# Excursions in Implementing the Quadratic Sieve Algorithm

Teodora Petkova, Samia Zaman

April 20 2021

## 1 Introduction

In modern cryptosystems like RSA encryption, the security of the system relies almost entirely on the difficulty of factoring large integers. The naive method of factoring a large integer N by test dividing by all primes $\leq \sqrt{N}$ is too slow to be effective.[Gor08]
The quadratic sieve method is the second fastest method known for factoring integers after General Number Field Sieve, and often the best attack on RSA for being easier to implement [Wik20].

We implemented the quadratic sieve as part of our project in a Mathematical Cryptography class. We applied the simple mathematical principles on which it is based, but did not greatly optimize for run time. Our implementation is not reflective of current state of the art in terms of run time, but optimization techniques can be easily added. Since our goal was to factor integers less than 20 or so digits, we were able to run our code in reasonable time by implementing in python and using numpy packages optimized for handling arrays and matrices. Additionally, our implementation **assumed that the n we have to factor is the product of just 2 primes**, i.e finding primes $p$ and $q$ such that $N = p \cdot q$

The code for our implementation can be found on the following link:
https://github.com/eliyanovva/quadratic_sieve/blob/master/Quadratic_Sieve_Jupyter.ipynb

## 2 Mathematical Explanation

### 2.1 Introduction:

In this section we introduce and lightly illustrate the mathematical principles using which the quadratic sieve works. Restating our goal: are interested in finding the two prime factors p and q of a large integer n, such $n = p \cdot q$

## 2.2 Motivation behind Quadratic Sieve

We start with an important Lemma:
Lemma 1: Let n be an integer, and suppose there exists integers x and y with $x^2 \equiv y^2$ mod n, but $x \not\equiv \pm y$. Then n is a composite number and gcd$(x - y, n)$ gives a non-trivial factor of n. [TW05].
The lemma motivates the idea that we can find a factor for the large integer n of our interest by finding squares $x^2 \equiv y^2 \pmod{n}$ such that $x \not\equiv \pm y \pmod{n}$. This is the basic strategy of the quadratic sieve in order to factor primes.

## 2.3 Sieving

But how do we find such squares $x, y$ mod n? In general, there are many techniques. One way is to generate:
$f(i) = (x+i)^2 - n = y_i \pmod{n}$, x = $\lfloor\sqrt{n}\rfloor + 1, i = [1, 2, 3...], x, i \in Z - -(1)$.
Modulo n, each of such an equation has a left-hand side equalling a square, namely $(x+i)^2$; however, there is no guarantee that the right hand side - i.e. the residue of $(x+i)^2$ mod n = $y_i$ mod n, should be a square number. This is why we should generate a *sequence* of such equations, so that the product of a subset of the those sequences (of $y_i$'s) can yield a square number. For example: choose $n = 15347$. The floor of square root of n +1 is 124 = x. We can generate the equations:

$$(x+0)^2 - n \equiv 124^2 - n \equiv 29 \equiv 2^0 \cdot 17^0 \cdot 23^0 \cdot 29^1 \pmod{n} - -(1)$$

$$(x+3)^2 - n \equiv 127^2 - n \equiv 782 \equiv 2^1 \cdot 17^1 \cdot 23^1 \cdot 29^0 \pmod{n} - -(2)$$

$$(x+71)^2 - n \equiv 195^2 - n \equiv 22678 \equiv 2^1 \cdot 17^1 \cdot 23^1 \cdot 29^1 \pmod{n} - -(3)$$

So we can see that the three equations multiplied together yield a square on the right-hand side, and of course, the left-hand side was already a square:
$(1) \cdot (2) \cdot (3) = (124 \cdot 127 \cdot 195)^2 \equiv 2^2 \cdot 17^2 \cdot 23^2 \cdot 29^2 \equiv (2 \cdot 17 \cdot 23 \cdot 29)^2$.

