Tianning Sun:
Team leader: Decision of the experiment theme. Tasks assignment to team members. Report reviewing and writing.
Researcher: In this research, I focus more on the Pollard p-1 algorithm. I give introduction in the presentation and write the section about Pollard p-1 algorithm. Related to others' work, I further define the suitable smooth bound for this algorithm and prove that this bound will have more than 50% possibility to finish factorization. Additionally, I make innovative work to deal with the situation that p and q share the same smooth bound. Moreover, based on the goal to break RSA algorithm, I re-analyse the time complexity of this algorithm. Finally, I implement this algorithm in Java based on my optimization.
In group works, I provide theoretical optimization opinions and prove for other team members.
In the final report, I analyse the four algorithms based on their time complexity. Additionally, I tried to apply the Monte Carlo Method, Dixon Factorization, binary division method for all these algorithms, which does improve the efficiency.

Weiguang Ma:
Responsible for the implementation and analysis of the Pollard Rho algorithm in this group project. This student collected relevant literature of the Pollard Rho algorithm and read the literature with all group members as well as studied this algorithm. Moreover, he tried to write algorithmic code by using Java programming language, then debug and optimize the java codes. In addition, this student also responsible for introducing the contents of Pollard Rho in the presentation, including definitions, problem background, implementation ideas and brief analysis of the algorithm. The Pollard Rho algorithm part of the project report is also completed by this student, in which the content related to the algorithm is discussed in detail, including the algorithm history, introduction to the mathematical theory of the algorithm, using examples to illustrate the algorithm and comparison with other fast decomposition algorithms. In this project, this student and the team members learn together, discuss in depth and actively participate in the project.

Yu Chao:
In this project, the part I am mainly responsible for is the introduction and the trial division in the presentation and the report.
After the theme of our project was decided, I searched and understood in depth with the four prime factorization methods we used in our experiment - Trial Division, Fermat Factorization, Pollard's Rho Algorithm and Pollard's P-1 Algorithm. Additionally, I wrote the Python program to generate the data set and Java program for the trial division used in the experiment.
During the production of the video, I proposed my own solution to the problem that had arisen through discussions with the other team members. For example, I provided the screen recording software – Bandicam and made the template of the PowerPoint. In addition, I successfully finished my own part in the presentation by summarizing the methods and data set we used, and a brief introduction of the trial division.
When we started to write the report, I finished the introduction part and the trial division

part with more details to explain and analyze.

Wai Yan Wong:

In this experiment, firstly, I have attended group meetings and share my ideas with my group member about the project.

Secondly, I have read some articles on fast prime factorization methods, such as Trial Division, Pollard rho Factorization, Brent's Factorization Method and XXX. In terms of this, I have a general understanding about several integer factorization methods now. And I decided to choose Fermat's factorization method as one of the experimented methods in this assignment.

In addition, I have implemented this method with Java, and test its efficiency with various input. Moreover, I have made some slides related to Fermat's factorization method and introduced this method during the presentation.

Lastly, I have written the concept, implementation and analysis of this method in this report.

The University of Melbourne

School of Computing and Information Systems

COMP90043
Cryptography and Security

Project – Fast Prime Factorization

Semester 2 – 2018

Group Number: 61

Tianning Sun

Weiguang Ma

Yu Chao

Waiyan Wong

1 Introduction

1.1 Aim

The aim of this project is to research, analyze, and implement different factorization methods against the RSA algorithm. By comparing the efficiency based on their time complexity, we can enhance the knowledge of prime factorization and get the fastest methods.

1.2 RSA algorithm

The RSA algorithm is a public key cryptosystem constructed by number theory and considered to be quite mature theoretically. It is widely used in key management and digital signature.And the security of RSA algorithm is based on the difficulty of large integer factorization in number theory. The attack on the RSA encryption system comes from many aspects, mainly including factorization and bypass attack methods, and the attack on the encryption algorithm is a sign to verify the security of the algorithm.

2 Methodology

2.1 Algorithm

In this experiment, we focus on the research, analysis, and implementation of four algorithm: trial Division, Fermat's factorization method, Pollard rho algorithm, and Pollard p-1 algorithm.

- The trial division is the simplest algorithm for dealing with prime factorization. It just tests to see if an integer n, the integer to be factored, can be divided by each number in turn that is less than n.

