An Overview of Factoring Algorithms

Calvin Roth

Submitted under the supervision of Daniel Johnstone to the University Honors Program at the University of Minnesota-Twin Cities in partial fulfillment of the requirements for the degree of Bachelor of Science, magna cum laude in Mathematics.

05/10/2020

1 Introduction

Prime numbers have played a prominent role in the development of mathematics and their use is a critical piece in the modern day computer security protocols. In this paper, we will explain the important developments leading to fast factoring and primality-proving algorithms. A factoring algorithm takes to find a factor of a given number where a primality-proving algorithm only has to determine if the given number is prime or not. The goalpost to be considered 'fast' is that the cost of these algorithms is that they run in subexponential time relative to the number of digits of the number. We will see that this can be achieved.

The math used to discuss the properties of prime numbers is deceptively complex. The concept of prime number, an integer that is divisible by no other positive integer besides itself and one, is a concept that elementary students learn. But one can dive much deeper. The fast algorithms we will see are only possible because of the developments of abstract algebra and analysis. For an example of the mathematical depths involved in the issue of factoring consider the Miller Primality Test. This is an algorithm that determines if a number is prime in deterministic polynomial time if the extended Riemann hypothesis is true otherwise it's correctness is probablistic. These sorts of unexpected connections between modern mathematics and our innocuous looking problem of factoring have led to interesting powerful developments in the issue of factoring.

Our capacity to handle larger and larger numbers has grown considerably over the years. The cause of this increased power is twofold. First, computers have gotten exponentially more powerful over the last 60 years. Second, the algorithms that are used have become much more sophisticated. For example of this growth, consider the number: 114 381 625 757 888 867 669 235 779 976 146 612 010 218 296 721 242 362 562 561 842 935 706 935 245 733 897 830 597 123 563 958 705 058 989 075 147 599 290 026 879 543 541.

This number was suggested by Martin Gardener in 1977 with the claim that finding a factorization of this number would be expected to possibly take quadrillions of years[1]. But instead the pace of developments made it factorisable by 1994 [2] by methods that we will see in this paper. Our abilities have continued to grow. By 2020, RSA250, a 250 digit number of two approximately equaled sized primes, can be factored. The scale of numbers that can be proven prime is even larger. Numbers such as $8656^{2929} + 2929^{8656}$ a number with over 30 thousand digits can be proven prime proven to be prime.

Primes play an important role in modern cryptography. An excellent example is the RSA algorithm. The goal of RSA is to do public key encryption. In public key cryptography one party Alice, publishes a public key that any one may use to send as an encrypted message to that person by performing some mathematical operation what uses the message and the public key as inputs. This operation should be such that it is 'hard' to get the original message from the result unless you have access to some other piece of information

the public key publisher keeps secret called the private key. RSA relies on the assumption that factoring is a difficult problem.

In this paper, we will discuss several fundamental factoring and primality proving algorithms. For primality proving, we will look first at a collection of tests that rely on Fermat's Little theorem and the AKS primality test. The AKS primality test was the first deterministic polynomial time primality test. For factoring, we focus on Pollard's p-1 factoring method, the Quadratic Sieve, the Number Field Sieve, and the Elliptic Curve Method. Pollard's p-1 is of interest because of how the strategy it employs is similar to the central idea of the Elliptic Curve Method. The elliptic curve method and the number field sieve stand today as the fastest subexponential algorithms in our arsenal to factor a large number.

2 Preliminaries

In this section, we will discuss some preliminaries that we will make frequent use of in this paper.

2.1 Common Notation

Throughout this book there is some common notation that we will use frequently Z_n and $\mathbb{Z}/n\mathbb{Z}$ will mean the integers mod n. $Z[x]$ will denote the ring of polynomials of the variable x with integer coefficients. I expect the readers to be familiar with big O notation but we will also use $\tilde{O})$ to represent soft O notation. In many of our algorithms we will come up with run times that are nearly some clean bound aside from being multiplied by log factors. $f(n) = \tilde{O}(g(n))$ if $f(n) = O(g(n) * log^k n)$. This will lead to nicer looking formulas for our run time at respresent having essentailly $O(g(n))$ runtimes since $log(n)$ grows slowly.

2.2 B Smooth Number

A class of number we will use frequently is B-smooth numbers. We say a number is B-smooth for some parameter B if all the prime factors of that number are less than our equal to B. For example, 12 is 4-smooth while 10 is not. The reason this will be useful is we can adept the basic Sieve of Eratosthenes to collect B smooth numbers. We just do the sieve like normal but stop after the number we are sieving by exceeds B. This is a very efficient way to collect a batch of B-smooth numbers for each we know the full factorization. We will see in the quadratic sieve and the number field sieve this is very effective.

2.3 Common Operations

There is a common core of operations we will find ourselves doing over and over throughout this paper. Mathematically, these are standard operations like multiplication, exponentiation, and finding the greatest common divisor but computationally how we efficiently compute these operations is critical since we will be dealing with very large numbers. There is a lot that can be do to speed up these operations but here we will give the basic idea and the runtime of each.

2.3.1 Multiplication

The standard multiplication of two B-bit numbers involves summing D D-bit terms, i.e $12 \times 34 = 4 * (12) + 3 * (12) * 10$ so the standard multiplication operation takes roughly $O(B^2)$. But using the fast Fourier Transform we can express multiplication as the convolution of two signals instead. Using this technique we may bring our runtime down to $O(B(lnB)(lnlnB))$ to multiply to B bit long numbers. For a detailed discussion, Bernstein takes a devoted look at the issue [3].

2.3.2 Exponentiation

The basic idea to calculate x^y quickly is a class of methods called binary ladders. These methods take advantage of the structure of y. The idea is that to find x^{2k} we can find x^k and calculate $x^k x^k$ and if we are computing x^{2k+1} do xx^{2k}. If we wish to do $x^y \mod M$ we use the same logic but with a mod operator applied each step. The following algorithm is the iterative rather than recursive form of this idea and for generality will compute the mod form.

```
# Assume that the ith bit y can be access with y[i]
def exp(x, y, M):
    result = x
    # Start at the second to highest bit
    for j in range(D-2, -1, -1):
        z = (z * z) % M
        if(y[j] == 1): z = ( z * x) % M
    return z
```

Assume that y is Y bits long. Then the complexity of doing the binary ladder is $O(M(d)Y)$ where M(d) is the cost of multiplying two d bit numbers.

2.3.3 Greatest Common Divisor

In this paper, finding the greatest common divisor will also be a method we use lot. We can make improvements to Euclid's algorithm by making use of the following set of insights.

