Algorithms for Factoring Integers of the Form n = p * q

Robert Johnson

Math 414 Final Project

12 March 2010

Table of Contents

Introduction

This paper is a brief overview of four methods used to factor integers. The premise is that the integers are of the form n = p * q, i.e. similar to RSA numbers. The four methods presented are Trial Division, the Pollard (p -1) algorithm, the Lenstra Elliptic Curve Method, and the Continued Fraction Factorization Method. In each of the sections, we delve into how the algorithm came about, some background on the machinery that is implemented in the algorithm, the algorithm itself, as well as a few examples. This paper assumes the reader has a little experience with group theory and ring theory and an understanding of algebraic functions. This paper also assumes the reader has experience with programming so that they might be able to read and understand the code for the algorithms. The math introduced in this paper is oriented in such a way that people with a little experience in number theory may still be able to read and understand the algorithms and their implementation. The goal of this paper is to have those readers with a little experience in number theory be able to understand and use the algorithms presented in this paper after they are finished reading.

Trial Division Algorithm

Trial division is the simplest of all the factoring algorithms as well as the easiest to understand. The algorithm is usually used on machines with little available memory as well as on "small" numbers.

The idea is to check every number up to the square root of n, the number to be factored. We need only check up to the square root of n since if p | n, then n = p * q. If q is less than p, trial division would have picked up q and n would have been shown to be divisible by q or factors of q. Eliminating all numbers by checking if they aren't prime speeds up the algorithm. Many algorithms implemented today may or may not find factors of a given number, i.e. the algorithms might fail or only split the number. However, because of the nature of trial division, if a number is outputted, then it is a factor. Also, trial division is the 'best' primality test in that if no numbers are outputted, then the number is prime.

The algorithm for finding all the primes up to a given number m is to use the prime number sieve. We shall use this sieve to create a list of prime number to use in the trial division algorithm. The algorithm is as such:

Algorithm (Prime Number Sieve)[1]: Find all primes up to a given number m:

[1] Stein,William. *Elementary Number Theory: Primes, Congruences, and Secrets.* S. Axler. New York: Springer, 2000. (12)

1. Let X = [3, 5, . . .] be the list of all odd integers between 3 and n. Let P = [2] be the list of primes found so far.
2. Let p be the first element of X. If $p \geq \sqrt{n}$, append each element of X to P and terminate. Otherwise append p to P.
3. Set X equal to the sublist of elements in X that are not divisible by p. Go to Step 2

To run the trial division algorithm, after implementing the prime number sieve, now requires checking fewer elements and only checking elements that are prime instead of composite. Below is code, implemented in SAGE for trial division:

```
#Uses trial division to find factors of n
def trial_division(n):
  for i in range(2, sqrt(n) + 1):
    if is_prime(i):
      a = n/i
      if a == a.floor(): #Check to see if a is an integer
        return (n/a, a)
  return "Prime"
```

Some examples of trial division are given below, implemented in SAGE:

Example 1-

```
SAGE: time print trial_division(35)
SAGE (Output): (5, 7)
SAGE (Output): Time: CPU 0.00 s, Wall: 0.00 s
```