- The Fermat's factorization method is to construct x and y, so that the integer n can be presented as $x^2 - y^2 = n$

- The Pollard Rho algorithm is the combination of trial division and the Monte Carlo Method.

- The Pollard p-1 algorithm is an application of Fermat's little theorem.

2.2 Dataset

The dataset used in this experiment is consisted of 2135 composite numbers. Each composite number is the result of multiplication of two prime numbers created by a Python program. To make the experiment closer to the reality, we add steps to check are both prime factors of n strong prime numbers.

3 Experiment

3.1 Trial Division

3.1.1 Introduction

Among these four factorization methods, trial division is the simplest but also the most time-consuming algorithm. This method is first proposed by Fibonacci in his book Liber Abaci(1).

3.1.2 Concept

For a given composite number n, we can see from the beginning of 2 whether n can be divisible by it. If it can, then 2 is one of the factors of n. If not, try to divide n by 3 until $\sqrt{n}$. That is to say, the trial division will test whether each number less than or equal to $\sqrt{n}$ can be divisible by n.

Algorithm 1 Basic trial division

```
1: function TRIALDIVISION(n)
2:     f ← 2
3:     if n ≡ 0 (mod f) then
4:         return n = n ∗ (n/f)
5:     else
6:         f+ = 1
7:     end if
8: end function
```

3.1.3 Time Complexity

In the worst case, trial division requires $\sqrt{n}$ divisions to complete the calculation. So the complexity is $O(\sqrt{n})$.

3.1.4 Implementation

In the RSA algorithm, since n must be obtained by multiplying p and q while p and q are prime numbers. Therefore, we can create a prime number table to reduce the number of comparisons and improve the efficiency of the trial division. In this way, the time complexity will be decreased as $O(\frac{\sqrt{n}}{\ln n})$(2).

Algorithm 2 Trial division

```
1: function TRIALDIVISION(n)
2:     t is a prime number table
3:     for all p in t do
4:         if n ≡ 0 (mod p) then
5:             return n = n ∗ (n/p)
6:         end if
7:     end for;
8: end function
```

3.1.5 Influence On RSA algorithm

For small composite numbers, trial division can decompose them with the help of computers. But for large numbers, such as a 1024-bit number, it could take $5.95 * 10^{211}$ years which is longer than the age of the universe(3). Therefore, for the RSA algorithm, the trial division method cannot effectively attack and crack the number n.

3.2 Fermat's Factorization Method

3.2.1 Introduction

Fermat's factorization method was invented by Fermat to deal with integer factorization. It bases on the theory that an odd number can be represented as the result of a perfect square minus another perfect square(4).

3.2.2 Concept

By Fermat's theory, an odd number can be represented as the result of a perfect square

minus another perfect square:

$$N = x^2 - y^2 \tag{1}$$

In this equation, x^2 - y^2 can be represented as $(x+y)(x-y)$. Thus, we can get

$$N = x^2 - y^2 = (x+y)(x-y) \tag{2}$$

Because N is an odd number, so all of its factors must be odd numbers as well. Let a and b become the two factors of N, we can easily get that:

$$a = (x+y), b = (x-y) \tag{3}$$

After solving these two functions, we can get:

$$x = \frac{x+y}{2}, b = \frac{x-y}{2} \tag{4}$$

Because a, b are two factors of N, so $a \geq b$, $b \geq 1$.

And for x and y, $x \geq \sqrt{n}$, $y \geq 1$. In terms of these, in order to find the factors of N, a proper x should be chosen which meet the requirement that $x^2 - N$ is a perfect square. If there is a particular x that meets this requirement, y can be easily calculated by getting the square root of $x^2 - N$. Hence, a and b can be calculated easily. However, the only way of finding the proper x and y is enumeration. Thus, it is of great importance finding the range of x to minimize the complexity. Because $N = ab$, so $b = \frac{N}{a}$, $x = \frac{a+b}{2} = \frac{a+\frac{N}{a}}{2} \geq \sqrt{N}$. Further more,$x \geq y+1$, $x^2 - y^2 = N$, so $N \geq x^2 - (x-1)^2 = 2x - 1$, $x \leq \frac{N+1}{2}$. So, the range of x is ($\sqrt{N}$,$fracN + 12$].