Theorem 2.1. *For non-negative integers x and y the following are all true.*

1. *If x and y are both even then $gcd(x, y) = 2gcd(x/2, y/2)$*
2. *If only x is even then $gcd(x, y) = gcd(x/2, y)$.*
3. *$gcd(x, y) = gcd(x - y, y$*
4. *if u and v are odd, then $|u - v|$ is even and less than $max(u, v)$*

[4] [5]

These claims should make sense. 1) is saying is $2|x$ and $2|y$ then 2 is a part of the gcd so we can divide x and y by two, calculate this new gcd, and multiply the result by two. 2) is saying if $2|x$ then $2 \not| y$ then 2 is not part of the gcd so we can forget the even part of x. 3) is the standard insight of the Euclidean algorithm, and 4) is simple algebra. We will let $e(n)$ be the number of ending zeros in our binary representation of n, i.e. $2^{e(n)}|n$ but $2^{e(n)+1} \not| n$.

```
def gcd(x,y):
    B = min(e(x), e(n))
    # get the odd parts of x and y
    x = x / (2**e(x))
    y = y >> e(x)
    while(x != y)
        (x , y) = ( min(x, y), |y - x| * (2 ** e(|y - x|)) )
    return x * (2 ** B)
```

3 Primality Proving

We start by considering that a primality test should look like "If Property P holds for n then n is prime" for a proof and likewise for a refutation "If n is prime then Property Q holds for n". We have already seen an example with trial division, "If for every prime q from 2 to $\sqrt{n}$ does not divide n then n is prime". This isn't fast though as we want a polynomial time factoring algorithm. A test simply being fast isn't sufficient to be a good test either. It should also give minimal false reports. For example, a test like "If n is prime then n

is either 2 or odd" is true and easy to compute but also very prone to reporting odd composites as possible primes. If a composites n passes a test of the form "If n is prime then Property P holds for n" we call n a P pseudo-prime. The tests we employ will start relatively simple with tests that follow from introductory abstract algebra, such as Fermat's Little Theorem. But the complexity of theory will ramp up considerably by the end of this section.

3.1 Fermat Pseudo-primes

This brings so to our first class of primality proving algorithms that we will examine in depth. The condition we will look at is based on Fermat Little Theorem. The strategy we use will start simple and grow more complex but Fermat's Little Theorem will still be the initial idea for these methods.

Theorem 3.1. *Fermat's Little Theorem If n is a prime then for any integer x,* $x^n \equiv x \mod n$

So to test if given a number n we pick a random base x and test if Fermat's Little Theorem holds for x. If it fails we have found a refutation that n is prime otherwise it may or may not be composite. For choice of base x, if n is a composite such that $x^n \equiv x \mod n$ we call n a Fermat Pseudoprime base x. This algorithm is fast, only needing to do one mod exponentiation. But running this test once there are a many Fermat pseudoprimes. The number of Fermat pseudoprimes f(n) is $o(\pi(x))$ so they are at least fairly rare. If x is a base such that $x^n \not\equiv x \mod n$, we call x a witness to n.

But in this algorithm we had a choose of base so next step would be try and run this test multiple times to uncover that a number really is a composite.

Example 3.1. *Suppose we are trying to use Fermat's Little Theorem to determine if 100 is prime. Suppose we first picked x to be 76. It turns out that* $76^{100} \equiv 76 \mod 100$. *But if we persisted an tried a different x, say 37, we find that 100 fails this test so 100 is indeed composite. As an aside, for any integer x between 2 and 99 aside from 25 and 76* $x^{100} \equiv x \mod 100$ *will fail to hold.*

Unfortunately, even this isn't enough as there are numbers n that are composite but for every base x that are Fermat pseudoprimes base x. This numbers are called Carmichael numbers. 576 is the first of these such numbers. This class of numbers is well studied.

Theorem 3.2. *Korselt Criterion An integer n is a Carmichael number if and only if n is a positive integer that is composite, positive, squarefree, and for each prime p that divides n, p -1 divides n - 1.*

There are infinitely many Carmichael numbers and for sufficiently large numbers are fairly common. In particular for a large enough n the number of Carmichael number's is greater than $n^{\frac{2}{7}}$ [6].

The Fermat test is fast but there are too many pseudoprimes and due to the existence of Carmichael numbers we can't repeat the test to get to a comfortable level of confidence. But not all hope is lost. We now move on to a similar but more useful theorem.

Theorem 3.3. *If n is an odd prime and let us express n-1 as $n-1 = 2^a b$ where b is odd. If $x \not\equiv 0 \mod n$ then one of the two statements is true.*

- $x^b \equiv 1 \mod n$
- $x^{2^i b} \equiv -1 \mod n$ *for some i such that* $0 \leq i \leq s-1$.

Proof. Suppose that n is an odd prime and we are given an x not divisible then we have

$$\begin{aligned} x &\equiv x^n \mod n \\ &\equiv x^{2^a b+1} \mod n \\ 1 &\equiv x^{2^a b} \mod n \end{aligned}$$

The only two roots of $1 \equiv y^2 \mod n$ for a prime n are $y = \pm 1$. If neither of conditions hold then $x^{2^{a-1}b} \not\equiv \pm 1$. The second condition covers the possibility of $x^{2^{a-1}b}$ being -1 . But $x^{2^{a-1}b} \not\equiv 1 \mod n$ either. If this were the case, then we have the same problem that then $x^{2^{a-2}b} \equiv \pm 1$. Continuing this argument our final statement in this chain would be $x^{2b} \equiv 1 \mod n$. This is true only if $x^b \equiv \pm 1$. But in this case either condition 1 or 2 holds here. So to satisfy Fermat's little theorem it must be the case that either of these conditions hold. □

Definition *Strong Pseudo-prime base a* We say an integer n is a strong pseudo-prime base a is n is an odd composite and 3.3 holds

Given a composite n, how likely is it that our choice of base yields that n is a strong pseudo-prime? It was proven independently by two researchers that for every odd composite n greater than 9, the number of strong pseudo-prime bases are less that or equal to $1/4\phi(n)$ where ϕ is Euler's phi function[7] [8]. This is actually a really useful statement. It says that for a composite n, at least $\frac{3}{4}$ of possible bases we can pick will refute the claim that n is prime. So if we run this test for some choice of base, there is less than $\frac{1}{4}$ chance that a composite isn't revealed to be a composite. The best part of this method is we can repeat this test for different bases to get a better bound on the chances n is an undiscovered pseudoprime. Specifically, if we repeat this test for k different bases the chance that n is a composite and not discovered in any of these tests is $\leq \frac{1}{4}^k$. Running this test for one round is the Miller Rabin test, our first algorithm that can be considered a success by itself.

```
// n is the original number, s and t are such that n -1 = 2^s t
def millerRabin(n, s, t):
    a = randint(2, n-2)
    // Strong pseudoprime test
    b = a**t % n
    if( (b == 1) or (b == n-1) ): return true
    for i in range(1, s):   //range hits 1 to s-1
        b = b**2 % n
        if( b == n - 1): return true
    return false
```

This test is fast and we if we run the test for a sufficient number of rounds we can very strongly conjecture that n is prime. Our next goal is to get to a deterministic test. We do this by asking if we try bases starting at 2 and increasing by one each time is there an upper bound for the smallest base that reveals a composite to be composite. This is called the least witness of n. There is in fact a useful upper bound provided by the following theorem.

Theorem 3.4. *If the Extended Riemann Hypothesis is true, for an odd composite n the least witness of n, W(n), is bounded by* $W(n) \leq 2ln^2n$

A proof can be found in Bach[9]. So we can test all the composites from 2 to W(n) and if it passes all of them then n is for sure prime otherwise we have found a witness to. This is better than the worst case of the randomized Miller Rabin test where we might have to test $\frac{1}{4}$ of all the possible bases before we found out that our composite n was a composite.

```
// n is the original number, s and t are such that n -1 = 2^s t
def millerRabinDetermin(n, s, t):
    W = min( floor( 2 * (log(n) ** 2) ), n -1 )
    for a in range(2, W+1):
        // Strong pseudoprime test
        b = a**t % n
        if( (b == 1) or (b == n-1) ): continue
        for i in range(1, s):   //range hits 1 to s-1
            b = b**2 % n
            if( b == n - 1): continue
```

```
            return false
        return true
```

This is our first deterministic polynomial time primality proving algorithm. But we are not satisfied since we had to rely on the unproved Riemann Hypothesis. So we will continue and until we arrive at a fully analyzed deterministic polynomial time primality-proving algorithm.