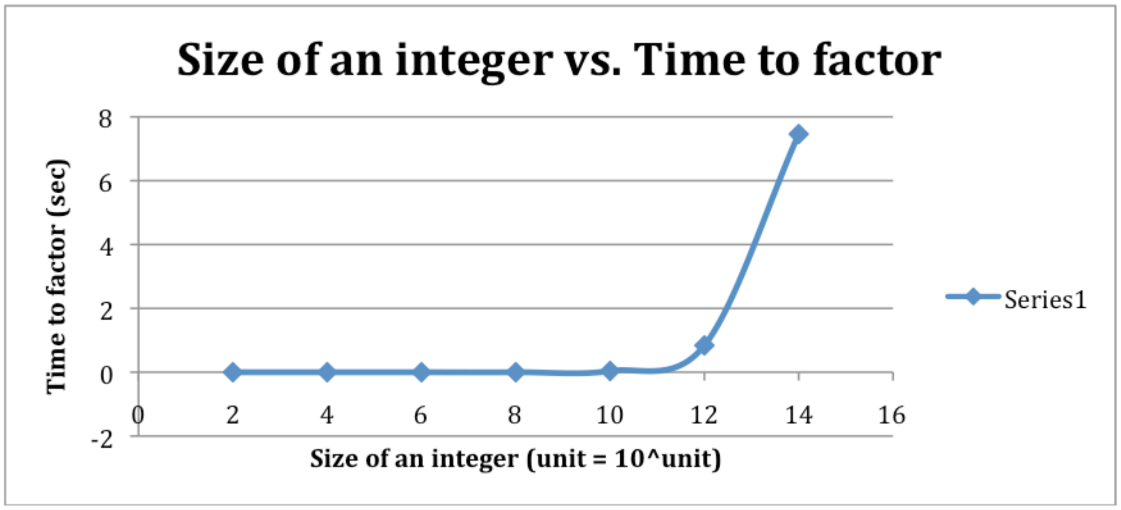
Example 2 –

```
SAGE: p = next_prime(ZZ.random_element(10^6))
SAGE: q = next_prime(ZZ.random_element(10^6))
SAGE: n = p * q
SAGE: time print trial_division(n)
SAGE (Output): (699319, 927287)
SAGE (Output): Time: CPU 5.14 s, Wall: 5.14 s
```

Example 3 (Primality Test) –

```
SAGE: p = next_prime(ZZ.random_element(10^6))
SAGE: time print trial_division(p)
SAGE (Output): Prime
SAGE (Output): Time: CPU 0.01 s, Wall: 0.01 s
```

After seeing these three examples, you might say, "Wow, that doesn't seem like that long to wait to find a factor of n. Why not just use trial division on larger numbers?" The answer to this question lies in the fact that your computer can divide small numbers quickly, but division on large numbers is expensive. Thus, the larger your number, the longer and longer it will take to complete a divison. Try running example 2 with random numbers on the order of 10^8 or larger, you will run into a time problem. The time to factor the integer increases exponentially. Below is a graph showing the average time to factor a random integer of different sizes.



Thus, trial division is an OK method for factorization when the number you want to factor is smaller than 10^{13}, or that the factors are smaller than 10^7. Thus, we can factor numbers with 12 digits[2]. After that, you will need more powerful computers running in parallel to complete the computations. But, RSA uses numbers with anywhere from 300 digits and up. So, trial division fails very short of being able to factor any of the numbers used in RSA or most security systems.

Pollard p-1 Algorithm

The Pollard (p -1) algorithm was created by John Pollard in 1974 as a special-purpose algorithm for factoring a composite number n. By special-purpose algorithm, we mean an algorithm that works best for integers whose factors have a special property. The property we are concerned with in this section is that at least one of the factors, p, of the integer must be such that (p – 1) has only small factors. We shall define this property as:

Def: (B- power Smooth)[3] – Let $B \in \mathbf{N}$, a number n is said to be B-power smooth if $p^e|n$, then $p^e \leq B$. (i.e. $n \mid m = \text{lcm}(1, 2, \ldots, B)$)

[2] Stinson,Douglas. *Cryptography: Theory and Practice.* Second ed. New York: Chapman & Hall/CRC, 2002. (182)
[3] Stein (129)

The natural question to ask is what is the motivating theorem behind this algorithm? Why are we interested in (p – 1)? Recall from group theory that the order of an element, a, in the group (G, *), is the smallest positive integer k such that $a^k = 1$, where $a^k = a * a * \ldots * a$ (k times). With this fact, we introduce

Theorem (Fermat's Little Theorem) – If p is a prime, then $a^{p-1} \equiv 1 \pmod p$, $\forall$ a coprime to p.

