Elliptic Curves and Factorization Algorithms

Bryce Taylor

May 10, 2015

1 Introduction

2 Specific Factorization Algorithms

2.1 Quadratic Sieve

2.1.1 Chinese Remainder Theorem

A common result from elementary number theory that we will make use of in our algorithm analysis is a way to combine solutions to linear modular equations. The Chinese Remainder Theorem tells us precisely this; it is stated below in a form specific to the type of analysis we will be doing:

Theorem 2.1 (Chinese Remainder Theorem)**.** *Let $n_1, n_2, \cdots n_k$ be k pairwise relatively prime integers. Let a_i be a residue mod n_i. Then, there is a unique solution modulo $n_1 n_2 \cdots n_k$ to the following system of equations*

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

2.1.2 Quadratic Residues

Definition 2.2 (Quadratic Residue)**.** A residue $a \pmod n$ is a Quadratic Residue if and only if there exists some residue $b \pmod n$ such that $b^2 \equiv a \pmod n$.

Lemma 2.3. *Let p be an odd prime. Then there are $\frac{p+1}{2}$ Quadratic Residues $\pmod p$.*

Proof. Let $Q = \{0^2, 1^2, \cdots (p-1)^2\} \pmod p$ be the squares of all residues modulo p. Note that Q contains precisely the Quadratic Residues (with repetition) modulo p. Notice that for $a \neq 0$, $a \not\equiv -a \pmod p$, and $a^2 \equiv (-a)^2 \pmod p$. This tells us that each non-zero element in Q is included at least twice. Thus, there can be at most $\frac{p+1}{2}$ distinct residues in Q. Now, assume for the sake of contradiction that some residue is included more than twice in Q. This means we have a, b such that $a \not\equiv b, -b \pmod p$ and $a^2 \equiv b^2 \pmod p$. Rearranging, this means that $(a-b)(a+b) \equiv 0 \pmod p$. But p is a prime, so either $p|a-b$ or $p|a+b$, which is a contradiction so our assumption must be false and no residue is included more than twice in Q. Finally, notice that 0 is included exactly once since if $a^2 \equiv 0 \pmod p$, then $a \equiv 0 \pmod p$ since p is a prime. Therefore, there are at least $\frac{p+1}{2}$ Quadratic Residues mod p. Combining this with our prior upper bound tells us that there are exactly $\frac{p+1}{2}$ Quadratic Residues mod p. □

Theorem 2.4. *Let n be a product of k distinct odd primes, $n = p_1 \cdots p_k$. Then there are at least $\frac{n}{2^k}$ Quadratic Residues modulo n.*

Proof. As in the proof of the above lemma, we consider the multiplicity of each Quadratic residue mod n, i.e. for a a quadratic residue $\pmod n$, we count the number of solutions to $x^2 \equiv a \pmod n$. By the Chinese Remainder Theorem, solutions to this are the same as solutions to the system of equations:

$$\begin{aligned} x^2 &\equiv a \pmod{p_1} \\ &\vdots \\ x^2 &\equiv a \pmod{p_k} \end{aligned}$$

Each of these k equations has at most two solutions for x by **??** Each combination of solutions provides a unique solution for x modulo n by the Chinese Remainder Theorem, so since there are at most 2^k combinations of solutions, there are at most 2^k solutions to $x^2 \equiv a \pmod n$, i.e. the multiplicity of each quadratic residue $\pmod n$ is at most 2^k. Since there are n total residues, this means that at least $\frac{n}{2^k}$ must be quadratic residues as desired. □

2.1.3 Quadratic Sieve Introduction

An alternative to elliptic curve factorization is the Quadratic Sieve Method. The basic idea is quite simple: find a pair of integers a and b such that $a^2 \equiv b^2 \pmod{n}$. Then $(a-b)(a+b) \equiv 0 \pmod{n}$ and hence it is likely that either $(a-b, n) \neq 1, n$ or $(a+b, n) \neq 1, n$ in which case this leads to a factorization of n. The factors are easily computed as $GCD(n, a-b)$ or $GCD(n, a+b)$.

Generating some square residues is easy: we can let a be anything and then a^2 will be one of our squares mod n. The idea of the Quadratic Sieve is to construct another residue (b) that is not $\pm a$ so that $b^2 \equiv a^2 \pmod{n}$. Another way to think about this is that given a residue $a^2 \pmod{n}$, we wish to construct a non-trivial square root of $a^2 \pmod{n}$. One way to achieve this is if $a^2 \pmod{n}$ is itself a perfect square; then we can simply take its square root (as an integer) to get b. Of course, the chances of randomly choosing a value of a so that $a^2 \pmod{n}$ is also a perfect square are very small, so instead, we find several of these residues that multiply together to get a perfect square. This is best shown in the following example:

Example 2.5 (Factoring 323)**.** We wish to factor 323. Choose residues as follows:

a_i	$a_i^2 \pmod{323}$	Residue Factorization
271	120	$2^3 \times 3 \times 5$
145	30	$2 \times 3 \times 5$
194	168	$2^3 \times 3 \times 7$
31	315	$3^2 \times 5 \times 7$
137	35	5×7

Notice that the particular values of a_i we chose all have square residues that factor into the product of small primes. This lets us figure out which residues to multiply together to get a perfect square. In this case, we computed 5 such residues with all prime factors at most 7 in order to guarantee that a product of some of them is a perfect square. One such product is the first two, which gives us the relation

$$(271 \cdot 145)^2 \equiv 60^2 \pmod{323}$$

This gives us a factorization since $GCD(271 \cdot 145 - 60, 143) = GCD(39235, 323) = 19$. In general, we would need to multiply together more than just two residues (even in this example, we could have also chosen 30, 168, and 137), but this example gives the basic idea of the quadratic sieve. The residues in the above table were in fact found by our python implementation of the Quadratic Sieve.

2.1.4 Quadratic Sieve Algorithm

Let y be a given parameter and let $k = \pi(y) + 1$, where $\pi(x)$ is the number of primes less than or equal to x. Let $p_1, \cdots, p_{k-1}$ be the primes that are at most y. To find the pair (a, b), we use the following high-level algorithm:

1. Choose k values of a_i such that the following conditions hold:
   - $b_i = a_i^2 \pmod{n} \neq a_i^2$, i.e. $a_i^2 \geq n$
   - b_i is y-smooth.
2. Compute a subset of the b_i whose product is a perfect square; let the set of indices of this set be I.
3. Let $a = \prod_{i \in I} a_i$ and $b = \left(\prod_{i \in I} b_i \right)^{1/2}$.
4. If $GCD(n, a-b) \neq 1, n$ return $GCD(n, a-b)$. If $GCD(n, a+b) \neq 1, n$ return $GCD(n, a+b)$. Otherwise, generate more a_i and find a new value of b.

In step 1, we choose values of a_i at random from all residues $\pmod n$ until they meet the listed conditions. The first condition is easy to verify and the second is verified by trial division of each b_i by all p_j until a factorization is found (or all such primes have been tested).

We use the following algorithm in step 2 to pick the set I:

2-a. For each b_i, factor $b_i = \prod_{j=1}^{k-1} p_j^{e_{j,i}}$. Set $\vec{b_i} = [\ e_{1,i} \quad \cdots \quad e_{k-1,i}\ ] \pmod 2$.

2-b. Set

$$M = \left[\ \vec{b_1}^T \quad \cdots \quad \vec{b_k}^T\ \right]$$

2-c. Row reduce M to find a linear combination of $\vec{b_i}$ that sums to 0. Set I to be the set of the b_i with non-zero coefficients in this linear combination.