3.2 AKS Primality Proving

The last major algorithm we will cover in this section the AKS algorithm. This algorithm reaches our goal of a true deterministic polynomial time algorithm for determining if a number is prime or composite without having to rely on unproven theorems such as the Riemann Hypothesis. The AKS algorithm relies on the following theorem.

Theorem 3.5. *Given $a, n \in \mathbb{Z}$ with $gcd(a, n) = 1$ then $(x + a)^n \equiv x^n + a \mod n$ for all x if and only if n is prime.*

An extension of this idea is to pick a $f \in \mathbb{Z}[x]$ and recognize that it the case that if n is prime, $(x+a)^n \equiv x^n + a \mod (f(x), n)$ is true for every a. The type of f the AKS algorithm using is a polynomial of the form $x^r - 1$. So we can refer back to it, the polynomial we care about is

$$(x + a)^n \equiv x^n + a \mod (x^r - 1, n) \tag{1}$$

From theorem 3.5, this is true if n is prime. Sadly, this isn't necessarily always false for composites. For example, consider $r = 1, a = 1$ then we have $(x + 1)^n \equiv x^n + 1 \mod (x - 1, n)$ which can be reduced to $2^n \equiv 2 \mod (n)$ by noticing that $x \equiv 1 \mod x - 1$. This is just the Fermat pseudo-prime test base 2 which is not sufficient test to conclude that n is prime. But we can pick an r that is large enough as to avoid issues like this.

Theorem 3.6. *Agrawal, Kayal, Saxena [10] If n and r are integers such that $n \geq 2$, $gcd(n, r) = 1$, the order of n in $\mathbb{Z}/r\mathbb{Z}$ is larger than $lg^2 n$, and $(x + a)^n \equiv x^n + a(\mod x^r - 1, n)$ is true each integer a in $0 \leq a \leq \sqrt{\phi(n)} lgn$ then n is a prime power.*

This will almost work except that it's not clear that finding an r such the order of n in $\mathbb{Z}/r\mathbb{Z}$ is larger than lg^2 will be easy. Luckily, it is not an obstacle.

Theorem 3.7. *For $n \geq 3$ there exists an $r \leq lg^5 n$ such that the order of n mod r is greater than $lg^2 n$*

The AKS algorithm is then in summary:

```
AKS Primality test.
1. Check if n is a prime power and if so return false.
2. Find the least integer r such that the order of n in Z_r* is more than lg^2 n
    2a. Check if n has a factor in the range 2 to \sqrt{\phi (r) } lg n. If so return false.
3. for a from 1 to \sqrt{\phi {r} } lg n do
        if ((x + a)^n \not \equiv x^n + a ( mod x^r - 1, n)) return false
Otherwise return true.
```

The cost to check one congruence is $O(r^2 ln^3 n)$. So the total cost is $O(r^{2.5} ln^4 n)$. With our above bound this translates to $O(ln^{16.5} n)$.

But that's a pretty high degree polynomial so perhaps we can do better. Indeed there are improvements that can be made. The first is in employing more sophisticated methods to reduce $(x+a)^n \mod (x^r - 1, n)$. This brings the runtime of checking a single congruence down to $\tilde{O}(rln^2 n)$ so we can already reduce the complexity of AKS to $\tilde{O}(ln^{10.5} n)$. We can for specific values go even lower by lowering the bound on r.

Theorem 3.8. *If $n \equiv \pm 3 \mod 8$ then the value of r is bounded by $8lg^2 n$.*

Proof. Let r_2 be the least power of two such the order of n in $\mathbb{Z}/n\mathbb{Z}*$, ℓ satisfies $n^\ell \equiv 1 \mod r_2$ and $\ell \geq lg^2 n$. Let's calculate a bound on what r_2 is minimally.

$$lg^2 n \leq \ell \tag{2}$$

$$\leq 2^{k-2} \tag{3}$$

$$4lg^2 n \leq 2^k = r_2 \tag{4}$$

The worst case scenario for the largest we would have to make k to satisfy this equation is if the power of two that is half of r_2 is $4(lg^2 n) - 1$ in which case r_2 will be twice as big $r_2 \leq 8(lg^2 n) - 2$. So we conclude that if n is $\pm 3 \mod 8$ then we can choose r to be $O(ln^2 n)$. □

As for lower bounds, it is suggested by Prime Numbers: A Computational Perspective that $r \geq lg^2 n$ a likely the best bound we could hope to get for r and that the optimal total complexity is probably $O(r^{1.5} ln^3 n)$ indicating that we expected the optimal we could hope to achieve with improvements is $O(ln^6 n)$[11]. Our ability to meet this goal for general case numbers could be satisfied if either of two conjectures are proven. The first is the Artin Conjecture, if n is not a square or -1 then there are infinitely many prime p with n as a primitive root for r. From there we would just have to pick r to be just a prime that is only slightly larger than $lg^2 n$. The other open conjecture is that the primes q such that $p = 2q + 1$ is also prime are both

infinite and reasonably frequent. We would hope to find one q close to $lg^2 n$ with $r = 2q + 1$ not dividing $n \pm 1$. This would also be a valid choice for r. If either of these conjectures were true we could speed up the AKS algorithm to $\tilde{O}(ln^6 n)$.

There is one enhancement to the AKS algorithm we will consider. We can replace the $x^r - 1$ with $x^r - b$. IF we pick b correctly then it is only necessary that we verify one congruence.

Theorem 3.9. *Let n, r, d be integers with $n > 1, r|n^d - 1, r > d^2 lg^2 n$. Let $f(t)$ be a monic degree d polynomial in $\boldsymbol{Z}_n[t]$, R the ring $\boldsymbol{Z}_n[t]/(f(t))$, and $g(t) \in R$ such that $b^{n^d-1} = 1$ but $b^{(n^d-1)/p}$ is unit in R for each prime p dividing r. Then if $(x-1)^{n^d} \equiv x^{n^d} - 1 \mod x^r - b)$ in R[x], then n is a prime or prime power.*

[12] We can bound d by $(lnlnn)^{O(lnlnlnn)}$ and in doing so create a randomized $\tilde{O}(ln^4 n))$ time primality proving algorithm.

```
def AKS_Final(n):
    1. Test if n is prime power, if so return false
    2. Find a r and d such that $r d^2$ is minimal with the conditions
       that r | n^d - 1 and d^2 lg^2 n < r <= (d + 1 ) d^2 lg^2 n
    3. Choose random polynomials f(t) in the ring Z_n[t] or degree d until
       we discover that that n is composite of t^n^d \equiv t ( mod f(t) ) and t^n^(d\q) -t  is
       coprime to f(t) for each prime q | d
    4. Choose random polynomials f(t) in the ring Z_n[t] of degree less than d unit we discover
      that that n is composite or b(t)^{n^d - 1} \equiv 1 \mod f(t))  and b(t)^(n^d - 1) \q) -1
      is coprime to f(t) for each prime q | r
    5. if (x - 1)^n^d \equiv x^n^d - 1 \mod x^r - b(t) , f(t), n) return false else return true
```

4 Factoring

Now it is time for us to move from primality proving the more difficult task of factoring. For this section since we know that we can quickly prove a number composite we will assume that we are given an odd composite number with the goal of discovering a single factor of it. If we have a way to find one factor, we can reiterate this process on the now smaller number. Additionally, we can exclude the particular case where $n = p^k$ for some prime. In this section, I will discuss Pollard's p-1 method, the quadratic sieve, and the number field sieve then in the next section we will develop the mathematics needed to support the elliptic curve method for factoring.

