Comparing the heuristic running times and time complexities of integer factorization algorithms

Computational Number Theory

New Mexico
Supercomputing Challenge
April 1, 2015
Final Report

Team 14
Centennial High School

Team Members

Devon Miller
Vincent Huber

Teacher

Melody Hagaman

Mentor

Christopher Morrison

1 Executive Summary

Mathematical and computational cryptography has been the subject of massive scrutiny and analysis in recent years as computer security struggles to keep up with the exponentially expanding computer market. The most common form of encryption is public key cryptography which subsists of large prime numbers. In public key cryptography, two very large coordinates of a function are given in modular arithmetic form, with modulo p (p equals a very large prime number). This system is very secure and has sustained it's integrity for the many years of it's implementation but there is a well known weakness in it's security. Factoring the very large semi prime values in the equation would allow an attacker to compute the hidden exponent d from a public key (the aforementioned coordinates) and thus decrypt c, the RSA cipher text, using the learned d, x, y, and p values. This would obviously allow the attacker to predict any encrypted information being sent through the encryption system. The problem of factoring such large numbers is what makes this feat such an infeasible one but much research is done into the efficiency of existing factorization algorithms. We mathematically analyzed the time complexities of a sample of integer factorization algorithms and proved their theoretical efficiency over one another. Then, in Java, we used the algorithms and a simple time metric to calculate their rough computational runtime in milliseconds across a number of trial, eventually testing with higher integers. We then compared the results of each algorithm with each other and with their theoretical runtimes in Big-O and L Notation. This project helped us understand the complexity and sophistication of integer factorization algorithms and displayed experimental data of heuristic performance times.

2 Introduction

2.1 Motivation

Our inspiration in this project was a result of our interest in mathematics and modern cryptography. RSA encryption, currently being the most significant form of public key cryptography, caught our attention. After investigating further we saw importance in integer factorization in both cryptography and number theory in general, and our project quickly shifted it's focus from cryptography to number theory. Initially we planned to code in Python but we

quickly nullified that idea. We then considered C++ due to it's strength in measuring running time and complexity metrics however we eventually decided against it due to it's foreignness. Finally we concluded on using Java due to it's familiarity and more complex yet simple math functions. With this in mind, we spent much of our time mathematically analyzing the algorithms in big-O notation.

2.2 Public Key Cryptography

In public key cryptography two cryptographic keys are used, one private, one public. These keys are linked through a mathematical algorithm and correspond to each other. The public key which, as it's name suggests, is fully released to the public, encrypts any outgoing bits while the private key decrypts cipher text or in some cases provides a digital signature. In this way public key cryptography is often referred to as a form of asymmetric cryptography since it's encryption and decryption employ the use of different algorithms. This strategy works because where a symmetric cryptographic system could be compromised by exposing the private key that is passed between individuals an asymmetric system makes it computationally infeasible to determine a personalized private key from it's corresponding public key. To generate a key two distinct prime numbers *p* and *q* are selected and multiplied to compute a semi prime *n* that is designated as the modulus for both the private and public keys. To compute the exponents *e* and *d*, for the public and private key respectively, Euler's totient function is used as follows $\varphi(n) = \varphi(p)\varphi(q) = (p-1)(q-1) = n - (p + q - 1)$ where φ is Euler's totient function. $1 < e < \varphi(n)$ is the range in which integer *e* is selected. The totient function can be shown with Euler's product formula (Figure 2.2.1).

$$\begin{aligned}
\varphi(n) &= \varphi(p_1^{k_1})\varphi(p_2^{k_2})\cdots\varphi(p_r^{k_r}) \\
&= p_1^{k_1}\left(1-\frac{1}{p_1}\right)p_2^{k_2}\left(1-\frac{1}{p_2}\right)\cdots p_r^{k_r}\left(1-\frac{1}{p_r}\right) \\
&= p_1^{k_1}p_2^{k_2}\cdots p_r^{k_r}\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\cdots\left(1-\frac{1}{p_r}\right) \\
&= n\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\cdots\left(1-\frac{1}{p_r}\right).
\end{aligned}$$

Figure 2.2.1

This function is initially used to compute exponent *e* and exponent *d* is computed as the modular multiplicative inverse of *e* (modulo φ (n)). The public key contains *n* and exponent *e* while the private key contains *n*, exponent *d*, and the aforementioned *p* and *q* primes. All components of the private key other than the modulus are kept secret. In essence, the modulus and exponents are used in conjunction with a padding scheme, which is employed to prevent *m* (the sent message in integer form) from falling into range of plaintext. Given the relationship between the *n*, *e*, and *d* values the encryption and decryption processes are fairly easily delineated (Figure 2.2.2). Variable *c* stands for ciphertext.

