it imposes restrictions on our factor base. Specifically, this algorithm will require n to have a square root mod p for all of the primes in our factor base.

2.3.1 Quadratic Residue. To ensure that $\sqrt{n} \pmod{p}$ is well-defined, we will use the idea of a quadratic residue. An integer q is said to be a quadratic residue mod p if it satisfies the following equation for some integer x (meaning q is congruent to some square mod p) [4]:

$$x^2 \equiv q \pmod{p}. \tag{1}$$

Using this definition, we will select each prime p in our factor base so that n is a quadratic residue mod p.

For approximately half of the primes, n will not be a quadratic residue. To filter out such primes, we ensure that the following equation holds (the mathematical concepts to support this equation are included in Section 4.1):

$$n^{\frac{p-1}{2}} \equiv 1 \pmod{p}. \tag{2}$$

Once we have verified this for each prime, we can create our updated factor base of only primes such that n is a quadratic residue mod p.

2.4 Creating the Sieve

Next, we need to generate our list of terms that we will use to try to factor n (step 2 in the introduction). When we do this, it is important that the square we use to generate each term is larger than n, as otherwise we will gain no information. Therefore, for all x up to S (see Section 2.1), we construct the sieve V_x as

$$V_x = (f(1), f(2), \dots, f(S-1), f(S)) \text{ with } f(x) = \left(x + \left\lceil \sqrt{n} \right\rceil\right)^2 - n \tag{3}$$

This equation uses $(x + \lceil \sqrt{n} \rceil)^2$ as one of the squares we will eventually check, and, as $n = (\sqrt{n})^2 < (\lceil \sqrt{n} \rceil)^2 < (x + \lceil \sqrt{n} \rceil)^2$, this square is slightly greater than n for small x values. This ensures that $f(x)$ is small relative to n. We will eventually select for the elements of V_x that are B-smooth, a property that is more likely to occur for small values of $f(X)$. At larger S values, and subsequently larger x values, it becomes more efficient to find B-smooth numbers by increasing the size of the factor base instead of increasing the size of the sieve.

In our implementation, we found that it was faster to use the log method outlined in the textbook so instead we opted to make the sieve contain the log of the numbers. This log sieve uses the function

$$l(x) = \ln\left((x + \lceil \sqrt{n} \rceil)^2 - n\right). \tag{4}$$

This log method functions very similarly to the first sieve, but has the computational benefit of allowing us to do addition and subtraction operations in place of multiplication and division. Since these operations constitute the bulk of our computation, it is more efficient to calculate the logs one time as we create our sieve than to do repeated multiplications and divisions.

2.5 Sieve to find B-smooth Numbers

Once we have generated our sieve, we need to ensure each of the numbers within are B-smooth. To do this, we will first have to calculate the modular square root of n for each prime in our factor base. Since we ensured n is a quadratic residue mod p for each prime, we know that the modular square root exists, and to find it we use the Tonelli-Shanks algorithm.

2.5.1 Tonelli-Shanks. The algorithm begins by finding $Q, S \in \mathbb{Z}$ such that Q is odd where $p-1 = Q2^S$. Then, we find $z \in \mathbb{Z}$ where z is not a quadratic residue mod p using Euler's Criterion. Finally, we establish our loop variables, M, c, t, R so that $M \leftarrow S$, $c \leftarrow z^Q$, $t \leftarrow n^Q$, and $R \leftarrow n^{\frac{Q+1}{2}}$. The Tonelli-Shanks algorithm then proceeds as follows:

Algorithm 1 Tonelli-Shanks

```
1:  procedure TONELLI-SHANKS(n, z, Q, S, p)
2:      M ← S
3:      c ← z^Q (mod p)
4:      t ← n^Q (mod p)
5:      R ← n^((Q+1)/2) (mod p)
6:      loop
7:          if t = 0 then
8:              return 0
9:          else if t = 1 then
10:             return R
11:         t_i ← t
12:         for i ← 1 to M do
13:             t_i ← t_i^2 (mod p)
14:             if t_i ≡ 1 (mod p) then
15:                 b ← c^(2^(M−i−1))
16:                 M ← i
17:                 c ← b^2 (mod p)
18:                 t ← tb^2 (mod p)
19:                 R ← Rb (mod p)
```

The above process will return R such that $R^2 \equiv n \pmod{p}$, but $-R \pmod{p}$ is also a valid square root. If n is a known quadratic residue mod p, it is guaranteed to work, as shown in the proof in Section 4.2. Through each iteration of the loop on line 6, $R^2 \equiv tn \pmod{p}$, meaning that once $t \equiv 1 \pmod{p}$, then we have found $R = \sqrt{n}$.