Algorithm 3 Fermat's Factorization Method

```
1: function FindPrimeFactor(n)
2:     for x from √N to fracN + 12 do
3:         if x^2 − N = ⌊√(x^2 − N)⌋^2 then
4:             return N = ((x+√(x^2−N))/2)((2N)/(x+√(x^2−N)))
5:         else
6:             p++
7:         end if
8:     end for;
9: end function
```

3.2.3 Time complexity

As the pseudocode shown above, there is a loop from $\sqrt{N}$ to $fracN + 12$ in Fermat Factorization function. In the worst case, the loop might do $fracN + 12 - \sqrt{N}$ times. So, the complexity of this method is $O(n)$.

3.2.4 Implementation

When this method is applied in RSA, N must be a large number. And the time complexity of $O(N)$ means that the algorithm may take a large amount of time for finding N's prime factors.

One way for improving the efficiency of this method is applying the properties of tens digit and unit digit of perfect square.

For any perfect square, the last two digits must be one of the following combinations: $00, e1, e4, 25, o6, e9$, where e means an even number and o means an odd number(5).

With this property, it is able to simply check the last two digits of $x^2 - N$, instead of calculating the square root of $x^2 - N$ for each

x and checking if the answer is an integer. If the combination of these two digits of the result is an element of the set shown above, the result must be a perfect square. Thus, the speed of this method would be well improved. It is obvious that for a and b, one is odd and the other is even. So there are $100 * 50 * 50$ combinations about the last two digits of a result of $x^2 - N$. Among all these combinations, the number of 00 is 2900, $e1$ is 30000, $e4$ is 30000, 25 is 2900, $o6$ is 45000, and $e9$ is 30000. So after filtering and ignoring the time cost to calculate the last 2 digital of $x^2 - N$, the time cost will decrease by $1-(2900+30000+30000+2900+45000+30000)/250000 = 45.52\%$. However, the time complexity remains $O(n)$ because this method only increase the algorithm by a constant rate.

3.2.5 Influence On RSA algorithm

With the increase of N, the range from $\sqrt{N}$ to $fracN + 12$ would increase as well. This means that the number of operations the system might do would also increase. In RSA, prime numbers p and q are big numbers. Thus, it might take a large amount of time for finding these two prime factors. This method is not suitable for RSA.

3.3 Pollard Rho algorithm

Although the trial division is a simple method that can implement integer decomposition easily, that algorithm has a high time complexity and is inefficient for decomposing particularly large integers. In this section, we introduce and analyze a fast and efficient integer decomposition algorithm: The Pollard Rho algorithm.

3.3.1 Introduction

Pollard's rho algorithm is an algorithm for quickly decomposing integers. It was implemented by John Pollard in 1975 to increase the efficiency of finding a composite factor by introducing the idea of birthday paradox [1]. The algorithm will provide a reliable and efficient way to break RSA.

3.3.2 Concept

Assuming that an integer N can be decomposed into a two-multiplied form, $N = pq$ where p and q are not equal. In order to factorize N, a random number between 2 and $N - 1$ is generated and check whether it is a factor of N.
Here, we introduce the birthday paradox to improve the efficiency of decomposition.
For the birthday paradox, among k students, the possibility that each student's birthday is different is $\frac{365-1+1}{365} * \frac{365-2+1}{365} * \frac{365-3+1}{365} * ... * * \frac{365-k+1}{365} = \prod_{i=1}^{k} \frac{365-i+1}{365}$. When the probability is greater than 50%, the value of k is $\sqrt{365}(6)$.
So if we generate a series of random numbers x_1, x_2, x_3, x_4,..., and x_m. If any pair of x_i, x_j that makes $x_i = x_j$, we use a function $f()$ to make $f(x_i, x_j, n) = q$. In this way,if we generate $sqrtn$ numbers, there is more than 50% possibility to find q.
Considering that the space complexity of this algorithm is $O(\sqrt[4]{n})$ if we store all the random

numbers, a scheme is designed to store only two numbers a and b in memory at a time. Both a and b are generated by a random number generating function.

Algorithm 4 Pollard Rho algorithm

```
1: function PollardRho(n)
2:     a ← random()
3:     b ← random()
4:     while 1 do
5:         if a = b then
6:             return f(a, b, n)
7:         else
8:             a = b (mod n)
9:             b ← random(1, n − 1)
10:        end if
11:    end while
12: end function
```

3.3.3 Time complexity

We know that n can be presented as $n = pq$, if we suppose $p \geq q$, then we can get $q \leq \sqrt{n}$. So we just generate random number in the range of $[2, \sqrt{n}]$. If we generate more than $\sqrt[4]{n}$ numbers, there is more than 50% possibility that we can get the prime number q, thus the time complexity of this algorithm is $O(\sqrt[4]{n})$

3.3.4 Implementation

There are aspects that we should consider during our experiment.