2.1.5 Proof of Correctness

We first show that the main algorithm will determine a factorization of n with non-zero probability at each step, provided y is chosen large enough that plenty of y-smooth quadratic residues exist. By Theorem 2.4, if n has a small number of factors, say 2 in the common case, then there are at least $\frac{n}{4}$ quadratic residues. Thus, there are at least $\frac{n}{4}$ possible values of b_i, and if we assume they are relatively uniformly distributed amongst the residues $\pmod n$ (a small assumption since there are so many), plenty of these will be y-smooth since we only need $\pi(y)+1$ of them and $y << n$. Then, since the a_i are chosen at random from all possible residues, eventually enough b_i will satisfy the required conditions to look for the subset I (substeps $2-a$ to $2-c$).

To see that steps 2,3 will actually produce an equivalence of squares modulo n, first note that there are $\pi(y)+1$ b_i chosen. Since each b_i is y-smooth, each b_i has a prime factorization using only the first $k-1$ primes. Thus, each vector $\vec{b_i}$ is a $k-1$ dimensional vector, so this implies that the $\vec{b_i}$ must be linearly dependent, i.e. some non-trivial linear combination of them sums to 0. Since we are working with integers modulo 2, the coefficient of each $\vec{b_i}$ in the linear combination will be 0 or 1. Row reduction will of course find such a combination since such a combination exists. It follows that step 2-c will find a linear combination of $\vec{b_i}$ that sums to 0.

To see that we produce a congruence of squares mod n, consider the following: Let

$$\begin{aligned} c &= \prod_{i \in I} b_i \\ &= \prod_{i \in I} \prod_{j=1}^{k-1} p_j^{e_{j,i}} \\ &= \prod_{j=1}^{k-1} \prod_{i \in I} p_j^{e_{j,i}} \\ &= \prod_{j=1}^{k-1} p_j^{\sum_{i \in I} e_{j,i}} \end{aligned}$$

From the row reduction in step 2-c, we have

$$\sum_{i \in I} e_{j,i} \equiv 0 \pmod 2$$

for all j. Thus, the prime factorization of c has only even exponents, i.e. c is a perfect square. c is not 1 since the linear combination was non-trivial. It follows that $b = c^{1/2}$ will produce an integer value of b in step 3.

Finally, from the definition of b, we have

$$\begin{aligned} b^2 &\equiv \prod_{i \in I} b_i \pmod{n} \\ &\equiv \prod_{i \in I} a_i^2 \pmod{n} \\ &\equiv \left(\prod_{i \in I} a_i \right)^2 \pmod{n} \\ &\equiv a^2 \pmod{n} \end{aligned}$$

Therefore, the quadratic sieve will produce a congruence of squares modulo n.

However, a congruence of squares modulo n does not guarantee a useful factorization of n in step 4. In particular, it could just give us the trivial factorization $n = 1 \cdot n$ since we could have for example $GCD(n, a+b) = n$ and $GCD(n, a-b) = 1$. This is certainly possible since the a_i and b_i are related in a very specific way, so it is plausible that we would always have $b \equiv a \pmod{n}$ using this method of generating b. We now consider the problem of solving the congruence

$$a^2 \equiv b^2 \pmod{n}$$

where we think of a as being fixed and we are finding solutions b. If n is the product of two primes, say $n = pq$, then this is equivalent to solving the equations $b^2 \equiv a^2 \pmod{q}$ and $b^2 \equiv a^2 \pmod{p}$. As long as $a \not\equiv 0 \pmod{p, q}$, there will be exactly two solutions to each of these congruences, which by the Chinese Remainder Theorem gives us 4 solutions (combining solutions in all possible ways) gives us 4 solutions $\pmod{n}$. Thus, if b were chosen at random from amongst those solutions, the probability of $b \not\equiv \pm a \pmod{n}$ would be $\frac{1}{2}$. This tells us that the probability of getting a nontrivial factorization is also $\frac{1}{2}$ and hence we can just run our algorithm several times until we are overwhelmingly likely to find a factorization.

While we are unable to give a formal proof that b is randomly distributed amongst the 4 possible solutions $\pmod{n}$, we provide a rough argument as to why that might be the case. The values of a and $a^2 \pmod{n}$ are essentially independent $\pmod{n}$, i.e. $a^2 \pmod{n}$ looks like a randomly chosen quadratic residue $\pmod{n}$, so b, being the product of these randomly distributed quadratic residues will itself be randomly distributed and thus equally likely to any of the roots of $a^2 \equiv b^2 \pmod{n}$.

2.2 Elliptic Curve Factorization

2.2.1 Elliptic Curves Introduction

The problem of factoring the integer N is similar to the problem of understanding the structure of the group given by $\mathbb{Z}/n\mathbb{Z}$ with addition as the group operation. This is because every integer $k \in \mathbb{Z}/n\mathbb{Z}$ has an order that divides n. For most k, that order will be n. But if k is a non-trivial factor of n, then the order of k is neither 1 nor n, so understanding the structure of $\mathbb{Z}/n\mathbb{Z}$ will provide insight into the factorization of n. More generally, we would like to understand $\mathbb{Z}/n\mathbb{Z}$ taken as a ring (with multiplication as the additional operation and same group operation of addition). The Chinese Remainder Theorem tells us the exact relation of $\mathbb{Z}/n\mathbb{Z}$ to rings whose multiplicative groups are cyclic (which are easy to understand) if we know the factorization of n:

Theorem 2.6 (Chinese Remainder Theorem)**.** *Let $n = p_1^{e_1} \cdots p_k^{e_k}$ be a prime factorization of n. Then*

$$\mathbb{Z}/n\mathbb{Z} \cong \mathbb{Z}/p_1^{e_1}\mathbb{Z} \times \cdots \times \mathbb{Z}/p_k^{e_k}\mathbb{Z}$$

In an effort to understand the ring structure of $\mathbb{Z}/n\mathbb{Z}$, we look at the behavior of common mathematical functions taken over it. In particular, it turns out that if we look at what are known as **Elliptic Curves** taken over $\mathbb{Z}/n\mathbb{Z}$, we can learn information about the structure of $\mathbb{Z}/n\mathbb{Z}$, which in turn gives us information about the factorization of n.

Definition 2.7 (Ellliptic Curve)**.** An Elliptic Curve is the set of points (x, y) satisfying a cubic relation of the form

$$ax^3 + bx^2y + cxy^2 + dy^3 + ex^2 + fxy + gy^2 + hx + iy + j = 0$$

Remark 2.8. The points on an elliptic curve must first be defined over a field (i.e. $\mathbb{R}$, $\mathbb{Q}$ or $\mathbb{F}_p$). However, once the curve is defined, we will restrict ourselves to a particular subset of points on that curve (namely rational points) and consider those points in $\mathbb{Z}/n\mathbb{Z}$. The fact that we cannot immediately think of the curve as being over $\mathbb{Z}/n\mathbb{Z}$ due to issues of singularities (with zero-divisors, etc) is what will lead to information about the structure of $\mathbb{Z}/n\mathbb{Z}$ that we want.

2.2.2 Weierstrass Normal Form

The general form for a cubic equation is

$$ax^3 + bx^2y + cxy^2 + dy^3 + ex^2 + fxy + gy^2 + hx + iy + j = 0$$

This form is difficult to work with since it contains so many terms, so instead we would like to focus on cubics of one of the following forms:

$$y^2 = x^3 + ax + b \text{ or } y^2 = x^3 + ax^2 + bx + c \tag{1}$$

The first equation is referred to as *Weierstrass Normal Form* (the second is the generalized form). For the purposes of cryptography and encryption, we will primarily be dealing with rational solutions to these equations. Thus, we first show that any general cubic containing at least one rational point is birationally equivalent to a cubic in Weierstrass Normal Form. In other words, we will show that there exists a rational bijection mapping rational points on a general cubic equation to a corresponding one in Weierstrass Normal Form. With this bijection, we will only have to deal with solving problems dealing with cubics in Weierstrass Normal Form.