Encryption **Decryption**

$$c \equiv m^e \pmod{n} \qquad m \equiv c^d \pmod{n}$$

Figure 2.2.2

2.3 Integer Factorization

As you may have caught in the first Euler's totient function equation, the *p* and *q* values are used to calculate the exponents in addition to their product *n*. Modulus *n* is the public portion of the key generation and is freely distributed whilst *p* and *q* are the privatized portion. It seems that breaking this system can be accomplished by simply factoring *n*. This is indeed simple in theory however the enormity of *n* makes it slightly less so. Factoring very large numbers, commonly known as the RSA problem, is an important and fundamental problem in modern number theory. The highest number factored so far is RSA-704 which is a 212 digit number that was factored by Shi Bai, Emmanuel Thorné, and Paul Zimmermann on July 2, 2012. While an incredible accomplishment, this number is no where near the titanic size of modulus *n* which is a 2048 bit number that clocks in at 617 decimal digits. RSA security believes this key size will likely be sufficient until 2030.

While this information provides context for our project we will not be focusing on RSA encryption due to time constraints and the complexity of the algorithms used for RSA factorization. We will be focusing instead on the big-O notation and mathematical analysis of

elementary integer factorization algorithms but will include these more complex algorithms in our mathematical analysis in Section 3.

2.4 Big O Notation

Big O notation, in computer science, is used to classify algorithms by their responses to varying changes in input size and data. Big O notations details the asymptotic analysis of functions as they tend to infinity or a specified value. In essence big O notation describes functions by their corresponding growth rates. A growth rate of a function is also known as the order of the function which explains the term big O. Determining O notation is a rather simple process. Locate the term in function *f* that grows at the fastest rate omitting all other terms and then omit any constants on the term. Example:

$$y = 5x^3 + 4x^2 + 7$$

$5x^3$ grows at the fastest rate therefore we omit the other terms

$5x^3$ has a constant and therefore is omitted

x^3 is the remaining term

We then write

$$f(x) = O(x^3)$$

Since we can classify algorithmic time as orders (e.g O(log(n)) or O(log(log(n)))) we can use the asymptotic bounds of their orders to calculate their response to changes in input size.

3 Mathematical Procedure

3.1 Mathematical Description

In the context of our project, we chose three different factorization algorithms: Dixon's Algorithm, Euler's Factorization Method, and Pollard Rho's Algorithm.

3.1.1 Dixon’s Algorithm

Dixon's algorithm is centralized on finding a congruence of squares modulo the integer N which is designated as the integer we are attempting to factor. Example:

N = 84923

5052 mod 84923 = 256 (16^2)

(505 − 16)(505 + 16) = 0 mod 84923

gcd(505 − 16 and N) =163

N is factorable by 163

More precisely, Dixon’s algorithm can be summarized in the following equations:

$$z^2 \equiv \prod_{p_i \in P} p_i^{a_i} \pmod{N}$$

$$z_1^2 z_2^2 \cdots z_k^2 \equiv \prod_{p_i \in P} p_i^{a_{i,1}+a_{i,2}+\cdots+a_{i,k}} \pmod{N}$$

3.1.2 Fermat’s Factorization Method

Fermat's factorization method is based on the representation of an odd integer as the difference of two squares:

$$N = a^2 - b^2.$$

This difference is factorable as (a+b)(a-b). If neither factor equals one, it is a proper factorization of N. If N = cd is a factorization of N, N is therefore represented below:

$$N = \left(\frac{c+d}{2}\right)^2 - \left(\frac{c-d}{2}\right)^2$$

In a worst case input scenario Fermat's factorization method clocks in slighter slower than the common trial division method of most factorization algorithms share roots with.

3.1.3 Pollard Rho's Algorithm

The algorithm first receives inputs n, the integer to be factored and p(x), a polynomial computed modulo n. Once these inputs are received the algorithm performs the following basic steps:

$x \leftarrow 2;\ y \leftarrow 2;\ d \leftarrow 1;$

While d = 1:

$x \leftarrow p(x)$

$y \leftarrow p(p(y))$

$d \leftarrow gcd(|x - y|, n)$

If d = n, return failure.

