Homework 7

Due **Saturday**, October 30

Implementation Part

Let's implement the factorization using difference of squares! We'll first do the quadratic sieve:

1. Build a function `quadraticSieve(a,b,B,N)` which returns the B-smooth numbers of the form $t^2 - N$ for integers t satisfying $a \leq t < b$, as well as their prime factorizations. Do this by implementing the quadratic sieve discussed in the Thursday 10/21 lecture, or in 3.7.2 in [HPS]. It will likely be useful to follow along the example for tha chapter, which should correspond to `quadraticSieve(15,30,7,221)`. The steps of the algorithm should loosely be as follows.

   (I) Make a list $a^2 - N, (a+1)^2 - N, (a+2)^2 - N, \cdots, (b-1)^2 - N$. You will sieve this list.

   (II) I found it easier to take care of the even prime powers first. Divide all the even numbers in your list by 2 as many times as possible. *Keep track of how many powers of 2 you factored out*

   (III) For each odd prime $p < B$, solve $x^2 \equiv N \mod p$. If there is no solution, move on. If there is one solutiion...well, then you've factored N! (Why?) Otherwise:

   i. There are 2 solutions α_p and $\beta_p = p - \alpha_p$. Find the smallest $t > a$ such that $t \equiv \alpha_p \mod p$. Starting at the element of your list corresponding to $t^2 - N$, divide every p'th element of your list by p (keeping track of how many factors of p you're pulling out!). Do the same for β_p

   ii. For the corresponding prime powers p^e such that $p^e < 2(b-a)$, solve $x^2 \equiv N \mod p^e$, α_{p^e} and β_{p^e}. Find the smallest $t > a$ such that $t \equiv \alpha_{p^e} \mod p$. Starting at the element of your list corresponding to $t^2 - N$, divide every p^e'th element of your list by p (keeping track of the factors again!). Do the same for β_{p^e}.

   (IV) Look through your list. Everything that's been reduced to 1 is B-smooth! Return the corresponding t values as well as the prime factorizations of $t^2 - N$

   A few remarks.

   - You are free to use Sage's function `prime_range(lower,Upper)`, which returns a list of primes between the lower and upper bounds.
   - Finding square roots mod p is easy, but we haven't discussed it yet in generality. Therefore you are free to use Sage's function `Mod(a,p).sqrt()` which returns a square root of a mod p. This will technically return an element of $\mathbb{F}_p$ so you may want to cast it as an `int` before continuing. It turns out that if $p \equiv 3 \mod 4$ there is a really slick 1 step algorithm to find a square root (see question 5).
   - The above is how I implemented it, but you can be creative if you can think of improvements. As for the way you store your data, again your use of data structures is your choice, but I kept track of a list of lists, which had one entry for each integer t between a and b. Then the entry in the list corresponding to t was

     $$[d, \#\text{Powers of } 2, \#\text{Powers of } 3, \cdots, \#\text{Powers of } p],$$

where p is the largers prime $< B$, and d is what remains after dividing $t^2 - N$ by primes during the sieving. Then I'm just adding to the appropriate powers at each step, and at the end I'm looking for entries where $d = 1$, and the prime factorization is right there!

2. Run `quadraticSieve(15,30,7,221)` and `quadraticSieve(15,30,11,221)`. Does this match [HPS 3.7.2]?

3. Making the sieve was all the hard work! What remains is pushing around data and doing linear algebra, and we're gonna let Sage do the linear algebra for us. Write a function called `sieveFactor(a,b,B,N)` which will try to factor $N = pq$ using the quadratic sieve from part 1. It should loosely run as follows.

   (I) First run `quadraticSieve(a,b,B,N)`. From this you can extract the the a_i between a and b such that $c_i = a_i^2 - N$ is B-smooth, as well as the prime factorizations of the c_i. In particular, you can extrac the e_{ij} such that $c_i = p_1^{e_{i1}} p_2^{e_{i2}} \cdots p_t^{e_{it}}$ where the p_i are precisely the primes $< B$. Make a matrix (or nested array) `E[i][j]`=e_{ij}

   (II) Turn E into a matrix M over $\mathbb{F}_2$ using the Sage command `M = matrix(GF(2),E)`. Then compute a basis for the nullspace of this matrix using `basis = M.kernel().basis()`. This does the row reduction you know and love from linear algebra! *Note: This turns out to be the slowest part of this algorithm! There are better ways to solve sparse matrices of these form, but this is not really a class in computational linear algebra.*

   (III) Each element of the basis gives you a subset of the c_i whose product is a perfect square! Let B be the square root of this product, and let A be the product of the corresponding a_i. Now $\gcd(N, B - A)$ might just be your factor! Try this for every element of your basis.

