AN ABSTRACT OF THE THESIS OF

James Simshaw for the degrees of Honors Baccalaureate of Science in Mathematics and Honors Baccalaureate of Science in Computer Science presented on May 30, 2008.

Title: Factorization of Integers.

Abstract Approved:

Robert Burton

Factorization of integers is an important aspect of cryptography since it can be used as an attack against some of the common cryptographic methods being used. There are numerous methods in existence for factoring integers. Some of these are faster than others for general numbers, while others work best on a specific category of integers. Currently, factorization takes a long time for large numbers, ensures the security of the cryptographic method.

Keywords: Factorization, Integers, Cryptography

Factorization of Integers

by

James Simshaw

A PROJECT

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Mathematics (Honors Scholar)

and

Honors Baccalaureate of Science in Computer Science (Honors Scholar)

Presented May 30, 2008
Commencement June 2008

Honors Baccalaureate of Science in Mathematics and Honors Baccalaureate of Science in Computer Science project of James Simshaw presented on May 30, 2008.

Approved:

Mentor

Committee Member

Committee Member

Dean, University Honors College

I understand that my project will become part of the permanent collection of Oregon State University, University Honors College. My signature below authorizes release of my project to any reader upon request.

James Simshaw, Student

Table of Contents

Introduction....1
Cryptography....2
    RSA Encryption....3
Factorizations....5
Fermat Factorization....7
Pollard p-1 Factorization....10
Elliptic Curves....12
Elliptic Curve Factorization....15
Run Time Analysis....17
Closing Remarks....20
Bibliography....21
Appendix A....23
    Main.c....25
    Mathfunc.c....27
    Ellipticops.c....28
    Fermat.c....29
    Pollard.c....30
    Elliptic.c....31
Appendix B....33

List of Figures

Figure 1: Elliptic curve $y^2 = x^3 + 3x + 2$ 13
Figure 2: Elliptic curve $y^2 = x^3 - 2x + 1$ 14
Figure 3: Elliptic curve $y^2 = x^3 - 10x + 15$ 14

List of Algorithms

Algorithm 1: RSA Encryption..4
Algorithm 2: Trial Division...5
Algorithm 3: Pollard P – 1..11
Algorithm 4: Elliptic Curve Method..16

List of Symbols and Acronyms

$\lceil x \rceil$ is the smallest integer greater to or equal to x

$O(f(x))$ is the "Big O" notation for running time.

$gcd(a, b)$ is the greatest common divisor of a and b.

$lcm(a, b)$ is the smallest common multiple of a and b.

Factorization of Integers

Introduction

Every positive integer can be categorized into one of two types, prime numbers and composite numbers. Prime numbers can only be written as the product of one and that number while composite numbers are written as the product of primes. The form that these numbers can be written as is $n = p_1^{t_1} * p_2^{t_2} * ... * p_m^{t_m}$ where t_i are integers greater than or equal to zero and p_1 to p_m are the prime numbers less than or equal to n. When given in canonical ordering, this representation is unique. Factorization is the decomposition of a composite number to its prime factors and finding that representation.

There are a lot of viable options available to one who wants to find the prime factors of an integer. However, these methods may not take the same amount of time, and they may have different classes of numbers that they are useful for. However, there still is no algorithm that can factor integers in a fast manner.

The greatest problem with factoring numbers is that it is a hard problem and takes a long time to determine a solution. Currently, a number of cryptographic systems are secure because of this. The faster we can factor numbers, the more these systems will need to use larger numbers for the keys, or new methods will have to start being used which do not rely upon factorization being a hard problem.

Cryptography

Information security and cryptography have brought around a time where factorization is an essential part of everyone's life, even if most are unaware of it. Many of the encryption methods that are currently employed require the use of factorization and/or could be broken if one of the composite numbers used in the method were factored.

Public key encryption is one of the branches of encryption systems. In this paradigm, one person, Alice, can release a public key to everyone, including Bob. With this key, Bob can encrypt a message and send it to Alice through any method he choses. These methods can be as public as he wants. Since the message is encrypted using Alice's public key, anyone who intercepts the encrypted message cannot read it because they do not have Alice's private key, which Alice is the only one who does have it. Once Alice receives the encrypted message, she can decrypt it using her private key to obtain the original message that Bob had written. Now, if Alice wanted to send a reply to Bob, then she would encrypt that message with Bob's public key and transmit the encrypted message to Bob via any method she so chooses. Again, if this message is intercepted, no one will be able to understand it because they do not have Bob's private key. However, once Bob receives the encrypted message, he can decrypt it with his private key to read the message that Alice had written. If someone was able to get either Alice's or Bob's private key, they would be able to decrypt any messages that were encrypted with that key's public key. This would mean that the public and private key pair are no longer secure and should not be utilized any further.