Note, as we sieve to find B-smooth numbers, we will be using both the simple calculated square root of n (i.e., $\lceil \sqrt{n} \rceil$) and the modular square root of $n \pmod{p}$ obtained via the Tonelli-Shanks algorithm. For clarity, we will define $k = \lceil \sqrt{n} \rceil$ to be the constant calculated square root, and use $\sqrt{n}$ to denote the modular square root for a given prime, as this value will vary for each calculation.

Now, to find the entries in our sieve that are B-smooth, we check that they are divisible by each prime in our factor base with

$$(x+k)^2 - n \equiv 0 \pmod{p} \implies x \equiv \sqrt{n} - k \pmod{p}. \tag{5}$$

We can calculate x for each prime in our factor base, which will give the least number in our sieve such that $p \mid x$. Due to the construction of our sieve, p similarly divides the numbers in the sieve $x, x+p, x+2p, \ldots$. In the typical algorithm, we would then divide out each of these values in our list by p. However, as V_x contains log values, we compute $V_x(x_i) = V_x(x_i) - \ln(p)$ for $x_i = x, x+p, x+2p, \ldots$.

2.6 Create the Exponent Vector Matrix Mod 2

Now that we have performed the sieving operation, each entry in the sieve with any prime factors within our factor base has been reduced to only primes outside of our factor base. If any entry in V_x is B-smooth it will be equal to 1 in the non-log implementation, or $0 = \ln 1$ in the log implementation. All other entries can be discarded, as they are not B-smooth.

For the remaining entries, we then factor by trial division, and create a new vector mod 2 for each entry of the corresponding exponents for each prime in our factor base. All such vectors are concatenated vertically to form a 2-dimensional matrix mod 2, where a value of 1 corresponds to an odd exponent for that prime, and a value of 0 corresponds to an even exponent. By finding the linear dependencies of the rows of this matrix, we can create a list of squares whose prime factors mod n multiply to be a square.

2.7 Perform Row Reduction Mod 2 to Find Linear Dependencies

To perform Gaussian elimination mod 2, we use the algorithm of Koç and Arachchige [3]. For an $n \times m$ matrix, this Gaussian elimination algorithm requires $m^2n + m^2 - m$ bit operations.

Algorithm 2 Row reduction mod 2

```
for j = 1, 2, ..., m do
    Search for A_ij = 1 in column j
    if found then
        Mark row i
        for k = 1, 2, ..., j − 1, j + 1, ..., m do
            if A_ik = 1 then
                Add column j to column k (mod 2)
```

All *unmarked* rows are linear dependencies.

2.8 Use GCD to Find Factors

Now that we have a set of linear dependencies, we select a dependency, and take the product of each number in it and call this product a^2. Similarly, we can take the product of the prime factors for each element in the dependency and call this b^2. This process produces two values a and b where $a^2 \equiv b^2 \pmod{n}$ such that $a \not\equiv b \pmod{n}$. We can then use these values to calculate a factor of n. The basic principle states that $\gcd(a-b, n)$ is a factor of n, and to calculate this, the Euclidean algorithm can be used. If this factor is trivial (1 or n), the algorithm is repeated with a different value for a and b from our linear dependencies.

3 PERFORMANCE AND LANGUAGE

3.1 Programming Implementation

In our programs, we exactly implemented the math outlined above. For ease of programming, we first implemented the quadratic sieve in Python using the NumPy library. [1] To improve performance, we translated the Python program to C. [2] In our C implementation, we used GMP [1], a library for arbitrary precision arithmetic. This means that our program supports any choice of $n, B, S \in \mathbb{Z}^+$ assuming that you have enough time and computer memory. Since Python is an *interpreted* language, there is more overhead executing the Python code compared to executing the equivalent *compiled* C code. In our tests, the C implementation was roughly 50-100 times faster than the equivalent Python implementation.

We compiled the C executable using the `-Ofast` flag which enables extreme compiler optimizations which disregard strict standards compliance [2]. For example, the compiler can make aggressive assumptions about floating-point arithmetic that can lead to faster code with the risk of no longer conforming to IEEE standards. We did not find any negative effects from enabling these aggressive compiler optimizations.