In this case, we get that $x \equiv (124 \cdot 127 \cdot 195)$ and $y \equiv (2 \cdot 17 \cdot 23 \cdot 29) \pmod{n}$, with $x - y \equiv 7331 - 1460 \equiv 5871$, and gcd$(5871, 15347) = 103$, which yields, $15347 = 103 \times 149$. [Wik20]

Although the method above worked, we seemed to have arbitrarily chosen to work with the primes $2, 17, 23, 29$. There is in fact good reason that these primes were chosen, namely that these are the first four primes p for which our given n=15437 is a *quadratic residue* modulo p. More about this is discussed in the improvement section. However, in our implementation, we did not implement a method to find such primes, and instead chose the interval $[1, X]$ and all primes less than the value $X$. This was simpler for us, and not computationally prohibitive at the same time, since we worked with numbers n that were not too large.

Returning to the need for choosing a bound on primes, notice that without setting a bound, the sequence of f(i) can get arbitrarily large, and with fewer and fewer chances of finding squares mod n, because once we have a large prime with

odd power, we need to find another equation with that prime to an odd power in its prime factorization (without the latter equation introducing a different large prime to odd power!) to make that prime power even (and hence a square, if the rest of the primes are even powered). Larger primes are further apart, and so finding such numbers $f(i)$ would require too many iterations, i. [Pom05].

## 2.4 Choosing a smoothness bound B

We choose an integer B as a bound for our primes and define a *B-smooth* number to be such that all of its prime factors are $\leq B$. The set of all such numbers is called the *factor base*. An important lemma leads the way for the next steps:
Lemma 2: If $a_1, a_2, ..., a_k$ are a sequence of positive *B-smooth* integers, and if $k > \pi(B)$ where $\pi(B)$ denotes the number of primes in the interval $[1, B]$, then a subset of the product of this sequence is a square.

The lemma guarantees that once we generate more equations (using our $f(i)$) than the size of our factor base, $\mathcal{B}(= \pi(B)$ for now), we will be able to find a product that is a square. This is a result from linear algebra. It relies on the following facts:
First, we can write a *B-smooth* number $A_k$ from the sequence of numbers generated as:

$$A_k = \prod_{i=1}^{\pi(B)} {p_i}^{v_i}$$

where $p_i$ is the $i^{th}$ prime and $v_i$ is a non-negative vector, we call $v(A_k)$ the exponent vector of $A_k$.
We can then write $v(A_k) = (v_1 + v_2 + .... + v_{\pi(B)})$. Now if we have a sequence of more than $\pi(B)$ such $A_k$, and we write their exponent vectors in a matrix, we will have a linear dependency in the matrix once we have more $A_k$'s than the number of primes. Since we are only interested in whether or not our exponent vectors sum to the zero vector mod 2 - which is the same as our $A_k$'s giving a square product, we reduce this matrix mod 2, and note that a linear dependency necessitates that a subset of these vectors will sum to the zero vector.

Thus, we are left with only finding this subset, which we can do with Gaussian elimination as is mentioned in the next and final section on mathematical explanation. We find a solution subset, say with indexing set $S$, and check whether it is valid for factoring n:
The square product found using that subset, call it $y^2 = \prod_{k \in S} A_k$, is valid for factoring n if this $y$ is not congruent to x mod n, where $x^2$ corresponds to the left hand side product of that solution set: $\prod_{k \in S} f(k)$, that we used to generate these *B-smooth* $A_k$'s. Thus, in the language of Lemma 1, we have a valid factor of n when $x \not\equiv y \pmod{n}$.

We would like to choose this bound B optimally. On one hand, if we choose a large B, we can quickly find B-smooth numbers, but we may have to spend

more memory in storing the large exponent matrix, and more time in computing on the matrix to find the solution subsets. On the other hand, if we choose a B too small, we may have to wait until a long time to generate enough B-smooth numbers. [Pom05].

It turns out that an optimum value for the size of the factor base is roughly:

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

[Lan01].