RSA Encryption

RSA encryption was created by Ronald Rivest, Adi Shamir, and Leonard Adleman in 1978. This was one of the first public key cryptosystems and is still widely used today (Yan).

To employ RSA encryption the public and private keys must first be created. In order to create these keys, two large primes must be chosen, called a and b. Let n be the product of a and b. Once n is computed, o, the public (encryption) key, and d, the private (decryption) key must be chosen so that $do=1\, mod\,((a-1)(b-1))$. The variable o was chosen to stand for obfuscate, while d was chosen to stand for decrypt. Our public (encryption) key must also be relatively prime to $(a-1)*(b-1)$. The person decrypting messages can now let anyone know o and n.

To encrypt the message that you want to send, it must first be in a numerical form. If that message is a computer file, that is a simple task since it is already stored in binary format and can be just be converted into a number from the binary string, which must be broken up into packets so that the number is not larger than n. After it is in a numerical format, the message is then raised to the power of the encryption key, o, and then we find the modular residue with respect to n, in other words $m^o\, mod\, n.$

Decryption is done in the exact same manner as encryption, just with a different key. You raise the encrypted message to the power of the decryption key. Once that is done, the original message will be the modular residue of the result with respect to n.

Since n is known to the public, if someone was able to factor n to its two prime factors, the private key could then be discovered, rendering all encrypted messages insecure. To find the decryption key once you have a and b, you just need to find a d such that $do = 1 \, mod \, ((a-1)(b-1))$.

In algorithm format, this looks like:

1. Pick two primes a and b.
2. Let n = ab.
3. Find $do = 1 \, mod \, (a-1)(b-1)$ and gcd(o, (a – 1)(b – 1)) = 1.
4. To encrypt: Put your message into a numerical format, call it m. Then calculate $c = m^o \, mod \, n$.
5. To decrypt: Calculate $m = c^d \, mod \, n$.

Algorithm 1: RSA Encryption. (Mathworld, RSA)

RSA encryption is considered secure because it takes a considerable amount of time to factor large numbers. If factoring became significantly faster, which could realistically happen, RSA encryption, as well as other factorization methods, will either become obsolete or require even larger numbers to be used.

Factorizations

The goal of any factorization method is to give us two factors of a number. Once we have those two factors, we can continue the factorization of the number by applying a factorization method to both of the factors if either or both of them are composite. This process continues until every factor is prime and the original number can then be written as a product of prime numbers.

The slowest method of factoring a number is trial division. However, this is also the easiest and most intuitive algorithm for an average person. The algorithm for this method is:

1. Set a = 2.
2. Set b = n/a.
3. If b is an integer, then a and b are factors of n, otherwise, increment a by one and as long as $a \leq \sqrt{(n)}$, go back to step 2.
4. If $a > \sqrt{(n)}$, then n is prime.

Algorithm 2: Trial Division.

This could be done a little quicker if instead of taking every integer, you increment in such a manner that a is always prime.

For someone who has to factor many numbers or large numbers, trial division could potentially take a really long time to complete, especially if the factors are large. This could have been especially problematic during the time when some of the first factorization methods were created, like Fermat Factorization, when people had to factor numbers by hand.

Fermat Factorization

Fermat factorization is one of the earliest methods utilized to factor an odd composite integer down to its prime factors. Pierre de Fermat was a lawyer who lived from 1601 to 1665, during an era where all computations had to be done manually and thus took far longer than it would take today with a modern computer (Wolfram Research Site).

Fermat factorization allows one to be able to factor a small composite odd integer down to its prime factors by hand; its applicability also extends to factor larger integers with the help of a computer. This is also one of the easiest method that can be programmed into a computer. However, to be programmed into a computer, it is useful to use one of the modified versions of the original algorithm so that computations can be kept to a minimum.