- The random number generating function $g()$: actually, we just choose $g(x) = x^2 + a$.
- The function $f()$:we choose $f(a, b, n) = gcd((a - b \pmod n), n)$. If $f()$ doesn't return 1, then $f()$ will be a prime factor of n
- Infinite loop processing:However, for some cases, this algorithm will fall into an infinite loop(7).To solve this problem, an algorithm proposed by Floyd is introduced to change the $f()$. Let $x_{slow} = g(x_1)$ and $x_{quick} = g(g(x_1))$. x_{slow} and x_{qucik} work as a pair of slow and quick iterators to detect the appearance of infinite loop.

3.4 loop

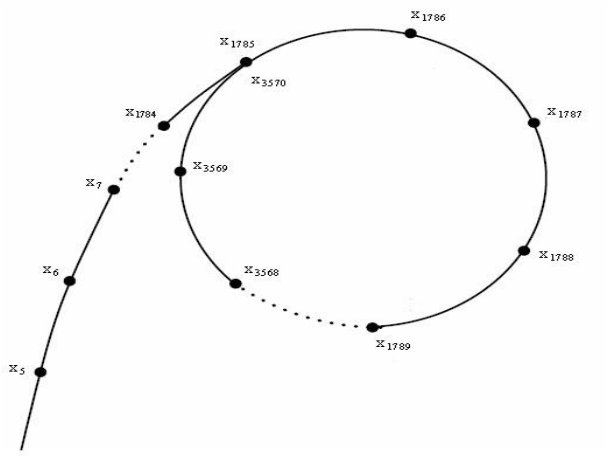

Figure 1: Infinite loop in Pollard rho algorithm

In addition, in these random numbers between 0 and $sqrtn$, the probability that they are different from each other is $\frac{\sqrt{n}-1+1}{\sqrt{n}} * \frac{\sqrt{n}-2+1}{\sqrt{n}} * \frac{\sqrt{n}-3+1}{\sqrt{n}} * ... ** \frac{\sqrt{n}-k+1}{\sqrt{n}} = \prod_{i=1}^{\sqrt[4]{n}} \frac{\sqrt{n}-i+1}{\sqrt{n}}$.

When this probability is 50%, about $\sqrt[4]{n}$ numbers have been generated, and this probability is at the constant level, thus it does not have a large impact on the time complexity of the algorithm.

Algorithm 5 Pollard Rho algorithm

```
1: function PollardRho(n)
2:     a ← 2
3:     x_1 ← 2
4:     y ← Handle(N)
5:     PollardRho(y)
6:     PollardRho(N/y)
7: end function
8:
9: function Handle(n)
10:    floydAlgorithm(n)
11:    returny
12: end function
13:
14: function floydAlgorithm(n)
15:    x_1 ← g(x_1^2 + a) (mod n)
16:    x_2 ← (x_2^2 + a) (mod n)
17:    x_2 ← (x_2^2 + a) (mod n)
18:    y ← gcd(abs(x_1 − x_2), n)
19: end function
20:
21: function g(x)
22:    return (x^2 + a) (mod n)
23: end function
```

3.5 Trial Division

3.5.1 Introduction

The Pollard p-1 algorithm is another algorithm invented by John Pollard to deal with the integer factorization problem(8). And it is an application of basic number theory.

3.5.2 Concept

By Fermat's little theorem, we can get (5), where p is a random prime number, a(usually 2) is coprime to p, and k is a random integer. It is obvious that $p-1$ is a compromise number and is divisible by some of the primes that are less than p.

$$a^(k(p-1)) \equiv 1 \pmod{p} \tag{5}$$

We construct M as the (6), where p_i is the ith prime from 2 and p_j is the largest prime that is less than p. So that it can proved that M is divisible by $p-1$ thus we can also find the k that makes $k(p-1) = M$.