### 2.5 Gaussian reduction of exponent matrix

As discussed above, we need to find a homogeneous solution vector for this matrix. There are many ways to do so, the Gaussian elimination being the most common one. The standard running time of Gaussian elimination is $O(\mathcal{B}^3)$, but the matrix we are dealing with is mod2 and most often sparse, so there is a lot of scope for optimization [Pom05]. However, the bottleneck in running time in quadratic sieve is often the sieving part - i.e. generating enough B-smooth squares. [Gor08] [Lan01] [Pom05]. In our implementation, we did not optimize beyond choosing sympy and numpy resepctively to reduce the matrix to reduced row echelon form and to compute on entries of matrices fast.

## 3 Algorithm Implementation

Based on the mathematical statements above, we implemented the quadratic sieving algorithm as described below.

### 3.1 Input and Output

The input is an integer $n$ which is the product of two large primes.

The output is two factors that multiply out to $n$. We do not run additional primality checks on the factors, and assume our input is always the product of two primes. But doing such a check is possible, and one effective method is the Miler Rubin-Primality test. [[Pom05], chapter 6]

### 3.2 Choice of $B$ and Generation of $\mathcal{B}$

After gathering the input $n$, we set the bound to be:

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

A version of Erathosthenes's sieve is run up to $B$, to generate $\mathcal{B}$ - an array of all primes less than $B$ .

## 3.3 Sieving

In the sieving part of our algorithm we search for numbers whose squares (mod $n$) are B-smooth numbers.

We consider numbers of the form $s = \lfloor\sqrt{mn}\rfloor + 1$, where $m = 1$. In every consecutive step $m$ is incremented with 1 and $s$ is recalculated, until we find $\mathcal{B} + 10$ such numbers, since we then have a probability of $\frac{1023}{1024}$ of factoring $n$ using just this matrix. [Lan01]

So we obtain a matrix $M$ with size $\mathcal{B} \times (\mathcal{B}+10)$, which always has a solution for $\mathbf{v}$ in the matrix equation $\mathbf{v}M = \mathbf{0}$.

Each column of $M$ shows the number of each of the prime factors from $\mathcal{B}$ needed to get the B-smooth number, and each row corresponds to the power of a specific prime from $\mathcal{B}$ in each of the B-smooth numbers. The matrix is then reduced mod 2.

An example of the matrix we build is shown on Figure 1.

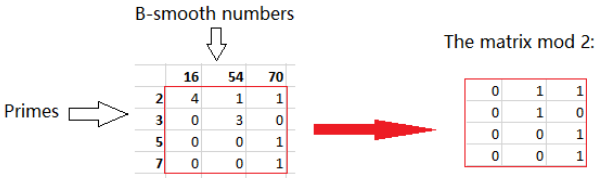


Figure 1: The initial matrix formed by the B-smooth numbers, which is converted to a matrix mod 2. The matrix we work with has more columns than rows, unlike this one.

## 3.4 Solving the Matrix

The next step is finding a solution vector $\mathbf{b}$ mod 2 for the equation described above. The 1s in this vector would correspond to the residues and their corresponding generating numbers which need to be included in the final calculations of $(x, y)$ as described by the Lemma 2 in section 2.

We row reduce the matrix and remove all rows containing only 0s, in order to ease further calculations.

Due to its size, $M$ has many columns which do not have a pivot number. The numbers which correspond to these columns can be chosen arbitrarily to be considered for $x, y$. Thus, their corresponding values in the solution vector can be either 1 or 0. However, the solution vector values which are multiplied by the numbers corresponding to pivot-columns depend on the non-pivot number columns.

Thus, we can have many different solution vectors, based on our choice for the non-pivot values in the solution vector. This is illustrated on Figure 2. We

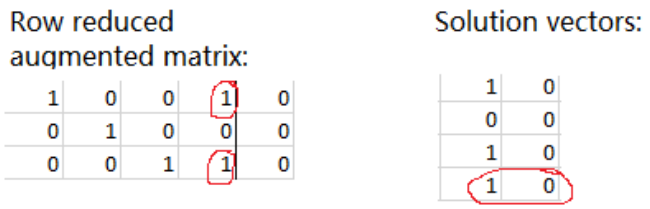


Figure 2: The values marked in red are from a non-pivot column. Hence, for each solution vector value corresponding to these columns, there is a different arrangement of the pivot values.

call the solution vector values which correspond to a non-pivot column *'free variables.'*

If the matrix has more than 1 non-pivot columns, then it will have a different solution vector for each unique combination of free variables. Thus, we go over all different combinations of free variables, until we find one which produces the $x, y$ such that $x \not\equiv \pm y \pmod{n}$. Our algorithm throws an exception if none of the combinations produces such numbers.