4. Let's test this out!

   (a) Run `sieveFactor(15,30,7,221)`. Did it work?

   (b) Recall that $L(X) = e^{\sqrt{\ln(X)\ln\ln(X)}}$. If $a = \lfloor \sqrt{N} \rfloor + 1$, and $B = L(N)^{1/\sqrt{2}}$, then we saw in class that `sieveFactor(a,a+L(N),B,N)` should work! Try that out to factor the following numbers:

      i. 8249

      ii. 7799773

      iii. 9488773076569

      iv. 1182692471909987

   Unfortunately I couldn't really get it to factor anything bigger than this, but I think there's a lot of optimization one can do both in how you store the data and in how you do the linear algebra.

Written Part

5. In problem 1 we computed square roots using Sage's built in functionality. But if $p \equiv 3 \mod 4$, there is actually an easy algorithm! So fix $p \equiv 3 \mod 4$ and let $a \in \mathbb{F}_p^*$ have a square root mod p. Give a $\mathcal{O}(\log p)$ algorithm to compute a square root of a modulo p, and prove its correctness. (*Hint: You can do this in a single exponentiation!*)

6. Let $L(X) = e^{\sqrt{\ln x \ln \ln x}}$. Prove that $L(X)$ is subexponential (in the number of bits of X) by proving:
   (a) $L(X) = \mathcal{O}(X^\beta)$ for every $\beta > 0$.
   (b) $L(X) = \Omega((\ln X)^\alpha)$ for every $\alpha > 0$.
7. Optimizing the various parts of our sieve factorization algorithm one can show that we can factor N in about $\mathcal{O}(L(N))$, which is subexponential! Let's see how good this is. For simplicity, suppose it takes about $L(N)$ computations to factor N, and we have a computer than can run a billion computations in a second. How long would it take to factor N of the following orders. (Put your answer in seconds, days, years...whatever is appropriate. Also if you do your computations on cocalc turn that part in too so the grader can see).
   (a) $N \approx 2^{100}$.
   (b) $N \approx 2^{250}$.
   (c) $N \approx 2^{500}$.
   (d) $N \approx 2^{1000}$.

Recall the function $\Psi(X, B) = \#\{n \leq X : n \text{ is } B\text{-smooth}\}$. In class we stated the following claim about the growth of Ψ in certain cases

Theorem 1 ([HPS] Theorem 3.43)**.** *Suppose there exists some* $0 < \varepsilon < 1/2$ *such that:*

$$(\ln X)^\varepsilon < \ln B < (\ln X)^{1-\varepsilon}.$$

Let u *be the ratio* $\ln X / \ln B$*. Then the number of* B*-smooth numbers less than* X *satisfies:*

$$\Psi(X, B) \approx X u^{-u}.$$

(Note, here $\approx$ can be taken to mean that their difference is a function whose limit as X goes to infinity is 0, although in the book they have something slightly more precise). This had the following Corollary, which is more useful for our analyis.

Corollary 1 ([HPS] Corollary 3.45)**.** *Let* $0 < c < 1$*. Then:*

$$\Psi(X, L(X)^c)) \approx X \cdot L(X)^{(-1/2c)}.$$

8. Prove Corollary 1 using Theorem 1. In particular, prove the following two steps.
   (a) Show that there exists some $0 < \varepsilon < 1/2$ with

   $$(\ln X)^\varepsilon < \ln(L(X)^c) < (\ln X)^{1-\varepsilon}.$$

   (b) Let $u = \ln X / \ln(L(X)^c)$. Show that:

   $$u^{-u} \approx L(X)^{-1/2c}.$$

   Then leverage that $\approx$ is transitive to deduce the corollary.
   (*Hint*: Write $u^{-u} = L(X)^{\frac{-1}{2c}(1+f(X))}$ for some function $f(X)$ such that $\lim_{X\to\infty} f(X) = 0$. In fact, this is the definition of $\approx$ given in the book!).