Any odd integer can be written as n = x*y where x and y are the factors of n. Since n is odd, the factors of n must also be odd. For simplicity sake, lets assume that x is greater than y. This implies that x + y and x – y must both be even positive integers. If we let a = (x + y) / 2 and b = (x – y) / 2, then x will equal a + b and y will equal a – b. This means we can rewrite n = x*y as n = (a + b)(a – b). We can simplify this to become $n = a^2 – b^2$. With this, we get that any odd integer can be written as the difference of two perfect squares.

One algorithm for Fermat Factorization is more meant for a pen and paper approach, but can work on a computer. It starts off with letting a = $\lceil(\sqrt{(n)})\rceil$ and b = 0.

Next, we test to see if the equation $a^2 – b^2 = n$ holds true for our a and b. If this is not true, we add one to b until $a^2-b^2<n$. Once it occurs that $a^2-b^2<n$, we increase a by one and with adding one to b until it is less than n again or exactly equal to n. Once we have a a and b such that $a^2-b^2=n$, we just need to take $(a+b)$ and $(a-b)$, each of those will be a factor of n.

This algorithm can be modified so that the equality is checked against 0 instead of n. Instead of taking when a^2-b^2 as our driving equation, we take a^2-b^2-n . We follow the same steps outlined above except we increase a by one when a^2-b^2-n is negative.

There is another way you can look at the algorithm. You can set it up so that you determine whether a^2-n is a perfect square. This algorithm has the advantage that you are only having to change one variable instead of two, which is a lot more useful for programming this algorithm compared to the other methods. We want to know when the square root of a^2-n is an integer. We can again start here with the $\lceil(\sqrt{(n)})\rceil$ and then for each iteration, increase the value of a by 1. Getting the factors is still as simple as it was above.

We can also modify the algorithm so that there are as few multiplications as possible in each iteration. The reason for implementing this algorithm in such a manner is because multiplications take longer to compute than additions. This can be accomplished by determining whether a^2-n is a perfect square. One needs only to examine the first few digits of a^2 and n since perfect squares come with a certain set of end digits. We can simplify the increasing of a by looking at $(a+1)^2-a^2$, which equals $2a+1$. This allows us to originally calculate $\lceil(\sqrt{(n)})\rceil^2-n$ and if that is not a perfect square we just keep adding $2a+1$ where we have a counter for what a should be by adding one to $\lceil(\sqrt{(n)})\rceil$ for each iteration (Mathworld, Fermat).

Pollard p-1 Factorization

The Pollard p – 1 factorization method was developed in 1974. This method is a probabilistic method where we are likely to get a prime factor within a certain amount of time, but could potentially take longer because of the randomness that this method utilizes. Those numbers that have a factor of p and for which p – 1 has no large factors are numbers which this method find factors for the fastest.

This algorithm hinges upon the greatest common divisor function returning a nontrivial divisor. If this nontrivial divisor is returned, we have found a factor for n. Another mathematical principle that Pollard p – 1 factorization relies upon is Fermat's little theorem (Crandall). This states that when p is an odd prime, then $2^{p-1}=1\, mod\ p.$ This means that p divides $2^{p-1}-1.$ If $k=(p-1)j$, we get $2^k mod\ p=2^{(p-1)j} mod\ p=(2^{(p-1)})^j mod\ p=1^j mod\ p=1\, mod\ p$. Which then leads to the idea that if p is a prime factor of n, then p will divide the greatest common divisor of 2^k-1 and n.

The algorithm is:

1. Pick a random base between 2 and n – 2, call this a. You can start at 2 and work your way up to n – 2.
2. Pick an upper bound for the integers that you want to test and call this b.
3. Let k = lcm(p_1, p_2, …, p_b) where p_1 to p_b are the first through bth primes.
4. Let $c = a^k \mod n$.
5. Let d = gcd(c – 1, n).
6. If d is not 1 or n, then d is a factor for n. Otherwise go back to the first step and pick a new base.

Algorithm 3: Pollard p – 1.

Elliptic Curves

Elliptic curves are third degree polynomials over a field. The generic form of an elliptic curve is

$$Ax^3+Byx^2+Cxy^2+Dy^3+Ex^2+Fxy+Gy^2+Hx+Iy+J=0$$ (Mathworld, Elliptic).

However, the typical form of an elliptic curve used in factorization is $y^2=x^3+ax+b$ (combination of sources).