4.1 Pollard's P-1

The first algorithm we will study is Pollard's p-1 algorithm. This method uses ideas that we have already made use of in primality proving. We already know from Fermat's Little Theorem that if p is an odd prime then $2^{p-1} \equiv 1 \mod p$. This also implies that if $p-1|M$ where M is some integer then it is the case that $2^M = 2^{k(p-1)} \equiv 1 \mod p$. So if n has a prime factor p then $p|n$ and $p|2^M - 1$ so $p|gcd(2^m - 1, n)$. So by computing this gcd, we hope to find a factor of n. The first issue is to determine what M should be. Pollard's idea is to pick M it is the least common multiple of all the integers up to some bound B. The outcome we want is that that the gcd we compute is not one or n. In the case where we find that $gcd(2^M - 1, n) = 1$ not all hope is lost. This situation suggests that we didn't pick our bound large enough and that more progress might be made if we restart with a larger bound B for M. This follow up attempt is called the second stage. We can use our previous computations so as to not have to restart the algorithm from scratch. Suppose we have already calculated M_B, the lcm of all the integers from 2 to B. We can rerun the test with $2^{q_1 M_B}$ where q_1 is the first prime larger than B. If that fails still we can choose q_2 the next smallest prime and so on. The cost of one reduction of the form $2^{q_i M_B} \mod n$ can be done in $O(lnq_i)$ time. This implies that we should use the approach of multiplying $2^{q_i M_B} \mod n$ by $(2^{q_{i-1} M_b} \mod n)$ because the gap between q_i and q_{i-1} is much less than the total size of q_{i-1}. When we discuss the Elliptic Curve Factorization Method

4.2 Quadratic Sieve Factoring

The quadratic sieve factorization(QSF) is a powerful algorithm in it's own right but also the Number Field Sieve employs many similar ideas and is even more powerful. The core idea of both is that we would like to find solutions to $x^2 \equiv 1 \mod n, x \neq \pm 1$. To see that these solutions exist, suppose that we are given an odd integer with k distinct prime factors. Using the Chinese remainder theorem we can rewrite x as the direct product of these prime subgroups. Therefore $x^2 \equiv 1 \mod n$ for each way we can express x as $x = (\pm 1)_{\mathbb{Z}/p_1\mathbb{Z}} \times (\pm 1)_{\mathbb{Z}/p_2\mathbb{Z}} \cdots \times (\pm 1)_{\mathbb{Z}/p_k\mathbb{Z}}$. That is there is 2^k possible solutions and only $\pm 1 \mod n$ are uninteresting to us. We can then use this x by noting that $n|x^2 - 1$ or equivalently $n|(x-1)(x+1)$ but from our assumption that $x \neq \pm 1$, n does not divide (x-1) nor does it divide (x+1) so $gcd(x-1, n)$ is a nontrivial factor of n.

Example 4.1. *Suppose we are trying to factor 4077. Using a black box for now to find roots of 1, we find* $3322^2 \equiv 1 \mod 4077$. $gcd(3321, 4077) = 27$. *This is correct as* $27 * 151 = 4077$.

Now all that's left is the difficult task of finding a square root of 1 which is the main difference between the two algorithms.

The first step we will take to finding a nontrivial square root of 1 mod n is to rewrite our goal to find x and y with the property that $x^2 \equiv y^2 \mod n, x \not\equiv \pm y$. This is equivalent since $(xy^{-1})^2 \equiv 1 \mod n$. We can find x and y by actually finding a sequence of x_i such that $x_i^2 = a_i$ such that $\prod a_i$ is a square. So what should we pick for these x_is? The way we will generate these x_is is to pick $ceiling(\sqrt{n}), ceiling(\sqrt{n})+1, \cdots$ but we will only keep the samples x_is such that $x_i^2 \mod n$ is B smooth. This terms will be easy to compute since for $ceiling(\sqrt{n}) \leq x \leq \sqrt{2n}$, $x^2 \mod n$ is given by $x^2 - n$. And furthermore, the residues we encounter first will be small and then gradually get bigger. This is good because small residues means we have a better chance of finding more B smooth numbers. Collecting x_is in the manner above, the question of how many x_is we need to guarantee that the for some subset S of them that $\prod_{x_i \in S} x_i = a^2$ for some a can be answered easily. The insight is that if we write each $x_i^2 \mod n$ as a product of primes $x_i = p_1^{a_{i,1}} p_2^{a_{i,2}} \cdots p_j^{a_{i,j}}$ then what we care about really is just the exponents. Making use of the restriction that all the numbers we picked will be B-smooth the number of primes is fixed at $\pi(B)$. We now write these exponents as a vector $v_i = a_{i,1}, a_{i,2}, \cdots, a_{i,\pi(B)}$ allowing for some of these elements to be 0 if the prime they would correspond to is not a factor of x_i. We are nearly there. Notice that multiplying $(x_i^2 \mod n)$s terms together can be expressed as adding their respective exponent vectors. A product of a subset of these vectors will be a square if this sum of vectors has components that are all even. So we only care about our exponent vectors mod 2. So our problem is how many vectors of length $\pi(B)$ do we need to ensure that a subset of these vectors sums to the zero vector mod 2. But this just a linear algebra question! This is just asking if we have a matrix of vectors of length k how many vectors are needed to ensure that there is a linearly dependent subset. With each vector having length $\pi(B)$, we are guaranteed to have a linear dependency with $\pi(B)+1$ rows.

But some important details have been left unattended. Namely, "How do we quickly find the B-smooth numbers that we desire?" and "What should B be?" are appropriate questions to currently have. The answer to the first question is given away by the name of the algorithm, we use a sieve. We can adapt the standard sieving algorithm for use in finding B-smooth numbers and even to get their full factorization. A good heuristic for what the optimal B in theory is given by B $\approx exp(\frac{1}{2}\sqrt{ln(n)ln(ln(n))}$ and with this choice the run time of the sieving step is $O(B^{2+\epsilon})$ [11]. We will see the function below frequently in this section so will name it $L(n)$.

$$L(n) = e^{\sqrt{(ln(n)ln(ln(n))}} \tag{5}$$

This is the square of what we suggested the optimal B to be. Provided we can do the linear algebra step in $O(B^{2+\epsilon})$ as well then the running time of the entire algorithm is $O(B^{2+\epsilon})$. This isn't polynomial time but it is subexponential as it grows slowly than n. But it's not clear how given a matrix of exponent vectors we going about find a nonempty set that is linearly dependent. Standard Gaussian elimination will give us an

answer in $O(B^3)$. Although can can do better for smaller n that we are factoring this will actually be fine in practice. This is especially true because the matrix will likely be sparsely populated which can be taken advantage of, reducing the practical time bound.