Proof: Let G = {1, 2, …, p – 1} with the binary operation of multiplication
Thus $G = (\mathbf{Z}/p\mathbf{Z})^*$, a group
The fact that (G, *) is a group will be shown in the following lemma
Let a be an element of G.
Then the order of a is k, such that $a^k \equiv 1 \pmod p$
By Lagrange's Theorem, the order of an element must divide the order of the group
But the order of the group G is p – 1 since the order of $(\mathbf{Z}/p\mathbf{Z})^*$ is $\varphi(p)$ and $\varphi(p) = p - 1$.
$\Rightarrow k \mid (p - 1) \Rightarrow p - 1 = km$, for some integer m.
$\Rightarrow a^{p-1} \equiv a^{km} \equiv (a^k)^m \equiv 1^m \equiv 1 \pmod p$ (QED)

Lemma – $(G, *) = (\mathbf{Z}/p\mathbf{Z})^*$ is a group

Proof: G is closed by multiplication under modular arithmetic
G is associative since multiplication associative
G has the identity element 1 by construction
Show G has inverses:
Every element b in G is relatively prime to p
Thus, $bx + py = 1$ has integer solutions x, y
However, this is equivalent to writing $bx \equiv 1 \pmod p$
Thus, G has inverses for all of its elements
Therefore, G is a group (QED)

From here, we note that if $a^k \equiv 1 \pmod p$, then $a^k - 1 \equiv 0 \pmod p$, thus p is a divisor of $a^k - 1$. So, if $a^k - 1$ is not divisible by a composite number n, then $\gcd(a^k - 1, n)$ is a proper divisor of n, thus a factor of n. This leads to:

Pollard's (p -1) Algorithm[4]: Given a positive integer n and a bound B, find a nontrivial factor g of n.

1. Compute the lcm: $m = \text{lcm}(1, 2, \ldots, B)$
2. Set a = 2
3. Compute $x = a^m - 1 \pmod n$ and $g = \gcd(x, n)$
4. If $g \neq 1$ or n, output g and terminate
5. If a < 10, set a = a + 1 and go to step 3, otherwise terminate

[4] Stein (130)

Note: a^m is computed using modular exponentiation.

A common concern that arises from using this algorithm is how to determine the smoothness bound B. The bound B should be small enough to ensure that the algorithm runs quickly, but large enough to ensure some reasonable chance of success. In practice, a bound for B is taken in the range $10^5 < B < 10^6$.[5] However, B is merely chosen based on the size of the smallest factor of n.

Below is code, implemented in SAGE, for calculating the lcm and the Pollard (p -1) algorithm.

```
#Calculates the lcm up to B
def lcm_upto(B):
   return prod([p^int(math.log(B)/math.log(p)) for p in prime_range(B+1)])[6]

#Uses the Pollard p -1 algorithm to factor a composite number
def pollard(n, B):
   m = lcm_upto(B)
   a = 2
   x = Mod(a, n)^m - 1
   g = gcd(x, n)
   while (g == 1 or g == n) and a < 10:
      a = a + 1
      x = Mod(a, n)^m - 1
      g = gcd(x, n)
   if g != 1 and g != n:
      return (g, n/g)
   return "failure"
```

Now, we shall give three examples using the Pollard (p -1) algorithm to factor n.

Example 1: Let (p, q) = (2003, 389). Thus, n = 779167
Using SAGE:

```
            SAGE: time print pollard(779167,  15)

            SAGE (Output): (2003, 389)
            SAGE (Output): Time: CPU 0.00 s, Wall: 0.00 s
```

Example 2:

```
         SAGE: p = next_prime(1940294583940)
         SAGE: print factor(p-1)
```

[5] Mollin,Richard. *RSA and Public-Key Cryptography.* Kennith Rosen. New York: Chapman & Hall/CRC, 2002. (97)
[6] Stein (130)

```
SAGE: q = next_prime(63729394029384)
SAGE: n = p*q
SAGE: time print pollard(n, 10^6)
SAGE (Output): 2 * 3 * 11 * 13 * 17 * 197 * 675251
SAGE (Output): (1940294583943, 63729394029409)
SAGE (Output): Time: CPU 0.32 s, Wall: 0.32 s
```