Otherwise return d

Also with pointing out is that this algorithm may fail to find nontrivial factors even though n is a composite number. In which case, the algorithm can be run again, using a different g(x).

3.2 Mathematical Method

By employing the varying complexities of these algorithms we can plug in various inputs to calculate their theoretical running times and compare them with that of experimental data. The complexity of Dixon's algorithm can be stated explicitly in big O notation as:

$$O\left(\exp\left(2\sqrt{2}\sqrt{\log n \log\log n}\right)\right)$$

assuming optimal complexity. This can be related to the upper bound size of the factor base required in the algorithm:

which is also the $\exp\left(\sqrt{\log N \log\log N}\right)$ hardest part of the

algorithm to calculate. Fermat's factorization method bears similarity to Dixon's algorithm as they both employ congruence of squares, however Fermat's factorization method is likely to be more efficient than Dixon's algorithm as Dixon's algorithm approaches the complexity of calculating an appropriate optimal bound for the size of it's factor base.

In Pollard Rho's Algorithm if the pseudorandom number x = g(x) occurring in the Pollard ρ algorithm were an actual random number, then successful factorization would be achieved half the time which delineates its complexity as a correlation with the Birthday Paradox. The Birthday Paradox discerns the probability that, in a set of n randomly chosen people, some pair of them will have the same birthday. Unusually, under these circumstances 99.9% probability is reached with just 70 people, and 50% is reached with 23 people. The formula of the paradox is as follows:

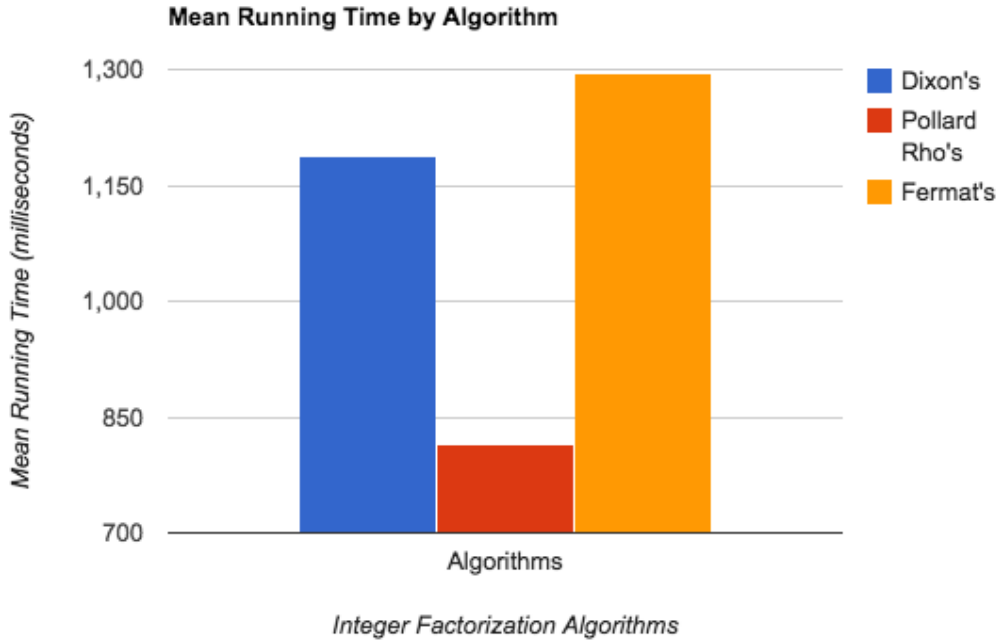
$$P(A') = 1 - \frac{365!}{365^N(365-(N))!}$$

Due to the expedient halved time of the Pollard Rho Algorithm we hypothesize that it will achieve the greatest calculated running time but on average will suffer the lowest running time. Pollard Rho's algorithm will therefore be the fastest, followed by Fermat's factorization method, and lastly Dixon's Algorithm. To calculate running time we will insert the following code into the main method of each algorithm. To obtain semi primes that could be inputted to each algorithm we used a Random Semi prime Generator implementation in Java.

```
long startTime = System.currentTimeMillis();
.....your program....
long endTime   = System.currentTimeMillis();
long totalTime = endTime - startTime;
System.out.println(totalTime);
```