The set of points on an elliptic curve form a group when combined with a form of addition. Call the operation +, which must satisfy certain properties. These properties are:

1. There exists a point called the point at infinity, labeled 0, which is the additive identity. Which means that for any point p = (x,y) on the curve, p + 0 = p.
2. For any p = (x,y) on the curve, -p = (x, -y), which is also on the curve.
3. For any two points p = (x_1, y_1) let q = (x_2, y_2) and r be the point where the line through p and q crosses the curve again. Then p + q = -r.
4. If p = -q, then p + q = 0.
5. If p = q, then p + q = -r (Koblitz)

There are a few things to note about some of these characteristics of an elliptic curve, of which

2. By having -p = (x, -y), we can see that it is on the curve since (-y)^2 = y^2 and -(-p) = (x, -(-y)) = (x,y).

4. The line through q and -q is a vertical line, which will never cross the curve, except at infinity, which is the point at infinity, and the negative of that is still the point at infinity.

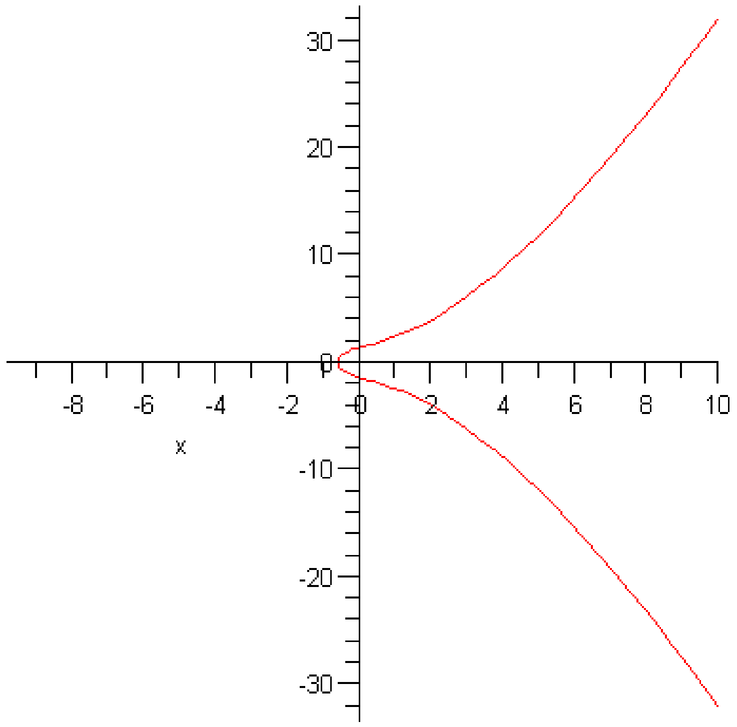

Figure 1: Elliptic curve $y^2 = x^3 + 3x + 2$

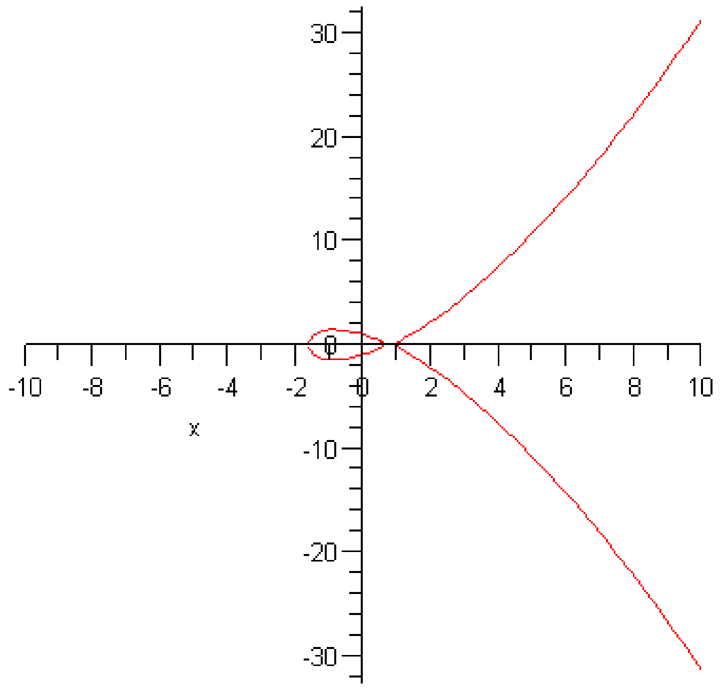

Figure 2: Elliptic curve $y^2 = x^3 - 2x + 1$

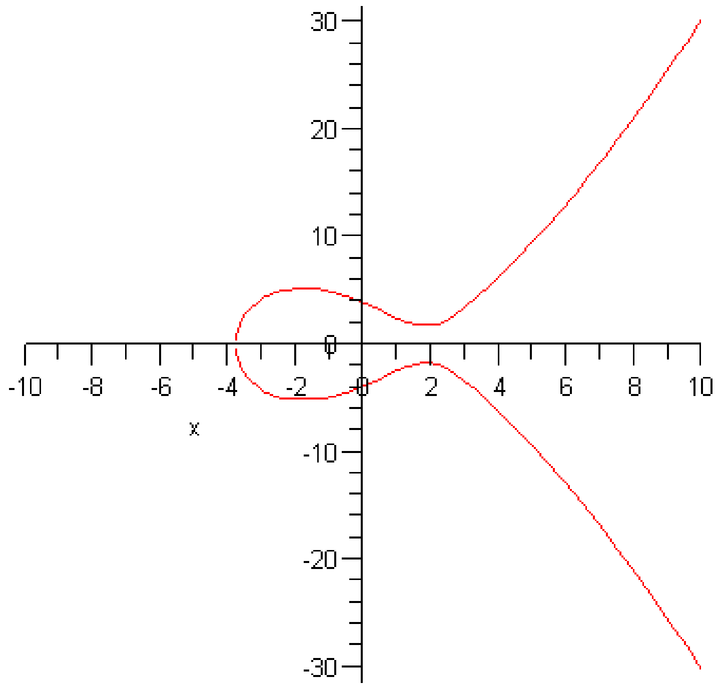

Figure 3: Elliptic curve $y^2 = x^3 - 10x + 15$

Elliptic Curve Factorization

The elliptic curve method for factorization was created in 1987 by H. W. Lenstra(Cohen). This method of factorization is similar to the Pollard p – 1 method of factorization. Following the idea of taking a different group instead of the group $2^k \mod p$ it was believed that you could perform similar operations as that done with the Pollard p – 1 factorization method (Yan). In this case the group is the set of points on an elliptic curve along with an operator.

As in the case of the p-1 factorization method, the elliptic curve method has two stages to it, though it was added after the original method was created around the same time the second phase was added to the p-1 factorization method(Zimmermann).