Example 3:

```
SAGE: is_prime(2^2*3*5*7*11^2 + 1)
SAGE: p = 2^2*3*5*7*11^2 + 1
SAGE: q = next_prime(ZZ.random_element(10^100))
SAGE: n = p*q
SAGE: print "n is " + str(n)
SAGE: time print pollard(n, 10^6)
SAGE (Output): n is
72681415718402454989136931108147780224770441970327662049197
2383634476082026493048006578621876523964455O9
SAGE (Output): (50821,
14301453280809597408381757759223112537094988679940902786091
819988478701364130832687404392315706577 29)
SAGE (Output): Time: CPU 0.48 s, Wall: 0.48 s
```

These examples show that the Pollard (p – 1) algorithm can factor numbers with about 100 digits fairly quickly if one of the factors is fairly small and B-power smooth. The factors that can be removed using this method fall in the range of eight to 15 digits. This is a vast improvement over trial division that could only handle numbers around 12 to 15 digits, thus having much smaller factors. Plus, the time to factor these numbers has also dropped from around five seconds to around one second. As stated above, the size of the numbers that can be factored using this method depend largely on the B that you choose as well as on the computer you are running the algorithm on. The larger the B and the more powerful the computer, the bigger the numbers are that you will be able to factor.

The downside to this method is that it requires n to have a prime factor p such that p – 1 has only small prime factors. Thus, if someone were constructing an RSA number, they might very easily pick two primes that are not small, i.e. larger than 20 digits, and that do not have the p – 1 property. If, for example, a person found a large prime p_1 such that $p = 2p_1 + 1$ is also prime and another large prime q_1 such that $q = 2q_1 + 1$ is also prime, then the resulting integer from p_1 and q_1 would be resistant to the p – 1 method.[7]

[7] Stinson (184)

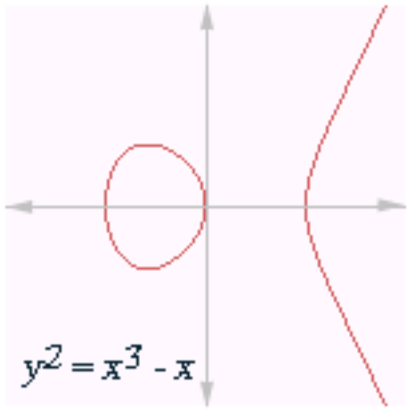
The Lenstra Elliptic Curve Method

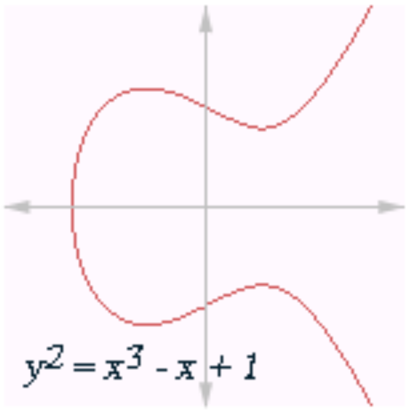
The Lenstra Elliptic Curve method was created by Hendrik Lenstra as a generalization to the Pollard (p – 1) algorithm. We know from the previous section that Pollard's method relies on the fact that one of the factors of a number must be B-power smooth for p – 1. But if none of the factors of a number is B-power smooth, then Pollard's method fails and we cannot find a factor. Lenstra, when creating the ECM, considered what was special about p – 1. He wondered if there was a generalization to p – t, for some t. This is where the ECM came into existence. Before we jump into the algorithm for the ECM, we must first introduce some background on elliptic curves.

Def: (Elliptic Curve) – An elliptic curve over a field K is a curve defined by an equation of the form $y^2 = x^3 + ax + b$, with $a, b \in K$ and $-16(4a^3 + 27b^2) \neq 0$

The lat part of the definition, $-16(4a^3 + 27b^2) \neq 0$, is the requirement that the curve be nonsingular, thus the graph has no cusps or self-intersections. An equation of the form described in the definition is called a Weierstrass Equation.