Theorem 2.9 (Birational Equivalence Theorem)**.** *Any general cubic equation containing at least one rational point is birationally equivalent to a cubic in Weierstrass Normal Form. In otherwords, for a general elliptic curve $C(x, y)$, there exists a rational bijective mapping $x \mapsto x'$, $y \mapsto y'$ such that $C(x', y')$ is in Weierstrass Normal Form.*

Proof. Let C be the curve corresponding to a general cubic equation over some ring R and O (the "origin") some point on C with rational coefficients. Let L be the tangent line to C at O. Assume that O is not an inflection point so that L intersects C at exactly one other point, P. Let M be the tangent line to C at P, intersecting C again at Q and let N be an arbitrary line (different from L) through O. We can make L, M, and N form the Z, X, and Y axes respectively of a projective plane, letting $O = (1, 0, 0)$ and $P = (0, 1, 0)$ (and the last point of intersection of the 3 lines (0,0,1)). Then think of X, Y, and Z as functions on C. X has a double zero at P and a single zero at Q, Y has a single zero at O and two other zeroes elsewhere, and Z has a double zero at O and a single zero at P. Thus, the function $x = \frac{X}{Z}$ has a double pole at O (and no other poles) and the function $y' = \frac{XY}{Z^2}$ has a triple pole at O (and no other poles). We let $y = \frac{Y}{Z}$ so $y' = xy$. Note that these original transformation of coordinates are rational. It follows that the 7 functions

$$1, x, x^2, x^3, y', y'^2, xy'$$

all have poles of order ≤ 6 at O and no other poles. Applying the Riemann-Roch Theorem, we know that the dimension of the vector space of the span of any such functions is at most 6, so some linear combination of the above functions must be 0, i.e. we have for some a, b, c, d, e, and f

$$y'^2 + axy' + by' = cx^3 + dx^2 + ex + f \tag{2}$$

We substitute $z = y' - \frac{1}{2}(ax + b)$, which is a rational linear transformation and this equation becomes

$$z^2 = cx^3 + dx^2 + ex + f - \frac{1}{4}(ax + b)^2$$

The leading coefficient can again be taken care of by a linear transformation $x \mapsto cx$, $z \mapsto c^2 z$ to get a monic cubic in x and a final standard linear transformation can eliminate any one of the coefficients in x to get to Weierstrass Normal Form. This completes the proof of equivalence to Weierstrass Normal Form in the case that O was not an inflection point; if O was an inflection point, we can choose any line for L and the same holds. Notice also that in order to make the last two simplifications, we had to divide by 2 and 3, so the Weierstrass Normal Form is only guaranteed to hold when we are working over rings in which 2 and 3 are invertible (i.e. $\mathbb{Q}$ or $\mathbb{F}_p$ for $p > 3$, etc).

□

Remark 2.10. The Riemann-Roch Theorem was only needed to show the existence of the form and was used to avoid unnecessarily complicated computations. The computations are quite messy, so in practice, we will make sure to choose cubics that are already in Weierstrass Normal Form whenever we need them for factorization or encryption. This theorem simply tells us that we do do not lose anything by only studying these equations.

2.2.3 Group Law

One of the most useful properties of cubic curves for factorization is the fact that the set of rational points on a cubic curve $\mathcal{C}$ forms a group under the following operation $(+)$:

- Choose an initial rational point $\mathcal{O}$ on $\mathcal{C}$ to be the origin.
- For any two points P and Q on $\mathcal{C}$, define $P * Q$ to be the third point of intersection of the line connecting P and Q and $\mathcal{C}$ (with multiplicity taken into account). This operation is well-defined since every line intersects a cubic in exactly 3 places counting multiplicity.
- Define $P + Q = \mathcal{O} * (P * Q)$.

Notice that from this definition, $\mathcal{O}$ is the identity element for this group since $\mathcal{O} + P = \mathcal{O} * (\mathcal{O} * P)$. $(\mathcal{O} * P)$ is the third intersection of the line containing $\mathcal{O}$ and P with $\mathcal{C}$, so $\mathcal{O} * (\mathcal{O} * P)$ must be P again and hence $\mathcal{O}$ is the identity element. Similarly, we can get inverses. Let the tangent to $\mathcal{C}$ at $\mathcal{O}$ meet $\mathcal{C}$ again at Q. Then, we can use $-P = P * Q$. This definition works since it gives $P * -P = Q$ and hence $P + -P = \mathcal{O} * Q = \mathcal{O}$. Associativity of $+$ also holds; a full proof can be found in Washington's Elliptic Curves: Number Theory and Cryptography.

One detail we glossed over in the previous description of the group law is what happens if the line connecting P and Q is vertical when computing $P * Q$. From the Weierstrass Normal Form of a cubic curve, it is clear that there are at most two solutions y for every point x since $y^2 = x^3 + ax + b$. In this case, we say that $P * Q = \infty$ since the line connecting P and Q will meet $\mathcal{C}$ again at infinity. This turns out to be a very important idea in elliptic curve factorization algorithms, so we go ahead and compute explicit formulas for $P + Q$, taking note of when this computation might lead to an ∞ at some point.

A final point to notice about elliptic curves is that while the group we constructed above depended on the choice of our origin, $\mathcal{O}$, groups constructed with different origins are isomorphic. Say we have a second construction using $\mathcal{O}'$ as the origin. We can easily define an isomorphism from the first construction to the second by $P \mapsto P + (\mathcal{O}' - \mathcal{O})$. Thus, the particular choice of origin is not important when working with cubic curves.

2.2.4 Explicit formulas for P+Q

Let $y^2 = x^3 + ax + b$ be a cubic in Weierstrass Normal Form, $\mathcal{C}$ and say that our cubic is taken over R where R is a ring such that 2 and 3 are both invertible. Let $\mathcal{O} = (x_0, y_0)$ be our origin with rational coordinates. Let $P = (p_x, p_y)$ and $Q = (q_x, q_y)$ be two rational points on $\mathcal{C}$. Then, the slope of the line L connecting P and Q is $\frac{q_y - p_y}{q_x - p_x}$, so L has equation $y = p_y + \left(\frac{q_y - p_y}{q_x - p_x}\right)(x - p_x)$. Plugging this into our equation for $\mathcal{C}$, we have:

$$\left[p_y + \left(\frac{q_y - p_y}{q_x - p_x}\right)(x - p_x)\right]^2 = x^3 + ax + b$$

We know two of the solutions to this equation are $x = p_x$ and $x = q_x$. The sum of the three solutions is negative the coefficient of x^2 when the equation is rearranged into standard polynomial form, so if we let s_x be the x-coordinate of our 3rd point of intersection of L with $\mathcal{C}$, then we have $s_x = -p_x - q_x + \left(\frac{q_y - p_y}{q_x - p_x}\right)^2$. The y-coordinate is of course $s_y = p_y + \left(\frac{q_y - p_y}{q_x - p_x}\right)(s_x - p_x)$. If we let the slope here be m, then we have

$$P * Q = (s_x, s_y) = (-p_x - q_x + m^2,\, p_y + m(s_x - p_x)) \tag{3}$$

To finish, we need to compute $\mathcal{O} * S$. We use our above formula with a slope of $m_s = \frac{s_y - y_0}{s_x - x_0}$ to get

$$P + Q = \mathcal{O} * (P * Q) = (t_x, t_y) = (-s_x - x_0 + m_s^2,\, y_0 + m_s(t_x - x_0)) \tag{4}$$

This formula only works in the case that $P \neq Q$ since we used two points to compute the slope of the line. In the case that $P = Q$, we intead must use the tangent line to $\mathcal{C}$ at P. We have $2y\frac{\partial y}{\partial x} = 3x^2 + a \Rightarrow \frac{\partial y}{\partial x} = \left.\frac{3x^2 + a}{2y}\right|_P = \frac{3p_x^2 + a}{2p_y} = m_p$. Substituting this slope in, our formulas for $P * P$ and $P + P$ are analogous to the ones above:

$$P * P = (s_x, s_y) = (-2p_x + m_p^2,\, p_y + m(s_x - p_x)) \tag{5}$$

$$P + P = (t_x, t_y) = (-s_x - x_0 + m_p^2,\, y_0 + m_p(t_x - x_0)) \tag{6}$$

As a final note, if one of the points is when computing $P * Q$ is ∞, the line connecting P and ∞ is the vertical line through P. If $P = (p_x, p_y)$, then the other point of intersection will be $(p_x, -p_y)$ (taking advantage of Weierstrass Normal Form to do so).