The first step to factor a number using the elliptic curve method is to generate an elliptic curve. We want a random curve in the form $y^2 = x^3 + ax + b$ over the field Zn, where $\gcd(4a^3 + 27b^2, n)$ equals 1. If $\gcd(4a^3 + 27b^2, n)$ does not equal 1 or n, then we have likely found a factor of n and if $\gcd(4a^3 + 27b^2, n)$ equals n, then we have not created an elliptic curve. Next we want to pick a point $p = (x_1,y_1)$ on our elliptic curve.

Next, we want to bound how many attempts we will make at trying to find a factor for n in this phase. The larger the number, the more attempts this phase will make at trying to find the factor, but also the greater chance that we will find a factor in this phase. Call this number B and let $k = \text{lcm}(1, 2, \dots, B)$.

Next, we need to calculate kp. kp is just p added via the group operation k times. If kp is not the point at infinity, then we need to choose a new curve and point. However, if kp is the point at infinity, we should then calculate gcd(m2, n). If the greatest common divisor is 1 or n, then we need to pick another curve and point and start over. Otherwise, we have a greatest common divisor between 1 and n, therefore it is one of the factors to n (Yan).

In algorithm form, this looks like :

1. Choose a random elliptic curve $y^2 = x^3 + ax + b$ over Zn.
2. Check to make sure $\gcd(4a^3 + 27b^2, n) = 1$. If it is n, choose another curve, otherwise, a factor of n has been found.
3. Choose a random point $w = (x_1, y_1)$ on the curve.
4. Choose a number and take the lcm of all the integers from 1 to that number and let that be called m.
5. Calculate mw by w + w + w + ... + w

   where $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$ and

   $(x_3, y_3) = (s^2 - x_1 - x_2 \bmod n, s(x_1 - x_3) - y_1 \bmod n)$ and

   $s = q/r = (3x_1^2 + a)/2y_1$ mod n if the two points are the same or

   $s = q/r = (y_1 - y_2)/(x_1 - x_2)$ mod n otherwise
6. If mw is not the point at infinity, go back to step 1 and try another curve.
7. Let d = gcd (r, n). If d is 1 or n, return to step 1 and try another curve, otherwise we have a factor.