$$M = \prod_{i=1}^{j} p_i^{\lfloor \log_{p_i} n \rfloor} \tag{6}$$

So we can get the (7). If the number we want to factorize is $n = p*q$ and $a^M - 1 = b*p$, the greatest common divisor of $a^M - 1$ and n is $gcd(a^M - 1, n) = gcd(b*p, p*q) = gcd(b,p)*p$. In this way, we can get a factor of n is $gcd(b,p)*p$.

$$M = \prod_{i=1}^{j} p_i^{\log_{p_i} n} = k(p-1) \tag{7}$$

3.5.3 Time Complexity

As $p * \log logp$ times are needed generate all the prime numbers that are less than p,

Algorithm 6 Pollard p-1 algorithm

```
1: function POLLARDP-1(a,n)
2:     b ← n^0.3
3:     M = getM(b)
4:     tag1 ← gcd(a^m − 1, n)
5:     tag2 ← gcd(a^m − 1, n)
6:     while tag1 = tag2 do
7:         b ← b − 1
8:         M = getM(b)
9:         tag2 ← gcd(a^m − 1, n)
10:    end while
11:    if tag2 = 1||tag2 = n then
12:        return failure
13:    else
14:        return n = n/tag2 ∗ tag2
15:    end if
16: end function
17: function GETM(b)
18:    m ← 1
19:    p ← 2
20:    while p ≤ b do
21:        if p.isPrime then
22:            m ← m ∗ p^⌊log_p n⌋
23:        end if
24:        p ← p + 1
25:    end while
26: end function
```

$\log n$ times are needed for each prime to construct $2^M \pmod n$, and another $\log n$ times are needed to execute $gcd()$ function, the total time complexity of this algorithm is $p * \log\log p * (\log n)^2$. Sometimes, $p-1$ can be presented as a product of many small prime numbers, which means that there exists a bound B(smooth bound) where any prime that is larger than B are not needed to construct M, and the p_j in (6) is the largest prime number that is less than B. In this case, the time complexity of Pollard p - 1 can be extremely small, which is $B * \log(\log B * (\log n)^2$.

3.5.4 Implementation

The core of this algorithm is to find a small but efficient smooth bound B. Here we suppose the two prime factors of n is p and q, and p is less than q. From Dixon's factorization method(9), we can get the possibility that both p and q are less than $n^(1/1.5569)$(calculated by Newton–Raphson method) is 50%. This also means that the possibility that q is larger than $n^0.7$ and p is less than $n^0.3$ is more than 50%. So if the smooth bound is chosen as $n^0.3$, there is more than 50% possibility that p can be factorized from n. In this way, the time complexity is $n^0.3^(\log n)^2$.

This algorithm returns $gcd(b,p) * p$ as its output. Here we designed a mechanism to process the output. If the first output is 1, which means that original smooth bound is too small, we should increment the bound B by 1 gradually until the output is not 1 but O. If the first output is n, which means

that original smooth bound is too large, we should decrease the bound B by 1 gradually until the output is not n but O. If O is neither n nor 1, O is the a prime factor of n, and the process of factorization succeed. In this way, all the situations that q - 1 has a different smooth bound with p - 1 will be solved because every smooth bounds will be processed in the function. However, if O is n or 1, which means that p - 1 and q - 1 share the same smooth bound. In this case, the typical Pollard p - 1 is not suitable. Successful and failed example is given below:
Successful example:
$N = 35$
$B = 4$
$B = 4$
$M = (2^5) * (3^3) = 864$
$a^M - 1 = 2^8 64 - 1$
$gcd(a^M - 1, n) = 35$
$B - 1 = 3$
$M = (2^5) * (3^3) = 864$
$B - 1 = 2$
$M = 32$
$a^M - 1 = 2^3 2 - 1$
$gcd(a^M - 1, n) = 5$
$N = 5 * 7$

Failed example
$N = 247$
$B = 6$
$M = (2^7) * (3^5) * (5^3) = 3888000$
$a^M - 1 = 2^3 888000 - 1$
$gcd(a^M - 1, n) = 247$
$B - 1 = 5$
$M = (2^7) * (3^5) = 31104$
$a^M - 1 = 2^3 1104 - 1$
$gcd(a^M - 1, n) = 247$
$B - 1 = 2$
$M = 2^7 = 128$
$a^M - 1 = 2^1 28 - 1$
$gcd(a^M - 1, n) = 1$