These formulas will provide the basis for our computations in the Elliptic Curve Factorization Algorithm. For notation, we will use $kP = P + P + \cdots + P$, where there are k P's on the right hand side.

2.2.5 Elliptic Curve Factorization Algorithm

The key to Elliptic Curve Factorization is the observation that when we try to compute $P + Q$ for various points P and Q on some elliptic curve $\mathcal{C}$, the only way in which we are unable to perform the calculation is when the slope was undefined in one of our computations in (4) or (6), i.e. when $q_x - p_x$ or $2p_x$ was not invertible in R. This is not particularly interesting in the case of $R = \mathbb{Q}$. However, in the case of $R = \mathbb{Z}/n\mathbb{Z}$ where n is a composite number, this becomes much more interesting. Here, our slope will be undefined whenever $(q_x - p_x, n)$ or $(2p_x, n)$ is not 1. Thus, if we can find places where the slope computation fails, there is also a good chance we could discover a factor of n by computing $(q_x - p_x, n)$ or $(2p_x, n)$. For the elliptic curve factorization algorithm in particular, we compute kP on $\mathcal{C}$ mod n for some n that has at least 2 distinct prime factors so that n can be written as $n = n_1 n_2$ with $(n_1, n_2) = 1$.

We use a lemma from Washington's Elliptic Curves: Number Theory and Cryptography to explain why this idea makes sense. Let $\mathcal{C}(\mathbb{Z}/n\mathbb{Z})$ represent the group of rational points of $\mathcal{C}$ taken over the ring $R = \mathbb{Z}/n\mathbb{Z}$ under the group law described previously. The lemma states that

$$\mathcal{C}(\mathbb{Z}/n_1 n_2\mathbb{Z}) \cong \mathcal{C}(\mathbb{Z}/n_1\mathbb{Z}) \oplus \mathcal{C}(\mathbb{Z}/n_2\mathbb{Z})$$

and is essentially a consequence of the Chinese Remainder Theorem. The proof consists of computations for addition of points and showing that the reduction of these mod $n_1 n_2$ is equivalent to combining the reductions of them mod n_1 and n_2. It is omitted since the equations are long, somewhat messy, and not particularly insightful.

Notice that a computation of kP mod n_1 will result in either some other point on $\mathcal{C}$ or a computation that gives ∞ (if n_1 is prime; otherwise it could also give a point that is neither ∞ nor an actual point on

P). In the elliptic curve factorization algorithm, we look for points such $kP \equiv \infty \pmod{n_1}$ but $kP \not\equiv \infty \pmod{n_2}$. When this is achieved, we have found our factors of n since we can compute n_1 / n_2 by taking the gcd of the denominators of the slope in the computation of kP with n.

It is impractical to compute $kP \pmod{n}$ for all k less than n until we are certain to find a point for which we are unable to compute the slope since n is very large in all practical applications. Thus, we compute $k!P$ only for some value of k, noting that if any of $P, 2P, \cdots kP$ were unable to be computed, then $k!P$ will be unable to be computed as well. Finally, we choose several random elliptic curves in Weierstrass Normal Form, each of which contains a known rational point and run the algorithm on each until we find a factorization. This is summarized below:

1. Choose several random elliptic curves $\mathcal{C}_i$ in Weierstrass Normal Form and a rational point on each, P_i. For example, choose $\mathcal{C}_i : y^2 = x^3 + 3x + 35$ and $P_i = (2, 7)$.
2. Choose some k moderately large and compute $k!P_i$ on $\mathcal{C}_i$ for each i. If any of the steps in the computations of slopes messes up, we have found the factorization of n needed and we are done.
3. Repeat with either new $\mathcal{C}_i$ or larger k until a factorization is found.

2.2.6 Termination of Elliptic Curve Factorization Algorithm

When given this algorithm, it is not initially clear that the algorithm will actually terminate; for example why must some computation of kP eventually lead to an undefined slope? To answer that question, consider $\mathcal{C}$ over some field $\mathbb{F}_p$ for a prime $p > 3$. There are only finitely many points on C (since there are only finitely many values of points (x, y), namely p^2 of them). The infinite sequence of points $P, 2P, 3P, \cdots$ must eventually repeat itself, say $jP = iP$. This then gives $(j - i)P = 0 = \mathcal{O}$. The *order* of P on $\mathcal{C}$ taken over some ring R is the smallest value for which this occurs. As we have shown, when R is finite, the order of P is also guaranteed to be finite. When R is not finite, some points may have finite or infinite order.

Now, we consider the termination of the factorization algorithm. By the decomposition of $\mathcal{C}(\mathbb{Z}/n\mathbb{Z})$ discussed above, we only need to consider the termination of the algorithm on $\mathcal{C}(\mathbb{Z}/p\mathbb{Z})$, where p is some prime. Our initial point P has finite order on this curve, so there exists j for which $jP = \mathcal{O}$. Since the choice of $\mathcal{O}$ was arbitrary (we can just translate between choices of origin), we let $\mathcal{O} = \infty$, which gives us a point, jP, during which the computation of it involves an undefined slope. Hence, after a finite number of steps, we will have found a strong candidate to be a factor of n. In fact, if $n = n_1 n_2$ are two factors of n and the order of P on $\mathcal{C}(\mathbb{Z}/n_1\mathbb{Z})$ is different from the order of P on $\mathcal{C}(\mathbb{Z}/n_2\mathbb{Z})$, we will have found our factor.

In summary, if k is chosen to be large enough during the elliptic curve factorization algorithm, we are guaranteed to be "extremely likely" to have found a prime factor. Using the Weierstrass Normal Form of the equation for an elliptic curve, for each value of x, there are at most 2 values of y such that (x, y) lies on the curve when considered over $\mathbb{F}_p$ since the equation $y^2 \equiv j \pmod{p}$ has at most 2 solutions when p is a prime. Thus, the order of P must in fact be at most $2p$. Since the smallest prime factor of n is $\leq \sqrt{n}$, we can thus say that the runtime is extremely likely to be $O(\sqrt{n})$. This is the run-time of the nieve trial-division algorithm, so it is good to know elliptic curve factorization is expected to be at least as good.

2.3 Polynomial Factorization over Finite Fields

Compared to integer factorization, polynomial factorization in $\mathbb{F}_p$ is quite different. One of the first things to notice about factorization in $\mathbb{F}_p$ is that the nieve algorithm of test dividing every polynomial of smaller degree does not work for polynomials of even moderate degree. To see this, say that we wish to factor some $f(x)$ of degree d. By trial division, we would have to potentially divide by every polynomial in $\mathbb{F}_p$ of degree up to $d/2$. Since there are $p^{d/2+1}$ polynomials in $\mathbb{F}_p$ of degree at most $d/2$, this means that even for relatively small values of d, if p is large, the algorithm will fail. The problem with our trial division approach is that we tried dividing by many more polynomials than we needed to. Ideally, we would just check a select subset of the polynomials (divisible by all the irreducible polynomials) and our algorithm would work much faster

and not have such a horrible dependence on p.

This motivates the factorization algorithm for polynomials over finite fields known as *Berlekamp's Factorization Algorithm*, where we look to find factors by comparison to a much smaller set of polynomials than all of $\mathbb{F}_p$. Given a degree n $f(x) \in \mathbb{F}_p[x]$, we wish to write

$$f(x) = p_1(x)p_2(x)\cdots p_k(x)$$

where $p_i(x)$ is a power of a monic irreducible in $\mathbb{F}_p[x]$. The key idea is to look for polynomials of degree less than n, $g(x)$ satisfying the following condition:

- $f(x) \Big| \prod_{s \in \mathbb{F}_p} [g(x) - s]$

Since each of the $g(x)$ has degree less than n, if we look at $GCD(f(x), g(x) - s)$ for all $s \in \mathbb{F}_p$, then we should find non-trivial factors of f. Since $\mathbb{F}_p[x]$ is a polynomial extension of a field, it is a Euclidean Domain and hence we have the Euclidean Algorithm to efficiently compute GCDs. Once we find a non-trivial factor, we can divide $f$by it and repeat the algorithm on the quotient to determine the remaining factors.