There are three good algorithms that we can use to replace Gaussian elimination in the sparse case. They are the conjugate gradient method, the Lanczos method, and the coordinate recurrence method. A full discussion of these methods can be found in the conference paper by Odlyzko[13]. But we will give a short description of the conjugate gradient method here for a taste. Suppose that we are given an $N \times N$ matrix A that is real, symmetric, and positive definite and we wish to find the vector x that solves $Ax = y$. Don't worry for now that our matrix can't be assumed to be symmetric. Then the conjugate gradient method is as follows

```
def congrad(A, y):
    x[0] = a random vector of length n
    p[0] = r[0] = y - A*x[0]
    for i in range(N):
        temp = inner_product(r[i], r[i]
        a[i] = temp / inner_product(p[i], A * p[i])
        x[i+1] = x[i] + a[i]*p[i]
        r[i+1] = r[i] - a[i]*(A * p[i])
        b = inner_product(r[i+1], r[i+1])/ temp
        p[i+1] = r[i+1] + (b * p[i]
        if( r[i+ 1 ] == 0):
            return x[i+1]
```

It has been proved if our algorithm doesn't lose precision $r[i] = 0$ for some i our range[14]. Furthermore, this algorithm runs in time $O(B^{2+\epsilon})$. But our matrix isn't symmetric and in fact it isn't even square. This is a minor problem. If our original system if $Ax = y$ then instead we can solve for $Bx = z$ where $B = A^T A$ and $z = A^T y$.

We are now ready present the Quadratic Sieve Method fully.

```
def quadsieve(n):
    B = ceiling( L(n) ** (1/2)
    p[0] = 2
    a[0] = 1
    i = 1
```

```
For each odd prime <= B:
    if(\frac{n}{p} == 1):
        a[i] = p
        i += 1
for j in range(2, k+1):
    find roots a[j] of a[j]^2 \equiv n mod p_i
x = ceiling(sqrt{n)}
S = K+1 Sieved pairs (x, x^2 - n) such that x^2 -n is B smooth
A = K+1 X K matrix with each row being the exponent vector of x^2 -n
    which we know from sieving
use linear algebra to set a subset of rows that sum to the zero vector mod 2
xp = product([a for (a,b) in S]) % n
y = sqrt(proudct([b for (a,b) in S] ))
# This is easy because we know the factorization of this square.
d = gcd ( x - y, n)
```

We can for practical use improve the quadratic sieve further. In practice, memory usage during the sieving step is a major constraint. If we have a large smoothness bound then we will need a larger matrix to contain all vectors that we have collected. This introduces the large prime variation of the quadratic sieve that allows us to in effect way cheat cheat and use a larger smoothness bound without the extra cost. As the name hints, in this variation we will allow numbers that are almost B smooth with the exception that they have one prime factor that is larger than B. If we set the bound for this larger prime factor to B^2 then finding these numbers as well during sieving isn't hard. If we have already factored all the primes less than B from some integer n and after it is greater than 1 and less than B^2 then the factored remains of n is prime. So n itself is almost B smooth. But we shouldn't include them directly in our matrix, after all we just complained that increasing the size of the matrix can be a major space constraint. Instead for each almost smooth number we will first store this additional prime as an associated value to each number. Once we have collected all of our samples to fill out our matrix, we can sort the entries that do have a large prime. Next, we can scan through the these primes and if a large prime appears only once it can't be part of our dependent set so we remove that entry. Otherwise if we have k copies the same large prime P then for each x_i that is contributing a prime we notice that $x_i - N = y_i P$. But then $(x_1 x_i)^2 \equiv y_1 y_i P^2 \mod n$ for $i = 2 \cdots k$ that in this product the P has been 'canceled out'. We can add these $y_1 y_i$ as exponent vectors and increase the number of elements in our matrix with little hassle. A birthday paradox like effect makes the probability

that we have multiple entries that are almost B smooth and share the same extra prime is not uncommon and only becomes more likely as we generate larger list of primes. In practice, researchers have experimented with allowing two large prime factors to try and continue this logic. It was in fact this method of use two primes that resulted in the large number in the introduction, RSA-129, being factored [2].

4.3 Number Field Sieve

The number field sieve shares many of same characteristics of the quadratic sieve. For the Number field sieve we will still want to find $x^2 \equiv y^2 \mod n$, we will still use exponent vectors and the linear algebra that doing so leads of to, and we will still sieve something. One of the key things that we liked about quadratic sieve was that the $x_i^2 \mod n$s were small. In the number field sieve we will pursue so this idea of having small residues by lifting the restriction that we are only considering squares. This will take some time to develop but it will be worth the invest since the NFS is the fastest factoring we have for 'worst' case factor. In this context, worst case factoring means that we are factoring a composite that is two approximately equal sized primes.

When we developed the quadratic sieve, we already had the x side of our goal of finding x and y such that $x^2 \equiv y^2 \mod n$ for free. Here neither side is free. Our starting place will be to generate pairs $(x_i, f(x_i))$ where x_i is in some algebraic polynomial ring $Z[a]$ and $f : Z[a] \mapsto \mathbb{Z}/N\mathbb{Z}$ and f is a homomorphism. We will look for $x_1 \cdots x_k \in Z[a]$ such that their product is a square, call it c^2, in the ring and the product of $f(x_1) * f(x_2) \cdots f(x_k)$ is simultaneously a square d^2 in $\mathbb{Z}/n\mathbb{Z}$. If we could accomplish this then $f(c)^2 \equiv f(c^2) \equiv f(x_1 x_2 \cdots x_k) \equiv f(x_1)f(x_2) \cdots f(x_k) \equiv d^2 \mod n$ then we could factor n with $gcd(d - c, n)$.

So the next task is how do we pick this ring and the homomorphism. We do this by first picking a degree d for our polynomial. Next we set $b = floor(n^{\frac{1}{d}})$ and write n in base b like $n = b^d + c_{d-1}b^{d-1} + \cdots + c_0$. It will suffice for us to let $g(x) = x^d + c_{d-1}x^{d-1} + \cdots c_0$ and our ring will be $Z[x]/g(x)$. We will sometimes consider the homogeneous form of g(x), $G(x, y) = x^d + c_{d-1}x^{d-1}y + \cdots c_0 y^d$. g is a useful creation because we can later use that $g(b) \equiv 0 \mod n$ since $g(b) = n$. Let α be a root of g. Our homomorphism can simply be the evaluation map by evaluating our polynomial at α and then reducing $\mod n$. We will also want g(x) to be irreducible but actually this isn't a problem. If we have discovered that $g(x) = h(x)I(x)$ with h and I being nontrivial then we have a factorization of $n = h(m)I(m)$ [15]. So we should check to see if there is a nontrivial factor of g(x), if there is great, we have a way to find a factor of n, and if not we continue with our algorithm.

A little bit more background is needed before we can proceed with the algorithm. Given an element in our polynomial ring $Z[\alpha]$, $E = c_0 + c_1 a + c_2 a^2 + \cdots + c_{d-1}a^{d-1}$ we define Norm N(E) to be $\prod_{\alpha_i} E(\alpha_i)$.

