Math 404 Computing Assignment 1 Final Report

Charles Lyu, Joseph Saldutti, Cheng Lyu

April 9, 2019

1 Overview

Our code replicates the algorithm discussed in the paper written by Carl Pomerance. We based the structure on the "proto-algorithm" that he overviews. This algorithm, however, is a very rough outline, and we made many decisions and optimizations in writing the code. Here we will explain it part by part.

2 Choosing B

The first step is choosing a bound B for the set of primes. Different value of B can greatly affects the computation time: This is because adding more primes increases the size of the matrix for the Gaussian elimination, which is our most computationally intensive step. However, if B is too small, then we must sieve through many more possible $x^2 - n$ values to find those that are B-smooth. Therefore, the value of B should be not too small nor too big.

For a number n, the suggested value of B in Pomerance's paper is that $B = \exp\left(\left(\frac{1}{2} + o(1)\right)(\log n \log\log n)^{\frac{1}{2}}\right)$. However, it does not specify the exact value of $o(1)$, so we have to use our own values in practice. Specifically, we choose a lower and upper bound for possible value of B that we want to try: The lower bound is $B = \exp\left(\frac{1}{2}(\log n \log\log n)^{\frac{1}{2}}\right)$, and the upper bound is the lower bound

to the third power. We compute the bounds in the `opt_bound` function, which takes as an argument N, the number we are trying to factor, and returns the lower and upper bounds of B.

We start by setting B to the lower bound, hoping that we can find enough B-smooth numbers to complete the computation. However, if we fail to find a solution, then we increase B by a factor of 10 and try again. This part of the code can be seen in the function `sieve_auto`, which takes as an argument the number N that we are trying to factor. If we somehow reach the upper bound of B without success, our code exits. This has never happened in practice, as if it did the computation may take a few days for the larger values of N in the assignment.

3 Finding primes $\leq B$

After we have chosen B, the next step is to find the list of primes less than or equal to B. The function we wrote to perform this task is named `get_prime` and takes an integer as a parameter, B. This function performs a sieve on the numbers between 2 and B. We start with the number 2 and "cross out" all multiples of 2. If we reach a number in this loop that has not been crossed out, we know it is a prime and add it to the primes list. We then repeat the process with this number, going down the list and "crossing out" all multiples of this number. We repeat until we reach B, and then return the prime list. It can be noted that this list is in order. We take note of the length of this list, i.e. the number of prime numbers that are at most B, $\pi(B)$. This will be important later in checking that we have found enough B-smooth numbers of the form $x^2 - n$.

4 Choosing `x_start` and `x_upper_bound`

After we obtain the list of primes in $[2, B]$, we use the list to actually begin finding numbers x such that $x^2 - n$ is B-smooth, that is, it factors completely as a product of primes that are at most B. We use a for loop to iterate through each x, starting from $\lceil \sqrt{n} \rceil$. We do this because starting with x just above n guarantees that $x^2 - n$ will be small positive numbers. We also choose a stopping point for the loop as, somewhat arbitrarily, $1.01\lceil \sqrt{n} \rceil$. Our goal is that the code will never reach this point, but if it does, we give up on factoring. This means that the value of B we chose is too small since there are not enough B-smooth numbers from all values of x in this range, and we must try again with a larger value of B and more primes.

In practice, our code never reached the upper bound of x, although it still took really long to obtain enough B-smooth numbers.

5 Beginning the sieve

After finding x_{start} and x_{upper}, we iterate through each value of $x^2 - n$ and check if it is B-smooth, until we find enough of such numbers. This is done using a quadratic sieve. This code can be found in `qs.py` in the `sieve` function.

To check whether $x^2 - n$ is B-smooth, we need to find all primes $p \leq B$ such that $x^2 - n \equiv 0 \pmod{p}$, and then divide $x^2 - n$ repeatedly by p. If $x^2 - n$ eventually reduces to 1 after considering all $p \leq B$, then $x^2 - n$ is B-smooth. However, it turns out that looping through all $\pi(B)$ primes for each x is too inefficient. Instead of fixing x and iterating through all p, we implement the quadratic sieve as described in Pomerance's paper, essentially fixing p and consider each x instead.

