Math 571-01, Cryptography Project 02
Quadratic Sieve
University of Massachusetts Amherst

Matthew Gramigna
Barry Greengus
Wei Xie

April 5, 2017

1 Introduction

The quadratic sieve is an efficient method to find many numbers greater $\sqrt{N}$ for some given N whose squares mod N are "B-smooth" for a given positive integer B. i.e. Let $N, B \in \mathbb{Z}$. Find $a^2 \pmod{N}$ s.t. $\forall p_i$ in $a^2 = p_1^{e_1} * p_2^{e_2} * ... * p_k^{e_k}$, $p_i \leq B$

1.1 Difference by Squares Factoring

Consider the problem of factoring, specifically the method of factoring using difference of squares. If a number N is known to be the difference of two squares, say $N = X^2 - Y^2$, then $N = (X+Y)(X-Y)$. So all we have to do to factor N is to find a number b such that $N + b^2$ is a perfect square. Then $N + b^2 = a^2$, so

$$N = a^2 - b^2 = (a+b)(a-b)$$

and we have just factored N.

A random value of b is unlikely to produce a perfect square, but it is fairly likely for a multiple k of N to equal the difference of two squares

$$kN = a^2 - b^2 = (a+b)(a-b)$$

such that $(a+b)$ or $(a-b)$, besides for being a factor of kN, is also a non-trivial factor of N. This means we only need to find a difference of two squares that equals a mutiple of N, which is the equivalent of finding a and b such that $a^2 \equiv b^2 \pmod{N}$. This fact enables a three step factoring algorithm comprised of:

Step 1: Find integers $a_1, ..., a_k$ such that $c_i \equiv a_i^2 \pmod{N}$ factors as a product of small primes.

Step 2: Take some product of the c_i's that forms a perfect square b^2.

Step 3: Take $\gcd(N, a-b)$ where a corresponds to the product of the corresponding squares for each c_i value. This gcd will give a non-trivial divisor of N for one or more of the product of c_i's. The other divisor can be found by dividing N by the gcd, and then we have the factorization of N.

Steps 2 and 3 are not very difficult in practice. The toughest part is step 1, finding such a_i which are said to be "B-Smooth." The Quadratic Sieve is used to find such numbers.

2 Overview of Quadratic Sieve

The quadratic sieve finds "B-Smooth" numbers for a given N, b, and B using the following method:

Setup Step: Given some number N and set of primes P, where all elements in $P \leq B$, let $a = \lfloor \sqrt{N} \rfloor + 1$. Set a quadratic polynomial to be the function used to build the list. We will use $F(T) = T^2 - N$.

Step 1: Build a list of $F(a)$ to $F(b)$ for some specified bound b. In this step, we want all of the number in the list to be between 1 and N, and for a large N, our choice of F accomplishes this.

Step 2: For each prime $p \in P$, determine if the congruence $t^2 \equiv N \pmod{P}$ has any solutions. If it does not, skip p and continue on to the next prime. If it has solutions say α_p and β_p, then we know that the elements

$$F(\alpha_p), ..., F(\alpha_p + kp)$$

and

$$F(\beta_p), ..., F(\beta_p + kp)$$

are all divisible by p. Sieve out a factor of p from all of these numbers.

Step 3: Repeat step 2 for prime powers $p^k \leq B$.

Step 4: Any elements that have been sieved to 1 are the "B-Smooth" numbers we are looking for.

This completes step 1 of the factoring by difference of squares method mentioned in section 1. Once we have these B-Smooth numbers, form a matrix representing the parity of the c_i exponents, and try the gcd until a non-trivial divisor is found.

3 Implementation

We used gp-pari for our implementation, mainly due to its power and useful built-in mathematical functions. The source code is in section 5 of this document.

3.1 Initial Approach

Initially, we wanted to get our code to work for a small example such as the one in the book, and then expand it to see how it worked for the larger examples. We also were not considering efficiency at this time, and wanted the algorithm to work before we tried any optimization. Naively, this approach involved iterating through every number in the list and checking divisibility by all primes in our list of primes then dividing each number as much as possible.

This approach works, but not efficiently. In fact, we need not consider primes p where the congruence $a^2 \equiv N \pmod{p}$ has no solutions. Also, once we know that some number a is divisible by p, we can also divide multiples of p from that number a in the list and skip over ones we know will not be divisible by p. This leads to our more efficient and final implementation.

3.2 Final Implementation

For our final implementation, we used the aforementioned techniques to improve the efficiency of the sieve by only considering numbers and primes that we know have a chance of being divisible by said prime.

As seen in the code, we use a guard for testing if a number is a quadratic residue modulo some prime in the list p, and if not then we don't consider that prime in the sieve. Moreover, we use the square roots modulo p to determine where to start the divisions for the sieve in the `forstep` loop. Once we have the sieved list, we find the matrix kernel to form the squares as mentioned in section 2, and return the factorization of N.

3.3 Interesting Details/Pitfalls

We struggled with determining appropriate values for b, the size of the list of numbers, and B, the bound on the size of primes to use in the sieve. We found that these values can be quite different depending on the number being factored. In the end, we adjust these values on a case-by-case basis given the number we are trying to factor.

We also initially wrote our own function to find the square roots modulo n. However, gp has a much quicker function `polrootsmod` that finds the roots modulo p where p is a prime. We switched to using this method when we have a prime, and the naive method when we have a prime power, as gp is not built to handle that. This noticeably improved the efficiency of the code.