Below is a picture of two elliptic curves:





In the first example, the curve has a = -1 and b = 0, and in the second curve, a = -1, and b = 1.

Before we consider a dilemma that we encounter with the construction we have made, we must first introduce a few concepts.

Def: (Characteristic) – A field K is said to have characteristic $p \neq 0$ if for some positive integer p, $px = 0$ for all $x \in K$, and no positive integer smaller than p enjoys this property.

An interesting question that comes from this definition is what characteristics can a field enjoy? This question yields the theorem:

Theorem (Field Characteristic) – The characteristic of a field is 0 or p, a prime number.

Proof: Let K be a field with characteristic not 0.

$\Rightarrow$ n * 1 = 0 by the definition of characteristic.

$\Rightarrow$ n is the smallest such positive integer with this property

Assume that n is not prime, since if n were prime, then we are done

$\Rightarrow$ n = s * t since n is composite

$\Rightarrow$ 0 = n * 1 = (s * t) * 1 = (s * 1) * (t * 1)

$\Rightarrow$ s * 1 = 0 or t * 1 = 0 since K is an integral domain

But this is a contradiction since s, t < n and n is the smallest (QED)

Now we can discuss a dilemma that occurs with our description of an elliptic curve. Let K be a field with characteristic 2, i.e. K might be $\mathbf{Z}/2\mathbf{Z}$. Thus for any choice of a and b, we find that $-16(4a^3 + 27b^2) = 0$. So we have run into a problem, that all elliptic curves over a field with characteristic 2 have singularities. This also occurs for fields with characteristic 3. The easy solution to this problem is to just not consider any curves over fields with characteristic 2 or 3. But this does not sit quite right. So, to fix this problem, if we have a field with characteristic 2 or 3, then we will use curves of the form $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$. This allows us to do arithmetic with elliptic curves over those fields. The other reason we wish to allow curves over a field of characteristic 2 or 3 is that arithmetic on them is usually easier to handle and can be done quicker and more efficiently on a computer.

The natural thing to wonder about elliptic curves is the set of solutions to them. Is there a way to describe the set of solutions to an elliptic curve over a field K? How many solutions are there on an elliptic curve? To answer the first question, we define:

Def: (Set of solutions to an Elliptic Curve) –

Let $E(K) = \{ (x, y) \in K\times K \mid y^2 = x^3 + ax + b\} \cup \{\theta\}$, where $\{\theta\}$ is the point at infinity. (The point at infinity is a point which when added to the real number line yields a closed curve called the real projective line).

The second question we asked is what is the cardnality of E(K)? The answer to this question is a theorem by Hasse which states that $|p + 1 - \#E(\mathbf{Z}/p\mathbf{Z})| \leq 2\sqrt{p}$. Thus, we construct the number $a_p = p + 1 - \#E(\mathbf{Z}/p\mathbf{Z})$. We will use this number later on.

The next thing we would like to introduce is the group structure on elliptic curves that will allow us to understand the workings of the ECM. How are we going to inflict a group structure on an elliptic curve? We shall do so by putting the group structure on the set of solutions of the elliptic curve, E(K). To set up a group, we need to define the binary operation we will use. The binary operation is going to be addition on the points of the curve. Here is how we are going to do this:

Algorithm: (Elliptic Curve Binary Operation)[8] – Given P_1 and P_2, find $R = P_1 + P_2$

```
1. If $P_1$ is the point at infinity, then set $R = P_2$ or if $P_2$ is the point at infinity,
   then set $R = P_1$, and terminate
2. If $x_1 = x_2$ and $y_1 = -y_2$, set R to the point at infinity and terminate
3. Find the slope of the line between $P_1$ and $P_2$ by the following:
     a. If $P_1 = P_2$ then differentiate your elliptic curve to obtain
        $\lambda = (3x_1^2 + a)/(2y_1)$
     b. If $P_1$ and $P_2$ are unique, then use $(y_2 - y_1)/(x_2 - x_1)$
4. Set $x_3 = \lambda^2 - x_1 - x_2$ and $y_3 = -\lambda x_3 - \nu$, where $\nu = y_1 - \lambda x_1$
```