For each $p \leq B$, we want to find the list of x that satisfies the quadratic congruence $x^2 - n \equiv 0 \pmod{p}$. This congruence has at most two solutions. If

there are no solutions, we can remove p from our factor base, since none of the $x^2 - n$ will be divisible by p. If the solution is $x = 0$, this means $p \mid n$ so we have found a factor of n. Thus, we focus on the general case where there are two non-zero solutions $x_1, x_2 \pmod{p}$. (Details for solving this congruence are described in the next section.)

Theoretically, we can then find all possible x for which $p \mid x^2 - n$. Conceptually, imagine a table with rows as x's and columns as p's. First we look at each column p and examine every p consecutive numbers in the column p: exactly two of them have $x^2 - n \equiv 0 \pmod{p}$ as given by the solution to the congruence (unless $p = 2$ in which case there's only one). We "mark out" this entry (x, p) in table. After we do this for all p, we start examining each x from x_{start}, take all p's that are marked out for this x, and divide $x^2 - n$ repeatedly by each p to see if it reduces to 1. If yes, $x^2 - n$ is B-smooth; we then add it to the list of valid x's, and repeat for the next x until we find $\pi(B)$ B-smooth numbers.

A problem with the approach above is that we don't know the stopping point for x, so we can't look at the entire column p without doing excess work. We thus use an optimization that we call **"delayed marking"**. Instead of examining the entire column p, we only mark out the *first* $x \geq x_{\text{start}}$ such that $x \equiv x_1 \pmod{p}$. We do the same for the first $x \equiv x_2 \pmod{p}$. After repeating this for all p, we start looking at the x's and finding B-smooth numbers. When we find a mark (x, p), we mark out the cell $(x + p, p)$ below. This makes sense because $x + p \equiv x \equiv x_i \pmod{p}$ so $p \mid (x + p)^2 - n$. In this way, as we gradually traverse down the rows, we're actually marking all columns accurately at the same time.

Empirically, we find this approach much faster than trying to divide each $x^2 - n$ by all possible $p < B$, especially when p is large. In the code, finding all the marks (x, p) for a specific x can be made even more efficient by keeping a list of all marked p's for each x, instead of storing the entire table.

5.1 Solving quadratic congruence

We now go back to the problem of solving the congruence $x^2 - n \equiv 0 \pmod{p}$. We do this using the the Tonelli–Shanks algorithm for performance. This is done with the helper method `solve_quad_congruence`, which takes as arguments n, the number we are factoring, and p, a prime in the primes list. (We implemented the algorithm before it was discussed in class, so the details are slightly different.)

We first check that n is a square $\pmod{p}$ by checking that $n^{\frac{p-1}{2}} \equiv 1 \pmod{p}$. This is true because of Euler's criterion, discussed in class. If n is not a square, then there are no solutions x to this congruence and we return that. We now have that $n^{\frac{p-1}{2}} \equiv 1 \pmod{p}$, which is important in later steps.

The algorithm needs a few pieces to start. First, we must find values of Q and S such that $p - 1 = Q \cdot 2^S$ where Q is odd. We do this by starting with $Q = p - 1$, and repeatedly dividing Q by 2 until Q is odd. We increment S each time. The next piece we need is a number z that is a quadratic non-residue, which is equivalent by Euler's criterion to $n^{\frac{p-1}{2}} \equiv -1 \pmod{p}$. About half of the numbers in the set $\mathbb{Z}/p\mathbb{Z}$ have this property, so we can find one easily by starting at $z = 2$ and checking Euler's criterion for each value of z.