Algorithm 4: Elliptic Curve Method. (Yan)

Run Time Analysis

How fast an algorithm is an important aspect of it. An algorithm can be certain to give an answer, but if the algorithm takes a thousand years to produce an answer, that answer is not going to be of much use to those living today.

When examining the running time of an algorithm, there are two different notation schema that one can use. The easiest to understand is “Big O” notation. It is written in the form $O(f(n))$ where expression is the term that controls the behavior of the function asymptotically as n approaches infinity. The other notation that is used is L notation. This notation allows for a more compact representation of some of the running times since $L(n)=e^{\sqrt{\ln n \ln \ln n}}$ (Cohen).

Speed in a general case is not the only consideration that should be looked into when deciding which algorithm is the fastest or is the one you would want to use when trying to factor a number. Most of the algorithms work better for certain classes of numbers, although there are some that will not work at all if you have other properties for the number. However, there are times when the characteristics of the number in question are not known, in which utilizing many methods in parallel would be a wise tactic in trying to factor the number.

Fermat Factorization works best for numbers in which its factors are close together and therefore close to the square root of n. The running time in O notation for Fermat Factorization is $O(\sqrt{(n)})$. The worst case occurs when n is prime. However, since this method comes across factors that are close to the square root of n first, this method works fast for numbers that would be slow with trial division.

Pollard's p – 1 algorithm is an example of an algorithm that most of the time will give an answer quickly, but has special cases where it could take tremendously long to get a factor. In the case $(p-1)/2$ is prime, the p – 1 algorithm takes $O(\sqrt{(n)})$ time to finish. This is no better than that of either Fermat factorization or trial division. However, for numbers for which p – 1 has no large prime factors, this method should give us an answer in a reasonable amount of time, depending partially upon the randomness that is evident in the methodology.

The elliptic curve method has a runtime based upon the size of the factor which we are finding. This method is best suited to finding factors up to twenty five digits, though that would take a good portion of a day. Factors smaller than fifteen digits would be considerably faster than those of twenty five digits.

With the differences in the classes of numbers that each of these methods can factor with ease, the choice of which algorithm to implement becomes difficult. If you know the number has factors close to one, trial division or the elliptic curve method would be methods that could find those factors fast. However, if you know that the factors are close to $\sqrt{(n)}$, Fermat factorization would be a method to run. Finally, if you know that the factor of n, p, has no large prime factors, Pollard's $p-1$ method would be one to try. In general, it is unlikely you will know anything about the properties of the number, so the easiest way to go about it would be to run the algorithms in parallel and then one of them should come across a factor. If resources are constrained, it may be necessary to limit the number of methods attempted at one time. The final option is to look for a more complex, general purpose, algorithm not discussed in this paper.

Closing Remarks

The problem of factoring integers is currently be a hard problem, but one day, it will likely become a problem that can be solved in a short amount of time. There are more complicated methods that work for different classes of numbers as well as others that could factor numbers slightly faster. Technological advances may also render the modern methods of factoring integers obsolete. Another possibility is that a new method could one day be discovered that can factor integers in a manner no one has thought of yet. The elliptic curve method is one example that shows us that we are still coming up with new methods using tools that may not have been applied in the past.

Currently, those cryptographic methods that gain their security from factorization being a hard problem are relatively safe. This, however, can and will change once integers become easy to factor, either by technological advances or advances in methodology. Once this happens, there are many other fields of mathematics that could be utilized to create cryptographic systems.

Bibliography

Bressoud, David M. Factorization and Primality Testing. Undergraduate Texts in Mathematics. New York: Springer, 1989.

Cohen, Henri. A Course in Computational Number Theory. Graduate Texts in Mathematics 138. New York: Springer, 1993.

Crandall, Richard and Carl Pomerance. Prime Numbers: A Computational Perspective. 2nd ed. New York: Springer, 2005.

Koblitz, Neal. A Course in Number Theory and Cryptography. Graduate Texts in Mathematics 114. New York: Springer-Verlag, 1987.

"Least Common Multiple." Wikipedia. May 26, 2008 <http://en.wikipedia.org/wiki/Least_common_multiple>

Riesel, Hans. Prime Numbers and Computer Methods for Factorization. 2nd ed. Boston: Birkhäuser, 1994.

Stopple, Jeffrey. A Primer of Analytic Number Theory: From Pythagoras to Riemann. Cambridge, UK: Cambridge University Press, 2003.