We now have that E(K) has a binary operation that is closed, commutative, the point at infinity is the identity element, and every point has an inverse element. The fact that addition is associative is also held, but a bit messy to do algebraicly. This can be shown geometrically, however, it will be omitted from this paper. So, we have shown that E(K) is an abelian group. We now introduce the ECM:

Algorithm: (ECM)[9] – Given a number n and a bound B, find a nontrivial factor of n.

```
1. Compute the LCM up to B (m = lcm(1, 2, ..., B))
2. Choose a random $a \in (\mathbf{Z}/n\mathbf{Z})$ such that $4a^3 + 27 \in (\mathbf{Z}/n\mathbf{Z})^*$, then P = (0, 1) is
   a point on the curve $y^2 = x^3 + ax + 1$ over $\mathbf{Z}/n\mathbf{Z}$
3. Compute mP using the group law. If we cannont compute the sum at some
   point because the denominator is not coprime to n, then compute the gcd of
   the denominator with n. If the gcd is a nontrivial divisor, terminate. If it is a
   trivial divisor, output 'fail'.
```

This algorithm has a major advantage over the Pollard (p – 1) method. This advantage is that if the ECM fails for one random elliptic curve, the ECM can be repeated with a different elliptic curve. The Pollard (p – 1) method is confined to $(\mathbf{Z}/n\mathbf{Z})^*$ but in the ECM, we can try many different groups $E(\mathbf{Z}/n\mathbf{Z})$ for many curves. This can be done since the number of points on an elliptic curve over $\mathbf{Z}/p\mathbf{Z}$ is of the form $p + 1 - t = a_p$ for some t with $|t| < 2\sqrt{p}$. Thus, if a prime p has the property that $p + 1 - t$ is B-power smooth, then the ECM has a chance at factoring n. Below is the code for the ECM, implemented in SAGE.

```
#Uses the ECM to factor a composite number
def ecm(N, B=10^3, trials=10):
   m = lcm_upto(B)
   R = Integers(N)
   # Sneaky: make Sage think that R is a field:
   def f(): return True
```

[8] Buchmann,Johannes. *Introduction to Cryptography.* Second ed. S. Axler. New York: Springer, 2004. (280)
[9] Stein (133)

```
    R.is_field = f
    for i in range(trials):
        while True:
            a = R.random_element()
            if gcd(4*a.lift()^3 + 27, N) == 1: break
        try:
            m * EllipticCurve([a, 1])([0,1])
        except ZeroDivisionError, msg:
            print msg
            # msg: "Inverse of <int> does not exist"
            return gcd(Integer(str(msg).split()[2]), N)
    return 1
```

Below are a few examples using ECM.

Example 1:

```
SAGE: p = next_prime(ZZ.random_element(10^100))
SAGE: q = next_prime(ZZ.random_element(10^8))
SAGE: n = p * q
SAGE: time print ecm(n, 10^4)
SAGE (Output): Inverse of
9642940894660325286590161352829897948840967517495608705211
956434633780021615732529922516585918244091247404l does not
exist
SAGE (Output): 77918219
SAGE (Output): Time: CPU 1.45 s, Wall: 1.45 s
```

Example 2:

```
SAGE: p = next_prime(ZZ.random_element(10^200))
SAGE: q = next_prime(ZZ.random_element(10^8))
SAGE: n = p * q
SAGE: time ecm(n, 10^6)
SAGE (Output): Time: CPU 1.27 s, Wall 1.28 s
```

Example 3:

```
SAGE: p = next_prime(ZZ.random_element(10^500))
SAGE: q = next_prime(ZZ.random_element(10^8))
SAGE: n = p * q
SAGE: time ecm(n, 10^6)
SAGE (Output): Time: CPU 43.98 s, Wall 44.03 s
```