4 Results


Running Time by Trial
3,000
2,300
1,600
900
200
Running Time (milliseconds)
1
2
3
4
5
Trial
Pollard Rho
Dixon
Fermat


Mean Running Time by Algorithm
1,300
1,150
1,000
850
700
Mean Running Time (milliseconds)
Algorithms
Integer Factorization Algorithms
Dixon's
Pollard Rho's
Fermat's

5 Conclusion

5.1 Concluding Evaluation

From our results we can determine that we were partially correct in our hypothesis of the integer factorization running times, showing that Pollard Rho's algorithm was quite faster than both of the other algorithms. Dixon's algorithm however came ahead of Fermat's factorization method in terms of running time due to the complexity of the factor base which to our surprise improved its running time by allowing a faster method *B-smooth* method as opposed to Fermat's factorization method. However these results are inconclusive due to the unpredictable nature of the running times. All three algorithms had major spikes in running time at seemingly arbitrary points which is both a result of the aforementioned worst case scenarios and unpredictable CPU speeds.

5.2 Future Work

We found that much of our project had to be omitted due to time constraints and the complexity of the algorithms. Much of our data is promising but ultimately inconclusive as we did not reliably account for variables such as CPU speeds and processing power. We performed these calculations on a 2.5 GHz Intel Core i5 processor and we hope in the future we will have access to other, more complex processors that will allow us to perform more elaborate calculations and make use of parallel processing. The algorithms we chose were effectively modeled and calculated but we wish we had opportunities to implement more complex algorithms, most notably the General Number Field Sieve which is known for its efficiency and significant presence in modern cryptography. In the future we plan on both continuing this project and achieving definitive results and also approaching a new topic in modern cryptography as it is a field that sparks great interest in both of us.

6 Appendix

6.1 Bibliography

Lenstra, Arjen K. “Integer Factoring.” Kluwer Academic Publishers, Boston, 2000. Web. 3 Dec 2014.

Jr. M. S. Manasse A. K. Lenstra, H. W. Lenstra and J. M. Pollard. The number field sieve. In "Proceedings of the twenty second annual symposium on theory of computing", pages 564–572. ACM Press, 1990. 3 Dec 2014.

Song Y. Yan. Primality Testing and Integer Factorization in Public-Key Cryptography. Kluwer Academic Publishers, 2004. Web. 4 Dec 2014.

K. Bimpikis and R. Jaiswal. “Modern Factoring Algorithms.” University of California, San Diego. Web. 2 Dec 2014

E. Barker and J. Kelsey. “Recommendation for Random Number Generation Using Deterministic Random Bit Generators.” Computer Security Division, Information Technology Laboratory. Web. 4 Dec 2014

All pictures are courtesy of *Wikipedia*

6.2 Code

6.2.1 Dixon's Algorithm

```
import java.util.Scanner;
public class Dixon_Algorithm
{
  long startTime = System.currentTimeMillis();
    public static void main(String args[])
    {
      long startTime = System.currentTimeMillis();
      Scanner ob=new Scanner(System.in);
      System.out.print("Enter Number:");
    double n=ob.nextDouble();
    dixon(n);
    long endTime   = System.currentTimeMillis();
    long totalTime = endTime - startTime;
    System.out.println(totalTime);
    }
    public static double  gcd(double a,double b)
    {
       if(b==0)
       return a;
       else if(b>a)
       return gcd(b,a);
       else
       return gcd(b,a%b);
    }

    public static int  checkprime(double n)
    {
    int f=0;
      for(int i=2;i<=n/2;i++)
      {
       if(n%i==0)
       {
          f=1;
          break;
       }
            }
           if(f==1)
    return 0;
```

```
  else
return 1;
}
public static void dixon(double n)
{
   int a,d,b1,d1;
   double m,x,p,q;
   m=Math.sqrt(n);
   double c=Math.floor(m);
   int f1=0;
   for(double f=c+1;f<=n;f++)
   {
      double s,m1;
      s=Math.sqrt((f*f)%n);
      m1=Math.floor(s);
      if((s-m1)==0)
      {
         if(s!=(f%n))
         {
         p=f+s;
         q=f-s;
         b1=checkprime(p);
         d1=checkprime(q);
          if(b1==1)
          {
         double z=gcd(q,n);
         int b=checkprime(z);
          if(b==1){

         System.out.println("Factors = "+(int)z+" and "+(int)p);


           }
          }
          else if(d1==1)
          {
         double z=gcd(p,n);
         int b=checkprime(z);
          if(b==1){

         System.out.println("Factors = "+(int)z+" and "+(int)q);}
          }
         f1=1;
```

```
            break;
            }
         }
            if(f1==1)
            break;
      }
   }
}
```