Now, using these numbers (z, Q, S), we set up the variables needed for the algorithm. They are as follows: $M = S$, $c \equiv z^Q$, $t \equiv n^Q$, and $R \equiv n^{\frac{Q+1}{2}} \pmod{p}$. Now, we have three important facts about these numbers:

$$\begin{aligned} c^{2^{M-1}} &= z^{Q \cdot 2^{S-1}} = z^{\frac{p-1}{2}} \equiv -1 \pmod{p}, \\ t^{2^{M-1}} &= n^{Q \cdot 2^{S-1}} = n^{\frac{p-1}{2}} \equiv 1 \pmod{p}, \\ R^2 &\equiv n^{Q+1} \equiv tn \pmod{p}. \end{aligned}$$

The goal of the algorithm is to iterate until we find $t = 1$. Then we know that $R^2 \equiv tn \equiv n \pmod{p}$ and we have solved the congruence, so we return R and $-R \pmod{p}$. If we find that $t = 0$, then we know that $n \equiv 0 \pmod{p}$ because $R^2 \equiv 0$. We return $(0, 0)$.

The first step in the iteration is to find the smallest value of i such that $t^{2^i} \equiv 1$. Then we can update the variables in the following way: let an intermediate value $b = c^{2^{M-i-1}}$. Then we set $M = i$, $c = b^2$, $t = tb^2$, and $R = Rb$. The same three facts will hold with these new values. Let the new values be c', M', etc. Now,

$$\begin{aligned} {c'}^{2^{M'-1}} &= \left(b^2\right)^{2^{i-1}} = c^{2^{M-i}2^{i-1}} = c^{2^M-1} \equiv -1 \pmod{p}, \\ {t'}^{2^{M'-1}} &= \left(tb^2\right)^{2^{i-1}} = t^{2^{i-1}} b^{2^i} \equiv (-1)c^{2^{M-i-1}2^i} \equiv (-1)^2 \equiv 1 \pmod{p}, \\ {R'}^2 &= R^2 b^2 \equiv tnb^2 \equiv t'n \pmod{p}. \end{aligned}$$

Note that $t^{2^{i-1}} \equiv -1$ because i is the least value such that $t^{2^i} \equiv 1$. $t^{2^{i-1}}$ is the square root of this, so it must be -1. Thus, we can use these new values, checking if $t = 0$ for a solution of $(0, 0)$ or if $t = 1$ for a $(R, -R) \pmod{p}$ solution.

6 Creating matrix of exponent vectors

If we find an x that is suitable (i.e. produces a value of $x^2 - n$ that is B-smooth), then we add this number to a list of valid x's and add its exponent vector to the list of corresponding exponent vectors. The exponent vector can be found as a by-product of repeatedly dividing $x^2 - n$ by all marked out p's. We keep checking for valid x's until we have as many valid exponent vectors as the size of our factor base. (This is often much smaller than $\pi(B)$ as we've discarded some primes when solving the quadratic congruence.) Once we have this many, we know there is probably some linear combination of these vectors that sums to $0 \pmod 2$. We can now find the solutions to this using Gaussian elimination. This part is the `try_solve` nested function in `qs.py`.

We have a few functions that perform these tasks. First, since our code stores the exponent vector as dictionaries mapping primes to their exponents (to save time), we need to convert them to actual vectors in `try_solve`. We then call

`find_subset` which takes as an input the list of x's (that give B-smooth numbers) and their corresponding exponent vectors. In this function, we construct a matrix using these exponent vectors $\pmod 2$. We do this by taking each number in each exponent vector and returning it modulo 2. This creates a matrix which is a list of exponent vectors, each of which are lists of the corresponding exponents $\pmod 2$. However, the matrix is not ready yet. The exponent vectors are currently the rows of this matrix, so we transpose the matrix so they are the columns. We can now do Gaussian elimination on this transposed matrix, M to find solutions.

7 Gaussian elimination

Gaussian elimination is done through the function `gauss_elim` which takes a matrix M as an argument and returns the solutions to $M\mathbf{x} = 0 \pmod 2$. (This function only works on $\mathbb{Z}/2\mathbb{Z}$.)

We examine the matrix row by row, and each time we try to create a new pivot column in the matrix M. For each row, we find any entry that is currently equal to 1. If there exists one, that column is not a pivot column yet: we then make that entry a pivot, and add a copy of this row to any other row whose entry in this column is 1. This column is then made a pivot column, with only the entry at this row being 1. (Note that our process doesn't make the matrix in row-echelon form, but swapping rows turn out to be too time-consuming so we decided to use an equivalent method.)