We start constructing $\mathbf{b}$ by initializing a vector of ones with length $\#\mathcal{B}$. Then, we iterate over the row reduced matrix, and for each column which contains a pivot number, the corresponding vector entry is turned to 0. In the end, the vector has ones for the free variables and zeros for the others.

Then, a combination vector is built, which contains a binary representation of the current combination of free variables. It is initialized to [0, 0, ..., 0, 1].

> Example: If the free variables are on the following indices in the solution vector: [2, 5, 9, 12, 13], then a binary representation of the combination [2, 9, 12] would be [1, 0, 1, 1, 0]. The binary representation of the combination of all free variables would be [1, 1, 1, 1, 1]. Hence, if we start from the following binary representation: [0, 0, 0, 0, 1], and we increment it analogically to binary addition: [0, 0, 0, 1, 0], [0, 0, 0, 1, 1], [0, 0, 1, 0, 0]... [1, 1, 1, 1, 1] we will get an exhaustive list of all combinations of free variables.

Based on the combination vector, all free variables from the current combination are set to 1. The rest of the free variables are set to 2, since they should be distinguishable from the dependent variables, but their value should be 0 (mod 2) for our further calculation.

To calculate the dependent variables, an iteration over the matrix is performed from the bottom to the top, and each dependent variable in the vector is determined from the sum of the values to its right mod2.

### 3.5 Special Case: the Matrix mod 2 has only 0s

In this case, every vector is a solution vector, thus using the same combination vector logic from above, we build a solution vector where every value is a free variable. This case is rare for big numbers, but is fairly common for a smaller choice of $n$.

### 3.6 Calculating $x$ and $y$

In the next part of our code, we calculate $x, y$ . To calculate $x$, we multiply all numbers with B-smooth squares, whose corresponding solution vector entry is 1. To calculate $y$, we do the same with the B-smooth residues, and take the square root of the result using a Python approximating algorithm[Pyt].

Finally, we find the greatest common divisor of the difference of $x$ and $y$ and $n$, and it is one of the factors of $n$. The other factor is obtained by dividing $n$ by the first factor.

## 4 Performance

Here is a table of tests which were executed showing their results and run-time.

| Number | 3,492,445,409 | 95,775,679 | 10,710,336,959,293 | 502,560,410,469,881 |
|---|---|---|---|---|
| Prime Factor | 59197 | 7757 | 1234577 | 32452843 |
| Prime Factor | 58997 | 12347 | 8675309 | 15485867 |
| Run-time (s) | 4.4318 | 4.137 | 4.5103 | 14.8568 |

## 5 Discussion

Our algorithm works for sufficiently small numbers, however it runs into several problems.

For some values of $n$ like $n = 6817540046645387$, the algorithm returns 1 and 6817540046645387 even though $x \not\equiv \pm y \pmod{n}$ and $x \not\equiv \pm - y \pmod{n}$.. We suppose that this comes from an incorrect generation of $y$, due to the squaring approximating algorithm.

For the numbers $n = 8876645503455770083$ and $n = 34927963314430763$ none of the combinations of free variables produces $x, y$ which satisfy the condition above. It is interesting to note that the B-smooth numbers were generated within a few minutes, but all combinations of them generate $x \equiv \pm y \pmod{n}$. We tried increasing the amount of B-smooth numbers, so we can have even more combinations of free variables, but this did not help us find a solution. We think that the problem can be approached by combining two or more different methods of generating B-smooth numbers, instead of only one, as we do. We tried looking for B-smooth numbers in the range $[\sqrt{mn} + 1 - c, \sqrt{mn} + 1 + c]$,

where $c$ is a constant, but this severly decreased the speed of our algorithm, without helping us find more B-smooth numbers.