The question of factorization then becomes a matter of determining the $g(x)$. As it turns out, these $g(x)$ form a vector space of dimension k over $\mathbb{F}_p$; call this vector space V. V is called a Berlekamp subalgebra. We wish to determine a basis of this vector space. The following Lemma explains how to do so:

Lemma 2.11. *Berlekamp Basis Lemma*
For $j = 0, 1, \cdots, n-1$, let

$$x^{pj} \equiv a_{0,j} + a_{1,j}x + \cdots + a_{n-1,j}x^{n-1} \pmod{f(x)}$$

Let

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}$$

Finally, let $g(x) = b_0 + b_1x + \cdots + b_{n-1}x^{n-1}$. Then the condition that $g(x) \in V$ is equivalent to

$$(A - I)B = 0 \tag{7}$$

Proof. Notice that multiplication of B by A is equivalent to computing $g(x^p) \mod f(x)$ since the rows of A give the reductions of x^{jp} and we can write

$$g(x^p) = \sum_{z=0}^{n-1} b_z x^{zp} \tag{8}$$

$$\equiv \sum_{z=0}^{n-1} b_z \sum_{y=0}^{n-1} a_{y,z}x^y \pmod{f(x)} \tag{9}$$

$$\equiv \sum_{z=0}^{n-1} x^z \sum_{y=0}^{n-1} b_y a_{z,y} \pmod{f(x)} \tag{10}$$

Thus, the condition $(A - I)B = 0$ is equivalent to the condition $g(x^p) \equiv g(x) \pmod{f(x)}$.

□

Since the dimension of V is k, the rank of $A - I$ is $n - k$. We then compute a basis $\{g_1(x), \cdots g_k(x)\}$ of V by solving (7). This can be done by row-reducing $A - I$.

A factorization of a polynomial $f(x)$ in $\mathbb{F}_p[x]$ can thus be determined in the following steps:

1. Compute $x^{jp} \pmod{f(x)}$ for $j = 0, 1, \cdots, n-1$. For $j = 1$, this is done by polynomial division of $f(x)$ by x^p. For $j > 1$, note that $x^{jp} = x^{(j-1)p}x^p$, so we can just multiply the result for $x^{(j-1)p}$ by x^p and reduce modulo $f(x)$.

2. Row reduce $A-I$ to solve $(A-I)B = 0$ and let a basis for the result be the coefficients of $g_1(x), \cdots g_k(x)$.

3. For $i = 1, 2, \cdots k$ and $s \in \mathbb{F}_p$, compute $GCD(f(x), g_i(x) - s)$. One of these computations will give a non-trivial factor of f.

Remark 2.12. If step 3 above is modified so that we divide $f(x)$ by any factors we find before looking for more GCDs, it will in fact produce a complete factorization of $f(x)$ into powers of irreducibles.

3 Theoretical Runtime Analysis

3.1 General Background

3.1.1 Prime Number Theorem

In analyzing the performance of our factorization algorithms, one important result we need to know is how prime numbers are distributed. Fortunately, we have one of the most important results in Number Theory, the Prime Number Theorem, the proof of which is far too complex to include in this paper. Its statement is as follows:

Theorem 3.1. *Prime Number Theorem Let $\pi(x)$ be the number of primes less than or equal to x. Then for large x, we have*

$$\pi(x) \sim \frac{x}{\log x}$$

3.1.2 Smooth Numbers

Before we give a better run-time analysis of the algorithms, we provide a little theory about k-smooth numbers. Define a smooth-number counting function $\Psi(x, y)$ as follows:

$$\Psi(x, y) = |\{n \leq x : p|n \Rightarrow p \leq y\}| \tag{11}$$

$\Psi(x, y)$ counts the number of numbers less than x that are y-smooth. We define u such that:

$$y = x^{1/u} \tag{12}$$

Note that we can equivalently define

$$u = \frac{\log x}{\log y} \tag{13}$$

We have one asymptotic formula regarding Ψ, namely

Theorem 3.2. $\Psi(x, y) = \rho(u)x + O\left(\frac{x}{\log y}\right)$, *where $\rho(u)$ is the Dickman Function defined by $\rho(u) = 1$ if $0 \leq u \leq 1$ and the relation $u\rho'(u) = -\rho(u-1)$ if $u \geq 1$.*

In general, $\rho(u)$ is difficult to evaluate precisely, although for the case $1 \leq u \leq 2$, we have $\rho(u) = 1 - \log u$. In particular, this gives us $\Psi(x, \sqrt{x}) \approx (1 - \log 2)x$ as an example that's easy to think about. However, notice that the error term becomes too large when approximately when $y < x^{1/\log\log x}$, so we must ensure y is reasonably large before using. For larger values of u, we fortunately have another theorem to bound $\rho(u)$:

Theorem 3.3. $\rho(u) = u^{-u+o(1)}$

Combining these two theorems gives us the result we will make use of in our runtime analysis:

Corollary 3.4.

$$\frac{\Psi(x, x^{1/u})}{x} \sim u^{-u} \tag{14}$$

3.2 Quadratic Sieve Runtime

3.2.1 Selection of b_i

The runtime of the quadratic sieve factorization algorithm is typically dominated by the step where a_i's are chosen. Let n be the number to be factored and let y be the smoothness factor of the quadratic sieve and let $y = n^{1/u}$. Each a_i is chosen uniformly at random mod n, so we assume that a_i^2 is also uniformly at random mod n. By INSERT COROLLARY REFERENCE, the probability of $b_i = a_i^2 \pmod{n}$ being y-smooth is thus u^{-u}. Thus, the expected number of a_i that must be chosen to get $\pi(y)+1$ values of b_i that are y-smooth is

$$(\pi(n^{1/u}) + 1)u^u$$

Notice that far more residues less than n are not smooth than smooth, so most iterations of the loop will be spent verifying that a residue b_i is not y-smooth. The only way to verify this is to check divisibility of b_i by all primes less than y, which will take $\pi(y) = \pi(n^{1/u})$ checks (notice that we can verify the y-smoothness of a number in at most this many divisions). Thus, producing $\pi(y) + 1$ values of b_i that are y-smooth will take time approximately

$$\pi(n^{1/u})^2 u^u$$

From the prime number theorem, we have $\pi(n^{1/u}) \sim \frac{n^{1/u}}{\log n^{1/u}} = \frac{un^{1/u}}{\log n}$. Plugging this into the above formula gives us that the asymptotic runtime to choose b_is is

$$\frac{u^{u+2}n^{2/u}}{\log^2 n} \tag{15}$$

3.2.2 Optimization

Clearly the runtime in (15) depends on the choice of u with respect to n. We will give an approximate minimum of this expression, which will serve two purposes: to give a good choice for u (and hence y) when implementing the Quadratic Sieve and also to give an upperbound on its runtime.