Afterwards, the non-pivot columns become the free variables useful for producing non-trivial solutions. For each non-pivot column j', $x_{j'}$ can be given an arbitrary value in $0, 1$; then for pivot columns j with pivot $M_{ij} = 1$, x_j should take on a value such that $M_i \cdot \mathbf{x} = 0$. Note that this equation only depends on x_j and free variables $x_{j'}$, x_j could be uniquely determined. Thus, every setting of $\{x_{j'}\}$ produces a distinct solution, and the solution is non-trivial as long as

one of the $x_{j'}$'s is nonzero.

Theoretically, we could produce all solutions to $M\mathbf{x} = 0 \pmod 2$ by considering each setting of $\{x_{j'}\}$; however, that takes exponential time to compute. Therefore, the `gauss_elim` function in our code returns 5 random non-trivial solutions by giving random values to each $x_{j'}$.

Each solution to $M\mathbf{x} = 0 \pmod 2$ corresponds to a subset of exponent vectors that sum up to $0 \pmod 2$, since the exponent vectors are column vectors in M: The subset S can be constructed as $S = \{j : x_j = 1\}$. This conversion is seen in the `find_subset` function.

8 Finding factors of n

After we obtain some possible subsets of x's whose exponent vectors sums to $0 \pmod 2$, as `find_subset` returns, we have the variables `chosen_nums_sets` and `chosen_vecs_sets`. As mentioned above, `chosen_nums_sets` contains 5 solutions; each solution is a set S of numbers x, whereas the corresponding element in `chosen_vecs_sets` is the set of corresponding exponent vectors V that represent the factorization of each $x^2 - n$.

We can now find a and b that satisfy $a^2 \equiv b^2 \pmod n$, setting them to $a = \prod_{x \in S} x$ and $b = \left(\prod_{x \in S}(x^2 - n)\right)^{\frac{1}{2}}$. This is done in the function `find_factor`, which takes as input a list of chosen numbers S and a list of chosen exponent vectors V. It calculates a by multiplying all values of the chosen numbers. b is calculated by summing up all exponent vectors (this sum represents b^2), then raising each prime factor to its exponent value divided by 2. This gives us the square root of b^2, or b.

We check at the end if $a \equiv \pm b \pmod n$: If this happens, unfortunately this particular solution S won't help us find a factor of n, but since our Gaussian elimination returns 5 solutions, we have a few tries, and hopefully in one of them this is not the case. If $a \not\equiv \pm b \pmod n$, $\gcd(a - b, n)$ is a non-trivial factor of

n. We then calculate the $\gcd(a-b, n)$ using our `gcd` function. `gcd` works by repeatedly replacing (a, b) with $(b, a\%b)$, returning a when $b = 0$.

9 Experimental Results

Our code was able to factorize all 4 large numbers given in the assignment. The results are as follows:

$n = 16921456439215439701$:

```
Time taken (in seconds): 6.521
A factor of 16921456439215439701 is 2860486313
```

$n = 468395662999369192342467268 09$:

```
Time taken (in seconds): 12.240
A factor of 468395662999369192342467268 09 is 468395662504823
```

$n = 617283580864197520363830491969135846966 3$:

```
Time taken (in seconds): 2469.330
A factor of 617283580864197520363830491969135846966 3 is
1111111111111111011
```

$n = 37448430805296159090191815103305542055009260219 47$:

```
Time taken (in seconds): 35685.467
A factor of 37448430805296159090191815103305542055009260219 47 is
3333222255555557777777777
```

Note the last number took almost 10 hours to factorize.

Empirically, it appears that bulk of the time is spent on finding enough B-smooth numbers. As an example, for the largest n our code took 8 hours to find 4610 B-smooth numbers and 2 hours for the Gaussian elimination. However, we think the current size of B is reasonable: this is because runtime for Gaussian elimination is effectively cubic in the size of the matrix, so it could blow up easily if B is larger and there are more primes in our factor base.