You can see that for the most part, the size of the integer that you wish to factor has a small effect on the ECM's ability to factor the number. In the three examples above, the number to be factored had 108, 208, and 508 digits respectively. The ability of the ECM to factor an integer relies heavily on the how big the factors of the integer are. The ECM works best if the number to be factored has primes that are smaller than 20 digits. That being said, it should not be surprising that the ECM is used to factor non-RSA numbers since the prime factors of RSA numbers have hundreds of digits and ECM is optimized for splitting off small factors of a number.[10]

Continued Fraction Factorization Method (The Brillhart-Morrison Algorithm)

The CFFM was described by D.H Lehmer and R.E. Powers in 1931 and was made into an algorithm by Michael Morrison and John Brillhart in 1975. The CFFM is a general- purpose algorithm for factoring integers. By general-purpose, we mean that the algorithm can factor any integer, even if it's factors do not have any special properties. The CFFM is based off of Dixon's Factorization method and uses the convergents of a simple continued fraction expansion of √n, where n is the number you want to factor. Before we introduce the algorithm, we need to introduce the machinery we will use in this algorithm.

Def (Continued Fraction) – A continued fraction is a number x written in the form:

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \ddots}}}}$$

, where $a_i \in$ field and $a_i > 0$, $i > 0$

A finite continued fraction is one where the a_i will eventually halt for some i. Thus, the range of i is [0, n], for some n < ∞.

Note: For simplicity in writing the finite continued fraction, we will introduce some new notation. We will write the continued fraction for x as follows,

$x = [a_0, a_1, \ldots, a_n]$, where the a_i are the coefficients in the continued fraction.

Also, we will denote a continued fraction as being simple if all the a_i's are integers.

An observation to note is that $[a_0, a_1, \ldots, a_n] = [a_0, a_1, \ldots, a_{n-1} + 1/a_n]$

When you consider a fraction, sometimes you might be interested in the rounded decimal part of a fraction. There is a similar idea in continued fractions. In a continued fraction, we might only be interested in a portion of the coefficients in the continued fraction, i.e. $[a_0, a_1, \ldots, a_m]$ where m < n. We will call this smaller continued fraction the partial convergent of our number. To understand what is going on in the partial convergents, we introduce the theorem,

Def (Partial Convergents) – Define numbers p_n, q_n recursively as follows:

$p_{-2} = 0$, $p_{-1} = 1$, $p_0 = a_0$, …, $p_n = a_n p_{n-1} + p_{n-2}$

[10] Mollin (101)

$q_{-2} = 1, q_{-1} = 0, q_0 = 1, \ldots, q_n = a_n q_{n-1} + q_{n-2}$
then, $[a_0, a_1, \ldots, a_n] = p_n/q_n$

The proof of this theorem will be omitted however; the proof is done by induction on n and using the definition of partial convergents for p_n and q_n.

We can now describe the CFFM. We first construct the simple continued fraction of $\sqrt{n}$. Next, we let A_n/B_n be the nth convergent of the continued fraction. We then define a number Q_n as the following, $Q_n = A_{n-1}^2 - nB_{n-1}^2$. From here, we notice that that there is a well-known formula that can be reached form the equation for Q_n. Thus, $(-1)^n Q_n \equiv A_{n-1}^2 \pmod{n}$. From the definition of Q_n, we construct a set of $Q = \{Q_1, Q_2, \ldots, Q_n\}$. Next, we factor each of the Q_i into their primes. This can be done quickly using trial division as each of the $Q_i < 2\sqrt{n}$. The next step is to look for a subset of Q such that $\Pi_i(-1)^n Q_i$ is a square, thus $\Pi_i(-1)^n Q_i = X^2$. This is done by searching through and finding a group of Q_i whose prime factors will all be squares when multiplied together (See example below).[11]