For the purposes of the algorithm, n is regarded as a constant, so the $\log^2 n$ term will not impact where a local minimum occurs. This means we can instead minimize the function $u^{u+2}n^{2/u}$. Taking a derivative with respect to u of our simplified expression and solving for where it is 0, we have:

$$\begin{aligned} \frac{\partial}{\partial u}\left(u^{u+2}n^{2/u}\right) &= 0 \\ u^u n^{2/u}(u(u + u\log u + 2) + 2\log n) &= 0 \\ u(u + u\log u + 2) &= 2\log n \end{aligned} \tag{16}$$

For a purely asymptotic approximation of the solution to this, we first notice that $u = \sqrt{\log n}$ is an approximate solution, which tells us that $\log u \approx \log\log n$. Plugging this back in, we see that the solution is approximated by the solution to

$$u^2(1 + \log\log n) = 2\log n$$

An approximate solution to this equation is of course $u = \sqrt{\frac{2\log n}{1+\log\log n}}$. Plugging this back in, the asymptotic runtime is bounded above by: INSERT SIMPLIFICATION OF RUNTIME HERE

For most numbers that we would reasonably want to (and expect to be able to) factor today, we have $\log n \leq 400$. For $\log n = 400$, using a graphing calculator we see that an approximate solution is given by $u = 15$. Below, we have a table of approximate values of u (as well as $y = n^{1/u}$) for common sizes of n we might wish to factor, as well as the approximate number of divisions required to generate the b_i (representing the runtime of the algorithm):

$\log n$	u	y	**Num Divisions**
20	3.7	210	$214,000$
30	4.5	770	$12,000,000$
40	5.1	2400	$428,000,000$
50	5.7	6600	$11,000,000,000$
400	14.5	9.6×10^{11}	8.3×10^{37}

3.3 Elliptic Curve Factorization Runtime

Under a few key assumptions, we can give an explicit expected runtime analysis. The first step is to notice that the computation of $k!P$ can be done in $\theta(\log k)$ time (keeping in mind that we would like to compute P, $2P$, $3P$, $\cdots$ in order so we can "short-circuit" the computation if the order of our chosen point happens to be small). This can be done by noting that we can compute xP in $\log_2 x$ time for any point P and any x by computing P, $2P$, $4P$, $\cdots$ and then adding up the ones in the binary expansion of x to get xP. We compute $2^y P$ from $2^{y-1}P$ in constant time since it takes constant time to calculate $2^{y-1}P + 2^{y-1}P = 2^y P$. Here, we have taken advantage of the group structure of rational points on elliptic curves to perform the computation more efficiently. Thus, when we compute $k!P$ as $k(k-1)(k-2)\cdots P$, the run time will be $\theta(\log k + \log(k-1) + \cdots) = \theta(\log k!) = \theta(k \log k)$.

Next, we notice that for our factorization algorithm to be successful at finding some factor p of n (where p is a prime), we must have the order of the chosen point P over $\mathbb{F}_p$, $\text{ord}_p P$ to be a divisor of $k!$. As an upper bound on the runtime, this will be very very likely to be true if $\text{ord}_p P$ is $k-$smooth (in fact, it wil hold with probability at least $1 - 1/k$).

Our three assumptions (two of which are unproven) are as follows:

1. **Hasse's Theorem on elliptic curves:** *The order of any point P on an elliptic curve over $\mathbb{F}_p$ is in the range $p - 2\sqrt{p} < ord_p P < p + 2\sqrt{p}$*
2. (unproven) *$ord_p P$ is approximately uniformly distributed around p in accordance with the range in assumption 1.*
3. (unproven) *The approximation for $\frac{\Psi(x,x^{1/u})}{x}$ given in* (14) *will also hold over the range $(x - 2\sqrt{x}, x + 2\sqrt{x})$.*

The first assumption lets us say that $\text{ord}_p P \approx p$ and the second two assumptions lets us say that the probability of $\text{ord}_p P$ being $p^{1/u}$ smooth is u^{-u}. In our previous notation involving k, let

$$k = p^{1/u} \Leftrightarrow u = \frac{\log p}{\log k}$$

What this tells us is that the probability of our algorithm finding a factor p is u^{-u}, so the expected number of iterations of the algorithm to find a factor of n is u^u. In the typical case, the number to be factorized, n will have only two factors. It suffices to find the smaller factor, which is of size at most $\sqrt{n}$. Hence, our general runtime of the algorithm can be expressed as:

$$O\left(u^u (k \log k)^{\log_p k}\right) = O\left(u^u \sqrt{n}^{1/u} \log \sqrt{n}^{1/u}\right) = O\left(u^{u-1} n^{1/2u} \log n\right) \tag{17}$$

As expected, the runtime depends on the precise choice of k (and thus u) relative to n, so we now consider the problem of optimizing this upper bound by choice of u. Maximizing the above function explicitly is unnecessarily tedious, so we instead come up with an approximation to the optimized solution, which in practice gives a good enough bound on the runtime. The $\log n$ won't affect the optimum value of u, so we focus on minimizing the quantity $u^{u-1} n^{1/2u}$, which will happen at roughly the same place as the minimum of the function $u^u n^{1/2u}$ since u is small. We minimize with respect to u as follows:

$$\frac{\partial}{\partial u}\left(u^u n^{1/2u}\right) = 0 \tag{18}$$

$$u^u(\log u + 1)n^{1/2u} - u^u n^{1/2u}\log n/2u^2 = 0 \tag{19}$$

$$(\log u + 1) - \frac{\log n}{2u^2} = 0 \tag{20}$$

$$2u^2(\log u + 1) = \log n \tag{21}$$

$$u \approx \sqrt{0.5\log n} \tag{22}$$

This gives us our first-order approximation ignoring the $\log u + 1$ term. We now approximate

$$\log u \approx \log\left(\sqrt{0.5\log n}\right) \approx \log\log n$$

Note that the facor of $\log\log n$ is relevant in practice since typical values of n to be factored have $\log n > 50$. Using this, we get a more precise bound approximation for u; our previous equation becomes

$$2u^2\log\log n \approx \log n \tag{23}$$

$$u \approx \sqrt{\frac{\log n}{2\log\log n}} \tag{24}$$

This gives us a good upper bound on the runtime. While this bound is not polylog, it is clearly sub-exponential in $\log n$, which is a good starting place for a factorization algorithm. Once we plug in u, our expression will have both $\log n$ and n raised to complex powers. To combine them, we plug in $n = (\log n)^{\log n/\log\log n}$. Then, our runtime bound is given by:

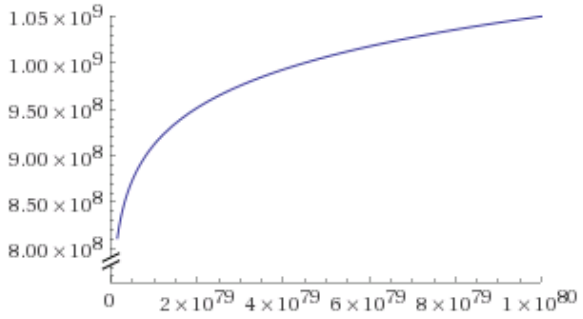
$$O\left(\left[\sqrt{\frac{\log n}{2\log\log n}}\right]^{\sqrt{\frac{\log n}{2\log\log n}}-1}(\log n)^{\frac{\log n}{\log\log n}1/2\sqrt{\frac{\log n}{2\log\log n}}}\log n\right) \tag{25}$$

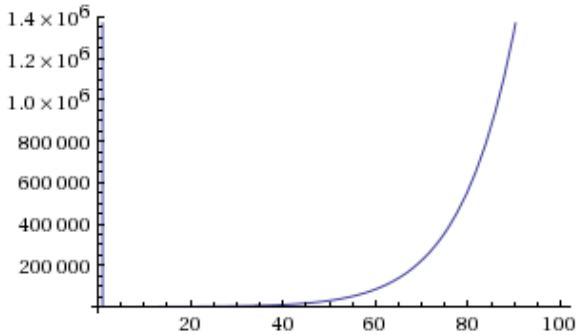
$$=O\left((\log n)^{\sqrt{\frac{\log n}{2\log\log n}}+\frac{1}{2}}(2\log\log n)^{-\frac{1}{2}\sqrt{\frac{\log n}{2\log\log n}}-\frac{1}{2}}\right) \tag{26}$$