Weisstein, Eric W. "Elliptic Curve." From MathWorld--A Wolfram Web Resource. May 26, 2008 <http://mathworld.wolfram.com/EllipticCurve.html>

Weisstein,Eric W. "Fermat's Factorization Method." From MathWorld--A Wolfram Web Resource. May 26, 2008 <http://mathworld.wolfram.com/FermatsFactorizationMethod.html>

Weisstein, Eric W. "RSA Encryption." From MathWorld--A Wolfram Web Resource.
<http://mathworld.wolfram.com/RSAEncryption.html>

Yan, Song Y. Number Theory for Computing. 2nd ed. Berlin: Springer, 2002.

Appendix A

The following code samples are for an application that can factor integers using the three methods outlined in this paper. It is a text based application written in C. It does not implement any arbitrary precision libraries. This is just one of the many ways that these algorithms could be programmed.

Main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void menu() {

    int selection = 0;;
    int num = 0;

    printf("Input the number to factor(0 to exit): ");
    scanf("%d", &num);

    while (num > 0) {
        printf("Select a factorization method:\n");
        printf("1. Fermat Factorization\n");
        printf("2. Pollard P-1 Factorization\n");
        printf("3. Ellitpic Curve Factorization\n");

        scanf("%d", &selection);
        // Insert command to clear the screen here.
        switch(selection){
        case 1: fermat(num);
            break;
        case 2: pollard(num);
            break;
        case 3: elliptic(num);
            break;
        default: printf("Invalid selection, please choose again.\n");
            menu(); break;
        }

        printf("Input the number to factor(0 to exit): ");
        scanf("%d", &num);
    }

}
```

```
int main(int argc, char *argv[])
{
  menu();
  return 0;
}
```

Mathfunc.c

```
#include <stdio.h>

int gcd(int a, int b) {
    if (a % b == 0)
       return b;

    return gcd(b, a%b);
}

unsigned long lcm(int a, int b) {
  unsigned long mult = (a*b)/gcd(a,b);

  return (a*b)/gcd(a,b);           // formula found on wikipedia
}
```

Ellipticops.c

```
#include <stdio.h>

void ellipticAdd(int point1[], int point2[], int curve[]) {
   //only does addition for curves for factorization
   int sum[3], l;


   if (point1[0] == point2[0] && point1[1] == point2[1]) {
      if (point1[1] != 0) {
         sum[2] = (2 * point1[1]);
         l = (3 * point1[0] + curve[0]) / sum[2];
      }
      else
         ;
   }
   else if (point1[0] != point2[0]) {
      sum[2] = (point1[0] - point2[0]);
      l =  (point1[1] - point2[1])/sum[2];
   }
   else
      ;
   sum[0] = (1 - point1[0] - point2[0]) % curve[2];
   sum[1] = (1 * (point1[0] - sum[0]) - point1[1]) % curve[2];

   point2[0] = sum[0];
   point2[1] = sum[1];
   point2[2] = sum[2];
}
```

Fermat.c

```
#include <stdio.h>
#include <math.h>

void fermat(int n) {

    int a = floor(sqrt(n));
    int diff;
    float sqdiff;

    printf("Starting Fermat Factorization of %d\n", n);
    printf("Starting with a value of %d for a\n", a);
    diff = n + a*a;
    sqdiff = sqrt(diff);
    while(sqdiff != floor(sqdiff)) {
        printf("%d + %d*%d = %d, which is not a square.\n",n,a,a,diff);
        a--;
        diff = n + a*a;
        sqdiff = sqrt(diff);
    }
    printf("%d + %d*%d = %d, which is a square.\n",n,a,a,diff);
    printf("The factors of %d are %d and %d.\n", n, (int)(sqdiff+a), (int)(sqdiff-a));
}
```

Pollard.c

```
#include <stdio.h>
#include <time.h>

void pollard(int n) {

    int base, bound, power, c, divisor;

    srand(clock());  // initialize the random number generator
    printf("Entered Pollard P-1 Factorization Function for %d\n", n);
    do {
        base = rand() % (n - 4) + 2;  // our base needs to
                                      // be between 2 and n-2
        printf("Using a base of %d.\n", base);
        bound = 0;
        power = 1;
        c = 1;
        divisor = 1;
        printf("Enter an upper bound: ");
        scanf("%d", &bound);
        int i;
        for (i = 1; i <= bound; i++)
            power = lcm(power, i);
        printf("Power is %d\n", power);
        for (i = 1; i <= power; i++)
            c = (c * base) % n;
        printf("C is %d\n", c);
        divisor = gcd(c - 1, n);
    }while (divisor == 1 || divisor == n);
    printf("The factors of n are %d and %d.\n", divisor, n/divisor);
}
```

Elliptic.c