6.2.2 Fermat's Factorization Method

```
import java.util.Scanner;
public class Fermat_Factorization_Algorithm
{
   public void FermatFactor(long N)
   {
      long a = (long) Math.ceil(Math.sqrt(N));
      long b2 = a * a - N;
      while (!isSquare(b2))
      {
         a++;
         b2 = a * a - N;
      }
      long r1 = a - (long)Math.sqrt(b2);
      long r2 = N / r1;
      display(r1, r2);
   }
   public void display(long r1, long r2)
   {
      System.out.println("Factors = "+ r1 +" and "+ r2);
   }
   public boolean isSquare(long N)
   {
      long sqr = (long) Math.sqrt(N);
      if (sqr * sqr == N || (sqr + 1) * (sqr + 1) == N)
         return true;
      return false;
   }
   public static void main(String[] args)
   {
```

```
      long startTime = System.currentTimeMillis();
       Scanner scan = new Scanner(System.in);
       System.out.println("Enter number:");
       long N = scan.nextLong();
       Fermat_Factorization_Algorithm ff = new Fermat_Factorization_Algorithm();
       ff.FermatFactor(N);
      long endTime   = System.currentTimeMillis();
      long totalTime = endTime - startTime;
      System.out.println(totalTime);
    }
}
```

6.2.3 Pollard Rho's Algorithm

```
import java.util.Scanner;
public class PollardRho_Algorithm
{
   private static final long C = 1;
  private long f(long X)
   {
      return X * X + C;
   }
   private long rho(long N)
   {
      long x1 = 2, x2 = 2, divisor;
      if (N % 2 == 0)
         return 2;
      do
      {
         x1 = f(x1) % N;
         x2 = f(f(x2)) % N;
         divisor = gcd(Math.abs(x1 - x2), N);
      } while (divisor == 1);
      return divisor;
   }
   public  long gcd(long p, long q)
   {
      if (p % q == 0)
         return q;
      return gcd(q, p % q);
   }
```

```
  public boolean isPrime(long N)
  {
     for (int i = 2; i <= Math.sqrt(N); i++)
        if (N % i == 0)
           return false;
     return true;
  }
  public void factor(long N)
  {
     if (N == 1)
        return;
     if (isPrime(N))
     {
        System.out.println(N);
        return;
     }
       long divisor = rho(N);
     factor(divisor);
     factor(N / divisor);
  }
  public static void main(String[] args)
  {
    long startTime = System.currentTimeMillis();
     Scanner scan = new Scanner(System.in);
     System.out.print("Enter number:");
     long N = scan.nextLong();
     System.out.println("Factors =");
     PollardRho pr = new PollardRho();
     pr.factor (N);
    long endTime   = System.currentTimeMillis();
    long totalTime = endTime - startTime;
    System.out.println(totalTime);
  }
}
```

6.2.4 Random Semi prime Generator

```
import java.util.Random;
public class Prime_Number_Generator {
  public static void main(String[] args) {
    boolean run = true;
```

```
    while (run) {
      int num1 = 0;
      int num2 = 0;
      Random rand = new Random();
      num1 = rand.nextInt(1000) + 1;
      num2 = rand.nextInt(1000) + 1;

      while (!isPrime(num1)) {
         num1 = rand.nextInt(1000000000) + 1;
      }
      while (!isPrime(num2)) {
         num2 = rand.nextInt(1000000000) + 1;
      }
      int semiprime = num1 * num2;
      String str = Integer.toString(semiprime);
      int length = str.length();
      if (length != 10)
        run = true;
      else
        run = false;
        System.out.println(semiprime);
    }
  }
  private static boolean isPrime(int inputNum){
      if (inputNum <= 3 || inputNum % 2 == 0)
         return inputNum == 2 || inputNum == 3;
      int divisor = 3;
      while ((divisor <= Math.sqrt(inputNum)) && (inputNum % divisor != 0))
         divisor += 2;
      return inputNum % divisor != 0;
  }
}
```