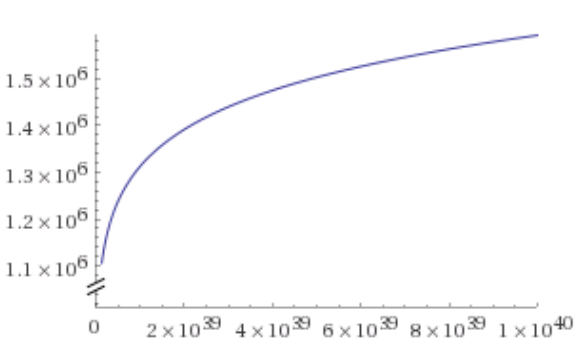
If we let $d = \log n$, the expression cleans up a bit and we can see the sub-exponential nature of it:

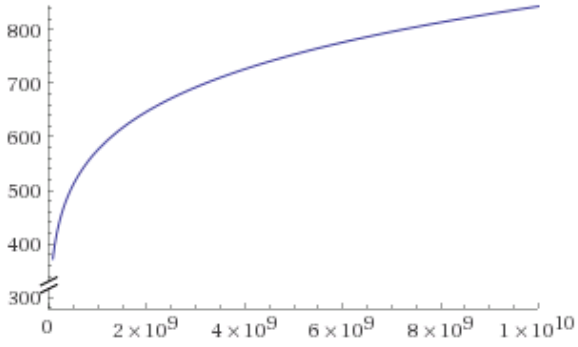
$$O\left(d^{\sqrt{\frac{d}{2\log d}}+\frac{1}{2}}(2\log d)^{-\frac{1}{2}\sqrt{\frac{d}{2\log d}}-\frac{1}{2}}\right)$$

A few plots showing the theoretical runtime of the algorithm are shown below. Note that the second plot is plotted as a function of $\log n$ rather than n.






1.5×10^6
1.4×10^6
1.3×10^6
1.2×10^6
1.1×10^6
0
2×10^39
4×10^39
6×10^39
8×10^39
1×10^40


800
700
600
500
400
300
0
2×10^9
4×10^9
6×10^9
8×10^9
1×10^10

4 Practical Runtime Analysis

4.1 Baseline

In all of the algorithms presented so far, we have only been able to provide asymptotic runtimes. While these runtimes are clearly superior to the nieve factorization algorithm for extremely large n, they may not become better in practice for any n for which the factorization can be realistically computed. For example, in Elliptic Curve factorization, each step requires computations of point additions on elliptic curves. As (4) shows, each of these calculations requires many smaller calculations, so the constants involved in the asymptotic formulas given may be quite large. Furthermore, our asymptotic bound for smooth numbers given in (14) may not converge quickly either, making the practical runtime difficult to analyze. As a solution, we provide basic non-optimized python scripts to compare to the nieve algorithm. Code for these scripts can be found in the appendices.

As a baseline, we compare our algorithms to the nieve trial-division algorithm whose python code is given below:

```
def nieve_factor(n):
    if n%2 == 0:
        return 2
    tester = 3
    while tester*tester <= n:
        if n%tester == 0:
            return tester
        tester+=2
    return n
```

Running this algorithm on select values of n gives the following benchmark values (∞ as runtime means the test was not run due to it taking too long):

n	log n	Process Time (s)	Num Divisions
5105857×5104933	30.9	1.25	2.55×10^5
15485653×15485059	33.0	3.68	7.74×10^6
49979183×49979653	35.5	11.8	2.50×10^7
85923853×86007937	36.5	20.2	4.30×10^7
$373058869 \times 373586009$	39.5	87.6	1.86×10^8
$961748941 \times 982451653$	41.4	231	4.81×10^8
$9999999929 \times 10000000033$	46.0		5.0×10^9

4.2 Quadratic Sieve

We now give the same analysis for our own python implementation of the Quadratic Sieve. Our implementation is far from optimized for speed, but is fully functional. One particular draw-back of our algorithm is that due to time restraints when implementing it, it is only able to produce a single congruence of squares modulo n (the number to be factored). This by itself does not guarantee a factorization since it may be a trivial congruence. However, generating another congruence given the initial sieving data is easy in theory since it can be guaranteed by producing just a single new smooth square residue. Since the probability of finding a non-trivial congruence is large, it only takes a small amount of additional time to find a factorization with extremely high probability. Thus, we report the runtimes required to generate $\pi(y) + 1$ smooth square residues instead. The full code is listed in an appendix. Note that to avoid over-use of slow randomization algorithms, we generated residues uniformly mod n and then counted forward until we found one whose square was smooth.

n	log n	CPU Time (s)	Actual Num Divs	Expected Num Divs	y
5105857×5104933	30.9	6.69	1.4×10^7	1.7×10^7	770
15485653×15485059	33.0	12.6	3.0×10^7	3.8×10^7	900
49979183×49979653	35.5	37.2	8.6×10^7	8.9×10^7	1600
85923853×86007937	36.5	47.2	1.2×10^8	1.3×10^8	1800
$373058869 \times 373586009$	39.5	109	3.3×10^8	3.6×10^8	2300
$961748941 \times 982451653$	41.4	209	6.5×10^8	6.9×10^8	2650
$9999999929 \times 10000000033$	46.0	967	3.0×10^9	3.2×10^9	4500
$71998888873 \times 72005833099$	50.0	3670	1.1×10^{10}	1.1×10^{10}	6600

Remark 4.1. Since the practical runtimes match the expected runtimes quite closely (in terms of number of divisions), quadratic residues and smooth numbers modulo n are likely close to independent (for large n). If they were related, we would expect to see a difference in the runtimes since independence was a key part of our theoretical runtime analysis.

4.2.1 Optimized Quadratic Sieve

Comparing the results of this quadratic sieve to the baseline, we see that even for 21-digit numbers, the quadratic sieve is not expected to be tremendously faster, nor is it tremendously faster in practice. Of course, once we get to much much larger numbers, like 40-digit numbers, the Quadratic Sieve as we have implemented it should be many orders of magnitude faster than naive trial division. However, our observation still leads us to consider improvements to the original algorithm.

The slowest part of the algorithm is of course finding smooth residues (i.e. choosing the b_i). As we mentioned in our earlier runtime analysis, if a_i is chosen uniformly at random in the interval $(0, n)$, then b_i will be roughly uniform in $(0, n)$ as well. Thus, to improve performance instead of picking residues entirely at random modulo n, we try picking residues $a \approx \sqrt{n}$ so that $a^2 - n$ is small and thus more likely to be smooth. In particular, if y is the smoothness parameter (note that $y << n$), we choose residues uniformly as follows:

$$a_i \sim \text{Uni}\left(\lceil \sqrt{n} \rceil, \lceil \sqrt{n} \rceil + \pi(y)^2\right)$$

We use the birthday paradox to give a first rough estimate of the width of the interval. Since we are choosing approximately $\pi(y)$ numbers from this interval, we don't want the residues we pick to collide too frequently. This happens around the square of the number of items chosen, so we set the width of the interval to be $\pi(y)^2$. Since a_i is chosen from this distribution, we will have

$$b_i = a_i^2 - n \approx \pi(y)\sqrt{n}$$

Or, up to good approximation since $\pi(y) << n$, we'll have

$$0 < b_i < 2\pi(y)\sqrt{n}$$

Notice that since b_i is of a very particular form, i.e. $a_i^2 - n$, b_i will not be distributed uniformly on the interval $(0, \pi(y)\sqrt{n})$. In particular, the terms will be quadratically distributed since a_i is still chosen uniformly. Thus, we no longer have strong reason to suspect that b_i is as likely as any other number in $(0, 2\pi\sqrt{n})$ to be y-smooth. However, if this were still the case, we could still estimate the probability of b_i being y-smooth using Corollary 3.4 as follows:

$$\text{P}(b_i \text{ is } y\text{-smooth}) = \frac{\Psi(2\pi(y)^2\sqrt{n}, y)}{2\pi(y)^2\sqrt{n}} = u_1^{-u_1}$$

Here, $u_1 = \frac{\log 2\pi(y)^2\sqrt{n}}{\log y} \approx \frac{2\log \pi(y) + \frac{1}{2}\log n}{\log y}$. Even assuming our old choices of the smoothness factor Compare this with our value of u when a_i was chosen from all residues,

4.2.2 Optimized Quadratic Sieve Practical Results

Running our algorithm in practice, however, we discovered that while congruences are found much quicker using this algorithm, the congruences that are found are far more likely to yield the trivial factorization. We used the following procedure to test this:

1. Generate a congruence of squares modulo n with all a_i randomly generated.
2. Test if congruence gives non-trivial factorization.
3. Repeat steps 1 and 2 until a non-trival factorization is found, recording how many iterations are needed.