3.2 Computational Fixes and Improvements

The main difference between our code and the math outlined above was fixing issues related to floating-point numbers. When the log of a large int is calculated with only 64 bits of precision, there is a lot of accumulated error in the result. This caused the sieve to not find some B-smooth numbers properly. Additionally, when calculating the log of numbers $> 10^{16}$, it could not calculate the log properly and the sieve could not factor numbers > 90 bits. We addressed this issue in the C code by doing the log calculations on 128-bit floats; however, this was not implemented in Python so we can only calculate up to 90-bit numbers in the Python version of the code.

After performance testing, we found that the `get_sieve_log` function is both a bottleneck and easily parallelizable. Therefore, we used multithreading to split up the work for generating the sieve. This optimization further improved performance linearly with respect to the number of threads. Similar techniques can be applied to other parts of the program that contain independent, computationally intensive tasks.

3.3 Performance

Our implementation can factor integers up to 40 digits long within a minute. See Table 1 for the performance of our implementation on other integers. See Table 2 for our computer specifications. See the appendix for sample output.

[1] See https://github.com/Will-Denton/QuadSieve/blob/final/quadratic_sieve.py for our Python implementation.
[2] See https://github.com/Will-Denton/QuadSieve/blob/final/quadratic_sieve.c for our C implementation.

n	Digits	B	S	Time
130607	6	50	1.0×10^3	1ms
29223973	8	200	1.5×10^4	1ms
7067947793	10	500	1.0×10^4	1ms
736055622283	12	600	1.0×10^4	2ms
5479839591439397	16	1000	3.0×10^5	8ms
92905709270744788219	20	1500	3.7×10^5	13ms
60381558672724747724459	23	2500	1.5×10^6	25ms
42121759369990237671075549 23	28	5000	2.5×10^7	423ms
611949000568242841838426129286641237026 9	40	30000	1.5×10^9	31.3s
37448430805296159090191815103305542055009260219 47	49	50000	2.0×10^9	NA

Table 1. Quadratic Sieve Performance for Varying n, B, and S. We did not have enough memory to factor the 49-digit integer.

CPU	Memory
Intel i7 (14 cores)	32 GB

Table 2. Computer Specifications

4 MATHEMATICAL EXTENSION

4.1 Euler's Criterion and Quadratic Residues

When we winnow down our factor base, we assume that n and p are coprime, as n will likely be comprised of large primes, and p is below some smoothness bound. From this assumption, we can use Fermat's Little Theorem, which states that if n and p are coprime, $n^{p-1} \equiv 1 \pmod{p}$. This leads naturally to a duality known as Euler's Criterion to check whether some number n is a quadratic residue mod p.

Specifically, Euler's criterion states that for some odd prime p and n coprime to p,

$$n^{\frac{p-1}{2}} \equiv \begin{cases} 1 \pmod{p} & \text{if } \exists x \in \mathbb{Z} \text{ such that } x^2 \equiv n \pmod{p} \\ -1 \pmod{p} & \text{otherwise.} \end{cases} \tag{6}$$

This criterion follows from Fermat's Little Theorem, which states

$$n^{p-1} \equiv 1 \pmod{p} \implies n^{p-1} - 1 \equiv (n^{\frac{p-1}{2}} - 1)(n^{\frac{p-1}{2}} + 1) \equiv 0 \pmod{p}. \tag{7}$$

From this equivalence, we know that every number in $(\mathbb{Z}/p\mathbb{Z})^*$ must satisfy one of the two cases of the Euler criterion.

If we take some generator g for $(\mathbb{Z}/p\mathbb{Z})^*$, the elements of $(\mathbb{Z}/p\mathbb{Z})^*$ must be represented by g^k for $0 \leq k < p-1$. Of these values for k, half are even and can be written as $k = 2m$.

$$g^k \equiv g^{2m} \equiv (g^m)^2 = a \pmod{p} \tag{8}$$

Therefore at least half of the elements of $(\mathbb{Z}/p\mathbb{Z})^*$ are quadratic residues, meaning there are at least $\frac{p-1}{2}$ quadratic residues. However, as $p - g^m \equiv -g^m$ is also a valid root, and $g^m \not\equiv -g^m \pmod{p}$ for $p \neq 2$, each quadratic residue has two roots. Therefore, there can be at most $\frac{p-1}{2}$ unique quadratic

residues without overlapping roots. This means that exactly half of the elements in $(\mathbb{Z}/p\mathbb{Z})^*$ are quadratic residues, and the other half are quadratic non-residues.

For a quadratic residue $n \pmod p$,

$$n^{\frac{p-1}{2}} - 1 \equiv (x^2)^{\frac{p-1}{2}} - 1 \equiv x^{p-1} - 1 \equiv 1 - 1 \equiv 0 \pmod p. \tag{9}$$

Therefore every quadratic residue must make the term $n^{\frac{p-1}{2}} - 1$ zero. As exactly half of the elements of $(\mathbb{Z}/p\mathbb{Z})^*$ are residues, the other half must satisfy the non-residue case in Euler's Criterion.