For the number $n = 55844846971764297694842228 34$ it took our algorithm more than 3 hours to generate enough B-smooth numbers, and it was not able to row reduce the resultant matrix.

Another issue with our code is that instead of using a factor base with size $\left(e^{\sqrt{\ln(n)\ln(\ln n)}}\right)^{\sqrt{2}/4}$, the bound of our factor base is this number, which makes the factor base itself much smaller than it should be.

# 6 Improvements

At the outset, we generated our sequence of B-smooth numbers using $f(i) = (x+i)^2 - n, i \in [1, 2, 3..]$, where $x = \lfloor \sqrt{n} \rfloor + 1$. This was not sufficiently fast for even numbers of 7-8 digits, so we redefined f to be: $f(i) = (\sqrt{in}+1)^2 - n, i \in [1, 2, 3..]$. This sped up our algorithm considerably, and we were able to find B-smooth squares within seconds for 16 digit numbers, depending on the start value of $i$. However, time-permitting, we were interested to refine our factor base to have only those primes for which n is a quadratic residue mod p. Such primes are numbers for which our $f(i) \equiv 0 \pmod{p}$ [TW05]. This is known in the literature as primes with the *Legendre symbol*, $\frac{n}{p} \equiv n^{(p-1)/2} = 1$. Checking for only such primes makes the sieving faster, but requires us to solve our $f(i)$ mod p using law of quadratic reciprocity, which we didn't get to doing.

Other methods we looked into of speeding up sieving is to use the *Multipolynomial Quadratic Sieve* [Sil] [Lan01][Wik20], but decided against implementing, because of both time constraints and that we wanted to keep our code simple, which we understood best.

Another improvement which can be done on the algorithm is to use a lower-level language for implementation, which would grant us much higher speed and allow us to experiment with more different ways of B-smooth numbers generation. Using a stable Big Integer library in a lower-level language would help us calculate $x, y$ with a better accuracy than now, especially for multiplication and division operations.

# 7 Conclusion

Our approach to implementing the Quadratic Sieve was certainly one of the simpler ones. The main problem with our implementation other than having minimal optimizations, was the failure to obtain factors as per *Lemma*1 guarantees, i.e. even when $x^2 \equiv y \pmod{n}$ but $x \not\equiv \pm y \pmod{n}$ for some large numbers. We believe it could be due to floating point conversion at some point in our code, guarantee of correct computations by means of python's "big int"

feature no longer being applicable. We hope to look into the code and discover this issue.

# References


[Lan01] Eric Landquist. *The Quadratic Sieve Factoring Algorithm*. 2001. URL: http://www.cs.virginia.edu/crab/QFS_Simple.pdf. (accessed: 12.04.2021).

[Pom05] Carl Pomerance. "Smooth numbers and the quadratic sieve". In: *in [Buhler and Stevenhagen 2007]. Citations in this document: §5*. University Press, 2005.

[TW05] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory (2nd Edition)*. USA: Prentice-Hall, Inc., 2005. ISBN: 0131862391.

[Gor08] Mark Gordon. *Implementation of the Quadratic Sieve*. 2008. URL: http://www-personal.umich.edu/~msgsss/factor/qs_rep.pdf. (accessed: 12.04.2021).

[Pyt] RIP Tutorial Python. *Computing Large Integer Roots*. URL: https://riptutorial.com/python/example/8751/computing-large-integer-roots. (accessed: 20.04.2021).

[Sil] Robert D. Silverman. *The Multiple Polynomial Quadratic Sieve*. URL: https://www.enseignement.polytechnique.fr/informatique/INF558/TD/td_5/S0025-5718-1987-0866119-8.pdf. (accessed: 20.04.2021).

[Wik20] Wikipedia. *Quadratic Sieve*. Dec, 2020. URL: https://en.wikipedia.org/wiki/Quadratic_sieve. (accessed: 16.04.2021).