We fully randomized the a_i for each congruence trial to avoid any dependence that might result from reusing some of the same a_i. Our results are shown in the table below. 'Trials' is the average number of congruences generated to get a non-trivial factorization. 'Divs' is the average number of divisions required to compute a congruence. We ran the experiment until we found 20 non-trivial factorizations of each n:

n	**y**	**Trials**	**Divs**
5105857×5104933	770	23.65	3.6×10^6

Unfortunately, for larger values of n, the program was not able to find 20 non-trivial factorizations after over 12 hours of searching (despite producing many distinct congruences of squares mod n). Thus, we conclude that this optimization on its own does not provide a useful way to speed up the algorithm. In particular, the smooth residues it produces seem to always multiply to provide a trivial factorization.

4.3 Elliptic Curve Factorization

5 Conclusion

A Quadratic Sieve Python Code

```
import random, math, fractions
import time
from gmpy2 import isqrt

def test_prime_small(n):
    # returns whether or not the given small integer is prime
    if n%2 == 0 and n>2: return False
    factor = 3
    while factor*factor <= n:
        if n%factor == 0:
            return False
        factor+=2
    return True

def generate_small_primes(n):
    # Returns a list of all prime numbers less than n
    small_primes = []
    for i in range(2,n):
        if test_prime_small(i):
            small_primes.append(i)
    return small_primes

def small_factor(n, primes):
    # Returns a factorization of n in the form of a vector of exponents
    # assuming all prime factors are in the given list of primes.  Returns
    # None if such a factorization does not exist
    exponentvector = []
    for p in primes:
        counter = 0
        while n%p == 0:
            n = n / p
            counter+=1
        exponentvector.append(counter%2)
    if n==1:
        return exponentvector
    else:
        return None

def generatesquares(n, smallprimes):
    # Returns a list of squares such that their reduced residues are
    # divisible only by primes in smallprimes.  Returns 1 more residue
    # than the number of primes given
    avalues = []
    bvalues = []
    bfactors = []
    # TODO:  check for duplicates
    while len(avalues) < len(smallprimes) + 1:
        a = random.randint(1,n-1)
        if a*a < n:
            continue
        if a in avalues:
            continue
```

```
        b = (a*a)%n
        if b in bvalues:
            continue
        if int(math.sqrt(b) + 0.5)**2 == b:
            # b is a perfect square, so move on
            continue
        factors = small_factor(b, smallprimes)
        if factors != None:
            avalues.append(a)
            bvalues.append(b)
            bfactors.append(factors)
            print(str(len(avalues)) + " out of " + str(len(smallprimes) + 1))
    return (avalues, bvalues, bfactors)

def vector01_add(v1, v2):
    v3 = []
    for i in range(len(v1)):
        v3.append( (v1[i]+v2[i])%2)
    return v3

def getbsquare(bfactors):
    # Determines which linear combination of the factored bs will
    # multiply to give a perfect square

    for i in range(len(bfactors[0])):
        # Find non-zero element in column i
        nonzeroindex = i
        while nonzeroindex < len(bfactors[i]) and bfactors[i][nonzeroindex] == 0:
            nonzeroindex+=1
        if (nonzeroindex == len(bfactors[i])):
            print("No non-zero entry for column " + str(i))
            continue
        # Swap row at non-zero index with top row
        for j in range(len(bfactors[i])):
            temp = bfactors[j][i]
            bfactors[j][i] = bfactors[j][nonzeroindex]
            bfactors[j][nonzeroindex] = temp

        # Subtract non-zero row from any rows with leading 1 (mod 2)
        for j in range(i+1, len(bfactors[i])):
            if bfactors[i][j] == 1:
                for k in range(len(bfactors)):
                    bfactors[k][j] = (bfactors[k][j] + bfactors[k][i])%2

    # Generate a non-trivial solution from reduced matrix
    solution_array = [0]*len(bfactors)
    # Find last non-zero entry on the main diagonal
    # maxindex = len(bfactors)-2
    maxindex = 0
    while bfactors[maxindex][maxindex] == 1:
        maxindex+=1
    maxindex-=1
    # TODO: make sure there are two indices here and loop if not
    # Find a second non-zero index in this row
```

```
    secondindex = -1
    while secondindex == -1:
        if (bfactors[maxindex][maxindex] == 0):
            maxindex-=1
            continue
        for i in range(maxindex+1, len(bfactors)):
            if bfactors[i][maxindex] == 1:
                # make sure all entries below in this column are 0
                foundone = False
                for j in range(maxindex+1, len(bfactors[0])):
                    if bfactors[i][j] == 1:
                        foundone = True
                        break;
                if not foundone:
                    secondindex = i
                    break
        if secondindex != -1:
            break
        maxindex-=1

    solution_array[maxindex] = 1
    solution_array[secondindex] = 1

    index = maxindex - 1
    for i in range(maxindex):
        value = 0
        if bfactors[index][index] == 0:
            index-=1
            continue
        for j in range(index+1, len(bfactors)):
            value = (value + solution_array[j]*bfactors[j][index])%2
        solution_array[index] = value
        index-=1

    print(solution_array)
    return solution_array

def factor(n):
    print("factoring " + str(n))
    # Factors any integer that is the product of two primes
    # we will require all bi to be smoothness-smooth
    smoothness = 1000
    smallprimes = generate_small_primes(smoothness)
    squares = generatesquares(n, smallprimes)
    avalues  = squares[0]
    bvalues = squares[1]
    bfactors = squares[2]

    # Find a linear combination of bfactor vectors that is 0
    bset = getbsquare(bfactors)
    bsquare = 1
    a = 1
    for i in range(len(bset)):
        if bset[i]:
```

```
            bsquare *= bvalues[i]
            a *= avalues[i]

    intsquare = isqrt(bsquare)
    f = fractions.gcd(intsquare - a, n)
    if f == n:
        return 1
    return f

#factor(541*7907)
#factor(5105857*5104933)
f = 1
t0 = time.clock()
t1 = time.time()
while(f == 1):
    f = factor(15485653*15485059)
    #f = factor(961748941*982451653)
print(f)
print(time.clock() - t0, "seconds process time")
print(time.time() - t1, "seconds wall time")
```

B Elliptic Curve Python Code

Future Work

I plan to work on improving the following points:

- Provide more precise analysis of runtime of the algorithms presented and compare them, paying particular attention to runtimes in practical applications.
- Research about the General Number Field Sieve and compare its performance as well.
- Research elliptic curve applications to primality testing. Also look into AKS test for primality.
- Make precise statements about how likely the algorithms are to "fail" (i.e. have $a^2 \equiv b^2 \pmod{n}$ not yield a factorization, etc).

Sources

1. Silverman and Tate: Rational Points on Elliptic Curves
2. Lawrence C. Washington: Elliptic Curves: Number Theory and Cryptography
3. Bruno Joyal: http://math.stackexchange.com/questions/489671/explicit-derivation-of-weierstrass-normal-form-for-cubic-curve (used in proof of existence of Weierstrass Normal Form)
4. Kannan Soundararajan: Math 155 Lecture (used for Quadratic Sieve)
5. Quadratic Sieve Wikipedia Article: http://en.wikipedia.org/wiki/Quadratic_sieve
6. `https://primes.utm.edu/lists/small/millions/`
7. Wolfram Elliptic Curve article
8. Dummit and Foote