The index of the Q_i's that are multiplied together must be kept track of since we then need to take the product of the corresponding $A_{n-1} \pmod{n}$ and set that equal to Y. Since we know that $A_{n-1}^2 = (-1)^2 Q_n$, then we obtain the relation $X^2 = \Pi_i (-1)^2 Q_i = \Pi_i A_{i-1}^2 = Y^2 \pmod{n}$. From here, there is a good chance that the gcd(X – Y, n) is a nontrivial factor of N. If the gcd is a trivial factor, take a larger partial convergence and repeat the steps above. It should be noted that the gcd is taken with X and Y and not X^2 and Y^2. The Y is easily found as the Y is calculated first before checking Y^2 with X^2. But how is the X created when we only have X^2 to begin with? This comes from how we picked our subset of Q. Below is a small example using the CFFM.[12]

Example:
Let n = 13290059.
We find that our set Q = {4034, 3257, 1555, ...}
Notice that $Q_5 = 2 * 5^2 * 41$, $Q_{22} = 41 * 113$, $Q_{23} = 2 * 113$
$\Rightarrow Q_5 * Q_{22} * Q_{23} = 2 * 5^2 * 41 * 41 * 113 * 2 * 113 = 2^2 * 5^2 * 41^2 * 113^2 = (2 * 5 * 41 * 113)^2$ (Thus, our $X = 2 * 5 * 41 * 113$)
$\Rightarrow Q_5 * Q_{22} * Q_{23} = 46330^2 = 6769401 \pmod{13290059}$
Multiple the corresponding A_{i-1}:
$A_4 = 171341, A_{21} = 175846005787, A_{22} = 271172278057$
$\Rightarrow A_4 * A_{21} * A_{22} = 1469504$
$\Rightarrow (A_4 * A_{21} * A_{22})^2 = 6769401 \pmod{13290059}$
$\Rightarrow$ Let X = 46330, Y = 1469504

[11] Vasilenko,O. N.. *Number-Theoretic Algorithms in Cryptography.* 232, New York: American Mathematical Society, 2007. (57)
[12] Pomerance, Carl. "Implementation of the Continued Fraction Integer Factoring Algorithm." *Congressus Numerantium* 37, no. (1983): 99-118.

$\Rightarrow$ gcd(46330 – 1469504, 13290059) = 4261
Thus, we have found a factor of 13290059, and that factor is 4261.

Of the algorithms described in this paper, the CFFM is the fastest, the ECM is the second fastest, the Pollard (p – 1) third, and trail division is the fourth. There are many other algorithms implemented for factoring RSA numbers, such as the Quadratic Sieve and the General Number Field Sieve. Both of these methods are beyond the scope of this paper but are just as interesting. Of the methods described above, the CFFM is the only one that might be used to factor RSA as it is a general-purpose algorithm and the other methods are special-purpose algorithms. The ECM and the Pollard method are used to take out small factors from a number and should be used when you are trying to factor a general number and not an RSA number. For RSA numbers, the best algorithms are the sieve methods just mentioned.

<u>Conclusion</u>

Factoring algorithms, like many algorithms in programming, should be implemented depending on the situation at hand. If you are interested in factoring small numbers, under 12 to 14 digits long, the simple trial division algorithm will suffice. However, as stated above, this is not a good choice for factoring an actual RSA number. Both the Pollard (p – 1) and the Lenstra ECM can handle much larger numbers. Both of these methods are special-purpose methods that can handle numbers whose factors have special properties. If the number you are trying to factor has the property one of its factors, p, has that p – 1 or p + 1 – t for some t, are smooth, then the Pollard and ECM methods are the most effective to use. Both of these algorithms are best at splitting off small factors, smaller than 20 digits. Thus, once again, these are not the best methods for factoring RSA numbers since the factors of those numbers are large, 100's of digits long. The last method, CFFM, is a general-purpose method for factoring integers. Thus, it does not have to worry about the factors having special properties. Also, as noted above, the CFFM is the fastest of the algorithms. It is also the best at factoring RSA numbers, as RSA numbers most likely will be constructed to resist having the special properties mentioned in this paper. So, in your quest to factor numbers, consider what you are trying to do, and pick the algorithm that best suites the situation.