This is actually an integer. It is also true that this function is multiplicative, $N(E_1E_2) = N(E_1)N(E_2)$. Furthermore, we have the following property:

$$N(x - ya) = (x - ya_1)(x - ya_2)\cdots(x - ya_d) \tag{6}$$
$$= y^d(x/y - a_1)(x/y - a_2)\cdots(x/y - a_d) \tag{7}$$
$$= y^d g(x/y) \tag{8}$$

We will let G(x,y) be the homogeneous form of g(x), that is $G(x, y) = x^d + c_{d-1}yx^{d-1} + \cdots c_0y^d$. So we also have that $y^d g(x/y) = N(x - ya)$

We are on to the step where actually find something to sieve. We will find pairs of integers (x, y) with $x, y \in \mathbb{Z}$ such that $\prod_{(x,y)}(x - y\alpha) = c^2$ and $\prod_{(x,y)}(x - ym) = d^2$. For each prime p, we define $R(p) = \{r \in [0, p - 1]|g(r) \equiv 0 \mod p\}$. This is useful because if $gcd(a, b) = 1$ then $G(a, b) \equiv 0 \mod p$ if and only if $a \equiv br \mod p$ for some R(p). Our sieving strategy will be to fix b . Each vector will have entries that correspond to the pairs p,r where p is prime and $r \in R(p)$.

Example 4.2. *Consider the polynomial $g(x) = x^2 + x + 1$. First let of find the non empty R(p)s.*

$$g(1) = 1 + 1 + 1 \equiv 0 \mod 3 \tag{9}$$
$$g(2) = 4 + 2 + 1 \equiv 0 \mod 7 \tag{10}$$
$$g(4) = 16 + 4 + 1 \equiv 0 \mod 7 \tag{11}$$

Using the pairs (1,3), (2,7), and (4,7) as coordinates we generate vectors of other numbers now.

$$G(1, -2) = 3 \text{ and since } 1 \equiv -2 \mod 3 \text{ This vector is (1,0,0)} \tag{12}$$
$$G(3, 5) = 7^2 \text{ since } 3 \equiv 5 * 2 \mod 7 \text{ This vector is (0,1,0)} \tag{13}$$
$$G(1, -5) = 21 \text{ since } 1 \equiv -5 * 4 \mod 7, 1 \equiv -5 \mod 3 \text{ This vector is (1,0,1)} \tag{14}$$
$$G(1, 4) = 21 \text{ since } 1 \equiv 2 * 4 \mod 7, 1 \equiv 4 \mod 3 \text{ This vector is (1,1,0)} \tag{15}$$

With α be a root of g, this tells us that $(1 - 2\alpha)(3 + 5\alpha)(1 + 4\alpha)$ is a square in $Z[\alpha]$.

Theorem 4.1. *If have a set of a, b pairs that are co-prime and each $a - b\alpha$ is B-smooth and the product $\prod_{(a_i,b_i)}(a_i - b_i\alpha$ is a square in the ring of algebraic integers in Q[α] then $\sum_{(a_i,b_i)} vec(a - b\alpha) \equiv 0 \mod 2$.*

This constitutes the core strategy of the NFS and for a full treatment into how to fix some of the technical wrinkles see Prime Numbers[11]. We will not present the full algorithm of the number field sieve.

```
# n is an n composite that is not a prime power
def NFS(n):
    d = floor ( (3 * ln n ) / ( ln ( ln ( n ) ) ** (1/3)))
    B = floor ( exp ( (8/9)**(1/3) * (ln n)** (1/3)  * ( ln ( ln ( n) ) ) ** (2/3) ))
    m = floor ( n ** ( 1/d) )
    #write n in base m and collect coefficients c[0] through c[d-1]
    c = writeInBase(n , m )
    def f(x):
        return ( x** d) + (c[d-1] * x**(d-1) + ... + c[0]
    # check if f is reducible and is so return early
    if(reduicble(f) ):
        return factors(f)
    # Homogenize turns a polynomail of one variable into a =omogeneious one with 2 variables
    F(x,y) = Homogenize(x,y)
    G(x,y) = x - (m*y)
    # genprimes gets the primes below a bound B
    P = genprimes(B)
    for p in P:
        R[p] = { r for r in range(0,p) if (f(r) % p == 0}
    k = floor ( 3 * lg n )
    Q = The first k primes q[i] , . . . , q[k] > B such that R(q[i] ) contains some element
    s[j] with f'(s[j] ) =- 0 (mod q[j] ), storing the k pairs (q[j] , s[j] )
    B' = sum( [len(R(p)) for p in P]
    V = 1 + pi(B) + B' + k
    M = B
    #Sieve(F,G, B, k) finds a set of comprime
    #integer pairs with with F(a,b) G(a,b) being B-smooth until k are collected
    S = {}
    while(len(S) < V):
        S = Sieve(F,G,B, V)
        M *= 2
    # Complute matrix
    A = []
```

```
# For we (a,b) pair of S we construct the matrix as follows
for (a,b) in S:
    v = [0]*V
    if( G(a,b) < 0 ): v[0] =1
    k = 1
    for p in P:
        gamma = # exponent of p^gamma in the factorization of abs(G(a,b))
        if( gamma % 2 == 1 ): v[k] = 1
        k += 1
    for (p,r) in R:
        if(v[p,r] % 2 == 1): v[k] = 1
        k++
    for (s, j) in Q:
        if( char(a - b*[s]/ q) == -1): v[k] = 1
        k++
    A.append(v)
# Use linear algebra like conjuage gradient to find a subset taht somes to the zero vector
S' = keyvectors(A)
v = squareroot( prudct[ a - bm for (a,b) in S)] , n)
u = findroot(f'(alpha)**2 * product [ a - b alpha for (a,b ) in S] )
return gcd(u - f'(m) v, n)
```

Was all this effort worth it? The bound for the expected value of numbers will will have to sieve until we find a square is given by $L(X)^{\sqrt{2}+o(1)}$ if we want a smoothness bound of $B = L(x)^{\frac{1}{\sqrt{2}}}$[16]. With the optimal choice of degree d being $d \approx \frac{3lnn}{lnlnn}^{\frac{1}{3}}$ we need that X is given by $lnX \approx \frac{4}{3^{\frac{1}{3}}}(lnn)^{\frac{2}{3}}(lnlnn)^{\frac{1}{3}}$. If one carries out all the arithmetic involved this brings the total complexity to $exp(((\frac{64}{9})^{\frac{1}{3}} + o(1))(lnn)^{\frac{1}{3}}(lnlnn)^{\frac{2}{3}})$. Asymptotically, this is a strong improvement over the quadratic sieve. As it stands, the number field sieve if the fast algorithm that we know of for factoring numbers that are made up of only large primes. For this reason many of the RSA cryptography challenges have been cracked by the quadratic and later the number field sieve. But as we will see in the next section we have a more powerful tool for factoring numbers more generally.

5 Elliptic Curves

The last section that we will discuss is the Elliptic Curve Method. Although the Number Field Sieve is preferred to the Elliptic Curve Method for worst case factoring, this case in general is rather uncommon. Most large composites are made up of several primes. Even in this worst case, the Elliptic Curve Method is asymptotically the same as the Quadratic Sieve. Especially since we often don't know a priori that a composite that we are handed is one of these rare worst case primes, the elliptic curve method is seen as the best general factoring algorithms.