4.2 Tonelli-Shanks

Using the process for Tonelli-Shanks [6] as defined in Algorithm 1, we can show that for each loop iteration, the following three loop invariants hold:

$$\begin{aligned} c^{2^{M-1}} &\equiv -1 \\ t^{2^{M-1}} &\equiv 1 \\ R^2 &\equiv tn. \end{aligned} \tag{10}$$

The key principle behind Tonelli-Shanks is the third invariant. As we modify c, t, R, we continuously search for an R value that satisfies $\sqrt{n} \equiv R \pmod p$, and decrease our search area by decreasing the possible values for the order of t.

At the initialization step, the first two invariants should be obvious, as c is constructed from a non-residue, therefore $c^{2^{M-1}} \equiv (z^Q)^{2^{M-1}} \equiv z^{\frac{p-1}{2}} \equiv -1 \pmod p$ by Euler's Criterion. The same computation can be performed on t to show that the second invariant holds at initialization. Finally, as $t = n^Q$ at initialization, $R^2 \equiv (n^{\frac{Q+1}{2}})^2 \equiv n^{Q+1} \equiv tn \pmod p$.

Now, for each iteration, the following hold for the updated values M', c', t', R' of M, c, t, R:

$$\begin{aligned} c'^{2^{M'-1}} &\equiv (b^2)^{2^{i-1}} \equiv c^{2^{M-i}2^{i-1}} \equiv c^{2^{M-1}} \equiv -1 \\ t'^{2^{M'-1}} &\equiv (tb^2)^{2^{i-1}} \equiv (t^{2^{i-1}})(c^{2^{M-i}2^{i-1}}) \equiv (t^{2^{i-1}}c^{2^{M-1}} \equiv (-1)(-1) \equiv 1 \\ &\quad (t^{2^{i-1}} \equiv -1 \text{ as } t^{2^i} \text{ is the least power of 2 of } t \text{ equal to 1.}) \\ R'^2 &\equiv R^2b^2 \equiv (tn)b^2 \equiv t'n. \end{aligned} \tag{11}$$

Therefore, the loop invariants hold for each iteration of the loop. As $t^{2^{M-1}} \equiv 1$, there must be some i that makes $t_i \equiv 1 \pmod p$. Therefore the inner loop will continue to run, and M will decrease with each iteration. As $n^{\frac{p-1}{2}} \equiv 1 \pmod p$ by Euler's Criterion, we are guaranteed to find an R so that $R^2 \equiv n \pmod p$ from this algorithm, thereby giving us a value for $\sqrt{n} \pmod p$.

REFERENCES

[1] 2024. The GNU MP Bignum Library. https://gmplib.org/

[2] 2024. Optimize Options (Using the GNU Compiler Collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[3] Çetin K. Koç and Sarath N. Arachchige. 1991. A fast algorithm for gaussian elimination over *GF*(2) and its implementation on the GAPP. *J. Parallel and Distrib. Comput.* 13, 1 (Sept. 1991), 118–122. https://doi.org/10.1016/0743-7315(91)90115-P

[4] Ben Lynn. [n. d.]. *Number Theory - Quadratic Residues.* https://crypto.stanford.edu/pbc/notes/numbertheory/qr.html

[5] Carl Pomerance. 2008. Smooth numbers and the quadratic sieve. (2008).

[6] Daniel Shanks. 1973. Five number-theoretic algorithms. *Proceedings of the Second Manitoba Conference on Numerical Mathematics (Winnipeg), 1973* (1973). https://cir.nii.ac.jp/crid/1572261550443913728

A APPENDIX

A.1 Sample Output

```
PS C:\Users\willd\OneDrive\Duke\MATH404\QuadSieve> .\quadratic_sieve.exe 611949000568242841838426129286641237026 30000 1100000000
Starting sieve with n: 611949000568242841838426129286641237026, B: 30000, S: 1100000000
Starting get_sieve_log...
Starting sieve_primes_log...
Starting create_matrix_log...
Starting find_linear_dependencies...
Number of dependencies: 60 with matrix shape: 1636 x 1639
Time taken: 34968.000000 ms
611949000568242841838426129286641237026 has factors: (65234815970413269013, 9380711686927842871 3)
```

Fig. 1. Sample output for the 40-digit number 611949000568242841838426129286641237026 9.