3.4 Testing

In order to ensure that the algorithm works as desired, we added a simple check to make sure the factorization is correct:

```
{
  validate(factors,N)=
    if(factors[1]*factors[2]==N, return(True));
    return(False);
}
```

This verifies that the factorization found by the algorithm is indeed the correct factorization. The string "True" or "False" is present in the return value of the `QSfactor` function.

4 Efficiency

After allocating some memory to gp, we were able to get through 11 examples using specific b and B values. The timing information is as follows:

```
[149, 2081, True] time = 1 ms.
[32803, 2161, True] time = 3 ms.
```

```
[32843, 524453, True] time = 8 ms.
[8388637, 524497, True] time = 7 ms.
[8388733, 134218069, True] time = 28 ms.
[2147483867, 134218391, True] time = 457 ms.
[2147483777, 34359738797, True] time = 586 ms.
[34359738779, 549755815003, True] time = 2,508 ms.
[8796093022933, 549755814841, True] time = 3,222 ms.
[8796093023383, 140737488355781, True] time = 1min, 6,096 ms.
[2251799813687429, 140737488355393, True] time = 1min, 56,229 ms.
```

5 Source Code

```
{
  getRoots(N,p)=
    local(sols=[], roots);
    if(isprime(p),
        roots=polrootsmod(x^2-N,p);
        for(i=1, #roots,
          sols = concat(sols, lift(roots[i])));
      return(sols);
    );
    for(i=1,p,
      if(Mod(i^2,p) == Mod(N,p),sols=concat(sols,i))
    );
    return(sols);
}
{
  validate(factors,N)=
    if(factors[1]*factors[2]==N, return(True));
    return(False);
}
{
  QSfactor(N,b,B)=
    local(
      roots, \\ solutions to t^2 = N (mod p)
      sieveStart, \\ starting index for each division
      reducedNums=[], \\ list of numbers that sieve to 1
      a=ceil(sqrt(N)), \\ starting T value of the list
      P=primes([2,B]), \\ list of primes <= B
      pow=1, \\ starting exponent for prime powers
      numberList=vector(b+1,i,((i+a-1)^2-N)), \\ T^2-N list
      fac, \\ prod of small primes in congruence
      M, \\ matrix to find kernel of,
      kernel, \\ kernel of M
      product1=1, \\ first product for the gcd subtraction
      product2=1, \\ second product for gcd subtraction
      product2Fac, \\ factorization of product2
      product2Base=1, \\ what is fed into gcd computation
      g \\ gcd(N, product1-product2Base)
    );
    for(i=1,#P,
      \\ Only need to consider square roots mod p
      if(issquare(Mod(N,P[i])),
        pow = 1;

        \\ continue dividing by powers of p
        while(issquare(Mod(N,P[i]^pow)),
          if(P[i]^pow > B, break);
          roots=getRoots(N,P[i]^pow);
          for(j=1,#roots,

            \\ Determine where to start dividing from
            for(n=1,P[i]^pow,
              if(Mod(n+a-1,P[i]^pow) == Mod(roots[j],P[i]^pow),
                sieveStart=n;
                break
              )
            );

            \\ Divide all multiples of p^pow in the list
            forstep(k=sieveStart,#numberList,P[i]^pow,
              numberList[k]=numberList[k]/P[i];

              \\ Make note of fully sieved numbers
               if(numberList[k]==1,
                 reducedNums = concat(reducedNums, k+a-1)
               )
            );
          );
          pow += 1;
        )
      )
    );

    \\ init M with B rows and col for each sieved to 1 num
    M=matrix(#P,#reducedNums);

    \\ fill M with exponents for each prime
    for(i=1, #reducedNums,
      fac=factor(lift(Mod(reducedNums[i],N)^2));
      for(j=1, #fac[,1],
        M[select((x) -> x == fac[j,1], P, 1)[1],i]=fac[j,2];
      )
    );

    \\ find the kernel of M mod 2
    kernel=matker(Mod(M,2));

    \\ try each product combination in the kernel
    for(i=1, #kernel[1,],
      product1=1;
      product2=1;
      product2Base=1;
      for(j=1, #kernel[,i],
        if(kernel[j,i] == Mod(1,2),
          product1 *= reducedNums[j];
          product2 *= lift(Mod(reducedNums[j],N)^2);
        )
      );
      product2Fac=factor(product2);
      for(j=1, #product2Fac[,1],
        product2Base *= product2Fac[j,1]^(product2Fac[j,2]/2)
      );
      g = gcd(N, product1-product2Base);

      \\ stop if we find a non-trivial divisor
      if(g != 1 && g != N, break);
    );

    \\ returns a list of the prime divisors
    \\ plus a boolean for if the factors are valid
    return([g, N/g, validate([g,N/g], N)]);
}
```

6 Group Organization/Administrative

6.1 Git

We used a git repository to easily contribute to the project across different machines, and track our progress along the way. Git is useful for seeing each step taken towards completing the algorithm.

6.2 Meetings

In addition to using a GitHub repository, we also met in person to work on the core functionality of the algorithm. To us, it was important that we accomplished the toughest parts of the algorithm while meeting in person to avoid confusion down the road. After we met, we had most of it done, which made the remaining pieces less of a hassle to implement