5.1 Fundamental Definitions

The study of elliptic curves involves solutions to a specific class of cubic polynomials. Specifically, we care about cubics of the form:

$$y^2 = x^3 + ax^2 + bx + c \tag{16}$$

where all of the coefficients are rational numbers. Equations of this form are said to be in Weierstrass form. If the roots of $y^2 = f(x) = x^3 + ax^2 + bx + c$ are distinct are distinct. When this happens we all the curve non-singular.

Our study of elliptic curves by considering their geometric properties This interpretation is that if we already have two points with rational coordinates then the line joining them will intersect the curve at a third point that is also a rational. There are a couple small adjustments to be made to this statement. First, if one our our points is a point of tangency, the line we draw is the tangent line of the curve at that point. In this way is P is a tangent this is what we will consider the line that touches P twice. Furthermore we need a way to handle vertical lines as it seems these will only intersect the curve twice. We fix this by adding a point at infinity that is where vertical lines meeting. We will call this special point O. We also allow for this point to be considered a rational point.

This is leading to a group operation that we can perform on the points on an elliptic curve. Let's call the operation of given two rational points P and Q finding third point that intersects this curve P op Q. For this to be a group we will need an identity I . But in general, P op I isn't P so our operation needs to be more involved. The operation for us is instead (P op Q) op I. To be a group it would be mathematically valid to pick any rational point to be I as the (P op I) op I is the third point of the line goes through the I and the point on the curve that intersects P and I. This can only be P. But for the remainder of this paper, we will fix $I := O$ and write (P op Q) op O as $P + Q$. Geometrically the operation $P + Q$ first finds the third point of the line joining P and Q and then since O is the point where vertical lines meet, the line we are trying to

find a third point of is a vertical line. The final result is P op Q except flipped across the x axis. It be useful to give an explicit formula for $P + Q$. If $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ then the slope s of the line joining P to Q is $s = \frac{y_2 - y_1}{x_2 - x_1}$. The equation of the line is $f(x) = s * x + c$ where $c = y_1 - sx_1 = y_2 - sx_2$. We can plug this back into our original cubic equation to find (x_3, y_3). If we do and perform the necessary algebra we find that $x_3 = s^2 - a - x_1 - x_2$ and $y_3 = sx_3 + c$.

But it should be clear that this is not a good formula to use if $P = Q$ in this case our slope would be undefined. Geometrically, we assigned P op P to be the tangent line at the curve P so in this case we can calculate the tangency to the curve at point P.

$$y^2 = x^3 + ax^2 + bx + c \tag{17}$$

$$\frac{d}{dx}y^2 = 3x^2 + 2ax + b \tag{18}$$

$$2yy' = 3x^2 + 2ax + b \tag{19}$$

$$y' = \frac{3ax^2 + 2ax + b}{2y} \tag{20}$$

So the slope for our for our line going through P with is $\frac{3ax_p^2 + 2ax_p + b}{2y_p}$, to make this line go through P $c = y_p - sx_p$. Putting this together we solve for the x coordinate of these result of $2P$. This will be called the duplication formula.

$$x_3 = \frac{x^4 - 2bx^2 - 8cx + b^2 - 4ac}{4x^3 + 4AX^2 + 4BX + 4C} \qquad \text{(Duplication formula)}$$

Additionally, this group is Abelian, associative, and the inverse of (x, y) is $(x, -y)$.

This group has some interesting properties. For example what are the points of order 2? In order for a point to have order two it must be the case that $P = -P$ or $(x, y) = (x, -y)$ so a point has order exactly 2 only when $y = 0$. So there are three elements of order 2, one for each of the roots of this cubic. More interestingly is what we can say about the set of points that are rational.

Theorem 5.1. *Nagell-Lutz Theorem Let $y^2 = x^3 + ax^2 + bx + c$ be a non-singular curve with integer coefficients and let $D = -4a^3c + a^2b^2 + 18abc - 4b^3 - 27c^2$ then if P=(x,y) is a rational point point of finite it is the case that both x and y are integers. Furthermore, either y=0 or $y|D$.*

But initially we defined an elliptic curve to allow rational coefficients not just integer coefficients. This

isn't an issue though. We can do the followin:

$$y^2 = x^3 + ax^2 + bx + c \tag{21}$$

$$d^6y^2 = d^6(x^3 + ax^2 + bx + c) \tag{22}$$

$$(d^3y)^2 = (d^2x)^3 + d^2a(d^2x)^2 + d^4b(d^2x) + cd^6 \tag{23}$$

Our new cubic will be defined by $X = d^2x$ and $Y = d^3y$ with $Y^2 = X^3 + d^2aX^2 + d^4bX + cd^6$. This a sufficiently large d we can clear out the denominators of a,b, and c. The discriminant of a cubic curve is $D = -4a^3c + a^2b^2 + 18abc - 4b^3 - 4b^3 - 27c^2$. This will be useful because it is a known result that $D = (a_1 - a_2)^2(a_1 - a_3)^2(a_2 - a_3)^2$ where a_1, a_2, a_3 are roots of f(x)[17]). So $D \neq 0$ if and only if all three roots are distinct. This gives us an easy way to check that a curve really is non-singular.

Know that we have established a strong foundation of elliptic curves in general its time we build up our knowledge of how these curves work when finite fields are applied. Specifically solutions to curves now will be (x, y) with $x, y \in \mathbb{F}_{\text{ı}}$ for some integer p. For now we will assume that this is p is a prime or prime power. This does ruin the geometric interpretation we had but the explicit formulas are still true as well as all of the group structure. This is finite group as there are only a finite number of distinct (x,y) tuples not even worrying about if they are solutions to the curve. But we give a better estimate for how many solutions there are in the finite field case. We remember that roughly half of the elements in $1, 2 \cdots, p-1$ will be squares and the other half will not be squares. This leads to three possibilities to consider depending on the value of $f(x) = x^3 + ax^2 + bx + c$. One, if f(x) evaluates to 0, the only choice for y is also zero. Next, if $f(x) = d^2$ for some a then $y = \pm a$ so there are two solutions. Finally, if $f(x) = e$ where e is not a square then we can't hope to find a y such that $y^2 \equiv e \mod p$. So this gives us a rough estimate that the number of solution should be approximately, $0.5 * 0 + 0.5 * 2 +$ # of solutions to f(x) + 1 with the one coming from O. In fact this estimate is close.

Theorem 5.2. *Hasse-Weil Theorem for Elliptic Curves If C is a non-singular, irreducible, elliptic curve of the field $\mathbb{F}_p$, then the the number of points, N, on on C is between $p + 1 - 2\sqrt{p} \leq N \leq p + 1 + 2\sqrt{p}$.*

In reality, this the proof of this bound is far more complicated and relies on Weil Conjectures being proven true.

5.2 Elliptic Curve factoring Method

We have developed at the core mathematics we need to do the elliptic curve method. As suggested earlier the idea of the ECM is in analogy with Pollard's p - 1 except we perform the operations on elliptic curves in

the domain $\mathbb{F}_n$ instead of Zmn. This is not a field and in fact this algorithm hopes to find a case where the group law will fail. The biggest advantage that this method has over Pollard's p - 1 is that when we failed with Pollard's p -1 we had to give up but here we can pick a new curve to work with and try again.