```
#include <stdio.h>
#include <time.h>

void elliptic(int n) {

    srand(clock());  // initialize the random number generator
    long divisor, ecurvechk, bound, m;
    int point[3], point2[3], curve[3];

    printf("Entered Ellitpic Curve Factorization Function for %d\n", n);

    printf("Enter an upper bound: ");
    scanf("%d", &bound);

    do {
       do {
         //Creates the curve
         curve[0] = rand() % n;
         point[0] = rand() % n;
         point[1] = rand() % n;
         curve[1] = (point[1]*point[1] - point[0]*point[0]*point[0] - curve[0]*point[0]) %
                  n;
         curve[2] = n;
         if (4*curve[0]*curve[0]*curve[0] + 27*curve[1]*curve[1] < 0) {
           printf("Integer value too large for this curve, overflow detected\n");
           ecurvechk = n;
         }
         else {
             ecurvechk = gcd(4*curve[0]*curve[0]*curve[0] + 27*curve[1]*curve[1], n);
         }
       } while (ecurvechk == n);
       if (ecurvechk != 1) {
         divisor = ecurvechk;
         printf("Divisor found from initial check.\n");
       }
       else {
          int i;
          m = 1;
          point2[0] = point[0];
```

```
            point2[1] = point[1];
            for (i = 1; i <= bound; i++)
               m = lcm(m, i);
            for (i = 2; i <= m; i++)
               ellipticAdd(point, point2, curve);
            divisor = gcd(point2[2], n);
         }
      } while(divisor == 1 || divisor == n);
      printf("Using the curve y^2 = x^3 + %d * x + %d.\n", curve[0], curve[1]);
      printf("The factors of %d are %d and %d.\n", n, divisor, n / divisor);
   }
```

Appendix B

```
Input the number to factor(0 to exit): 143

Select a factorization method:

1. Fermat Factorization

2. Pollard P-1 Factorization

3. Ellitpic Curve Factorization

1

Starting Fermat Factorization of 143

Starting with a value of 11 for a

143 + 11*11 = 264, which is not a square.

143 + 10*10 = 243, which is not a square.

143 + 9*9 = 224, which is not a square.

143 + 8*8 = 207, which is not a square.

143 + 7*7 = 192, which is not a square.

143 + 6*6 = 179, which is not a square.

143 + 5*5 = 168, which is not a square.

143 + 4*4 = 159, which is not a square.

143 + 3*3 = 152, which is not a square.

143 + 2*2 = 147, which is not a square.

143 + 1*1 = 144, which is a square.

The factors of 143 are 13 and 11.

Input the number to factor(0 to exit): 143

Select a factorization method:
```

```
1. Fermat Factorization

2. Pollard P-1 Factorization

3. Ellitpic Curve Factorization

2

Entered Pollard P-1 Factorization Function for 143

Using a base of 24.

Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 135.

Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 28.

Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 54.

Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 71.
```

```
Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 51.

Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 79.

Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 74.

Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 97.

Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 54.

Enter an upper bound: 10

Power is 2520
```

```
C is 1

Using a base of 80.

Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 84.

Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 137.

Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 139.

Enter an upper bound: 10

Power is 2520

C is 1

Using a base of 99.

Enter an upper bound: 10

Power is 2520

C is 66

The factors of n are 13 and 11.
```

```
Input the number to factor(0 to exit): 143

Select a factorization method:

1. Fermat Factorization

2. Pollard P-1 Factorization

3. Ellitpic Curve Factorization

3

Entered Ellitpic Curve Factorization Function for 143

Enter an upper bound: 10

Divisor found from initial check.

Using the curve y^2 = x^3 + 123 * x + 36.

The factors of 143 are 11 and 13.

Input the number to factor(0 to exit): 143

Select a factorization method:

1. Fermat Factorization

2. Pollard P-1 Factorization

3. Ellitpic Curve Factorization

3

Entered Ellitpic Curve Factorization Function for 143

Enter an upper bound: 10

Using the curve y^2 = x^3 + 36 * x + 126.

The factors of 143 are 13 and 11.

Input the number to factor(0 to exit):0
```