5.2.1 The Basic Algorithm

We will compute kP with a doubling ladder type system.

```
def mult(k, P):
    Q = P
    while(k > 1):
        if( k % 2 == 1):
            Q = P + Q
            k -= 1
        Q = double(Q)
        k /= 2
    return Q
```

In the case when we were working with a field this is fine but now that we have moved $\mathbb{Z}/n\mathbb{Z}$ where n is composite this might not work. Specifically, we might be asked to compute $s = \frac{y_2 - y_1}{x_2 - x_1} \mod n$ in this group. But $\mod n$ there isn't always an inverse to $x_2 - x_1$. The trick of the ECM factoring method is that we want to find this group value. When we are ask to find this inverse there are three things that may happen. First, if $gcd(x_2 - x_1, n) = 1$ then an inverse exists and we are able to perform this addition. Second which is the case we want, if $n > gcd(x_2 - x_1, n) > 1$ then an inverse doesn't exist but the gcd is a non trivial factor of n so our algorithm succeeded. Finally, if $gcd(x_2 - x_1, n) = n$, we have been unlucky and we choice a new curve. The last case corresponds to the failure case in Pollard's p-1.

This is the general plan but three are three questions to be answered still.

1. What elliptic curves should we look at?
2. How do we go about finding a point on this curve?
3. What do we multiply this point by?

For the basic variant of the ECM the best way a curve is actually to pick on that makes answering question 2) easy. One way to accomplish this is to pick the point (x_1, y_1) and make a curve that hints this point. Our curve is $y_1^2 \equiv x_1^3 + bx_1 + c$ so we still have to parameters to choose, b and c. We can pick b randomly and

assign C to be $c \equiv y_1^2 - x_1^3 - bx_1$. That answers questions 2 and 3. If the number of elements in $C(\mathbb{F}_p))$ k then $kP = 0$ so to we pick k to be $p_1^{a_1} p_2^{a_2} \cdots p_\ell^{a_\ell}$ such that $p_i^(a_i) \leq B$ for some smoothness bound B.

```
def ecm(n):
    B = 10000 #or whatever smoothness we want
    if(p = isPerfectPower(n) != 0 ): return p
    x = randint(0, n-1)
    y = randint(0, n-1)
    b = randint(0, n-1)
    c = ( (y**2) - (x**3) - (a *x) % n)
    P = (x, y)
    for i in range(1, pi(B))
         # Find the largest integer a such that p[i] ** a <= B
         mx = findmax(p, B)
         for j in range(1, a):
             #mult' is mult but halts is we are asked to compute inv(d) which doesn't exist.
             # flag indicates if this happened and d is the
             # value that led to this illegal inverse operation
             [P, flag, d] = mult'(p[i] * P)
             if(flag):
                 return gcd(n,d)
```

The expected complexity of the ECM depends on the size of the least prime factor of n. Specially, the expected runtime is $L(p)^{\sqrt{2}+o(1)}$. So the worst case is if we are asked to factor a composite made up of two roughly equal sized primes. In this case the expected run time is $L(n)^{1+o(1)}$ the same runtime as the quadratic sieve.

6 Conclusion

In this paper, we have seen how the fundamental methods of modern day factoring and primality proving have developed. There are several different paths that could be taken after reading this paper. For the computationally interested, there are several improvements that can be effected to take this methods further. There are improvements that can be made to the quadratic sieve[18][19], the number field sieve[20], and the Elliptic Curve Method[21][22]. The is also the open question on if there are better tests that are based on

the ideas that we have already explored here like how the elliptic curve method works on a similar goal as Pollard's p-1.

References

[1] M. Gardner, "Mathematical games," *Scientific American*, vol. 237, no. 2, pp. 120–125, 1977.

[2] D. Atkins, M. Graff, A. K. Lenstra, and P. C. Leyland, "The magic words are squeamish ossifrage," in *International Conference on the Theory and Application of Cryptology*, pp. 261–277, Springer, 1994.

[3] D. J. BERNSTEIN, "Fast multiplication and its applications,"

[4] J. Stein, "Computational problems associated with racah algebra," *Journal of Computational Physics*, vol. 1, no. 3, pp. 397–405, 1967.

[5] D. E. Knuth, *The art of computer programming*, vol. 3. Pearson Education, 1997.

[6] W. R. Alford, A. Granville, and C. Pomerance, "There are infinitely many carmichael numbers," *Annals of Mathematics*, pp. 703–722, 1994.

[7] L. Monier, "Evaluation and comparison of two efficient probabilistic primality testing algorithms," *Theoretical Computer Science*, vol. 12, no. 1, pp. 97–108, 1980.

[8] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of number theory*, vol. 12, no. 1, pp. 128–138, 1980.

[9] E. Bach, *Analytic methods in the analysis and design of number-theoretic algorithms*. MIT press Cambridge, 1985.

[10] M. Agrawal, N. Kayal, and N. Saxena, "Primes is in p," *Annals of Mathematics*, vol. 160, no. 2, pp. 781–793, 2004.

[11] R. Crandall and C. B. Pomerance, *Prime numbers: a computational perspective*, vol. 182. Springer Science & Business Media, 2006.

[12] F. Guccini, T. E. Engeler, E. Specker, and J. Waldvogel, "Efficient "quasi"-deterministic primality test improving aks,"

[13] A. M. Odlyzko, "Discrete logarithms in finite fields and their cryptographic significance," in *Advances in Cryptology* (T. Beth, N. Cot, and I. Ingemarsson, eds.), (Berlin, Heidelberg), pp. 224–314, Springer Berlin Heidelberg, 1985.

[14] M. R. Hestenes, E. Stiefel, *et al.*, "Methods of conjugate gradients for solving linear systems," *Journal of research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, 1952.

[15] J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman, and S. Wagstaff Jr, *Factorizations of* $b^n \pm 1, b = 2, 3, 5, 6, 7, 10, 11, 12$ *Up to High Powers*, vol. 22. American Mathematical Society, 1988.

[16] C. Pomerance, "Multiplicative independence for random integers," *PROGRESS IN MATHEMATICS-BOSTON-*, vol. 139, pp. 703–712, 1996.

[17] J. H. Silverman and J. T. Tate, *Rational points on elliptic curves*, vol. 9. Springer, 1992.

[18] C. Pomerance, "The quadratic sieve factoring algorithm," in *Workshop on the Theory and Application of of Cryptographic Techniques*, pp. 169–182, Springer, 1984.

[19] A. K. Lenstra and M. S. Manasse, "Factoring with two large primes," *mathematics of computation*, vol. 63, no. 208, pp. 785–798, 1994.

[20] B. Murphy, "Modelling the yield of number field sieve polynomials," in *International Algorithmic Number Theory Symposium*, pp. 137–150, Springer, 1998.

[21] P. L. Montgomery, *An FFT extension of the elliptic curve method of factorization*. PhD thesis, UCLA, 1992.

[22] R. Brent, R. Crandall, K. Dilcher, and C. Van Halewyn, "Three new factors of fermat numbers," *Mathematics of computation*, vol. 69, no. 231, pp. 1297–1304, 2000.