However, for some p and q where $p - 1$ and $q - 1$ share the same smooth bound, we invented a mechanism to deal with this situation. It is obvious that we can get Formula 4 and Formula 5, and we can also factorize $a^M - 1$ as Formula 6. Notice that the factorization factors and of Formula 6 can be further factorized based on Formula 4 and Formula 5.

$$2^2 a - 1 = (2^a - 1)(2^a + 1) \tag{8}$$

$$2^(2a+1) \pm 1 = (2 \pm 1)(2^a \mp 2^{a-1} + 2^{a-2} \mp ... \mp 2 + 1) \tag{9}$$

$$2^M - 1 = (2^{\prod_{i=1}^{j} p_i^{\lfloor \log_{p_i} n \rfloor}} - 1)$$
$$= (2^{\frac{\prod_{i=1}^{j} p_i^{\lfloor \log_{p_i} n \rfloor}}{2}} - 1)(2^{\frac{\prod_{i=1}^{j} p_i^{\lfloor \log_{p_i} n \rfloor}}{2}} + 1) \tag{10}$$

And the (10) can be further factorized as Formula (11)

$$2^{k(a+1)} \pm 2^{ka} + 2^{k(a-1)} \pm ... \pm 2 + 1 =$$
$$(2^{a+1} \pm 2^a + 2^{a-1} \pm ... \pm 2 + 1) tobedefined(2) \tag{11}$$

In this way, we can check if n is divisible by these factorization formulas to determinate the specific prime factors of n is the smooth bound of p - 1 and q - 1 is the same. Example

is shown below: $N = 247$
$B = 6$
$M = (2^7) * (3^5) * (5^3) = 3888000$
$a^M - 1 = 2^3888000 - 1$
$gcd(a^M - 1, n) = 247$
$B - 1 = 5$
$M = (2^7) * (3^5) = 31104$
$a^M - 1 = 2^31104 - 1$
$gcd(a^M - 1, n) = 247$
$B - 1 = 2$
$M = 2^7 = 128$
$a^M - 1 = 2^128 - 1$
$gcd(a^M - 1, n) = 1$
$BacktoB = 6$
$M = (2^7) * (3^5) * (5^3) = 3888000$
$a^M - 1 = 2^3888000 - 1 = (2^194000 - 1)(2^194000 + 1)$
$gcd(2^1944000 - 1, n) = 247$
$gcd(2^1944000 + 1, n) = 1$
$2^194000 - 1 = (2^972000 - 1)(2^972000 + 1)$
$gcd(2^972000 - 1, n) = 247$
$gcd(2^972000 + 1, n) = 1$
$...2^121500 - 1 = (2^60750 - 1)(2^60750 + 1)$
$gcd(2^60750 - 1, n) = 19$
$gcd(2^60750 + 1, n) = 13$

So 247 is factorized as 19 * 13.The pseudo-code advanced Pollard p - 1 algorithm is below:

3.5.5 Influence On RSA algorithm

To prevent these cases in RSA system, the concept of strong prime number(or safe prime numbers). For any strong prime number p, p - 1 has a large prime factor d, which means the smooth bound ofp - 1 is also large. And if the two prime numbers of RSA algorithm p

Algorithm 7 Pollard p-1 algorithm

```
1: function POLLARDP-1(a,n)
2:     b ← n^0.3
3:     M = getM(b)
4:     tag1 ← gcd(a^m − 1, n)
5:     tag2 ← gcd(a^m − 1, n)
6:     while tag1 = tag2 do
7:         b ← b − 1
8:         M = getM(b)
9:         tag2 ← gcd(a^m − 1, n)
10:    end while
11:    if tag2 = 1 then
12:        b ← n^0.3
13:        M = getM(b)
14:        factorization(M, n, 0)
15:    else
16:        if tag2 = n then
17:            factorization(m, n)
18:        else
19:            return n = n/tag2 * tag2
20:        end if
21:    end if
22:    if tag2 = 1||tag2 = n then
23:        return failure
24:    else
25:        return n = n/tag2 * tag2
26:    end if
27: end function
28: function GETM(b)
29:    m ← 1
30:    p ← 2
31:    while p ≤ b do
32:        if p.isPrime then
33:            m ← m * p^⌊log_p n⌋
34:        end if
35:        p ← p + 1
36:    end while
37: end function
```

Algorithm 8 factorization

```
1: function FACTORIZATION(b)
2:     if x = 0 then
3:         if m%2 = 0 then
4:             m ← m/2
5:         end if
6:         if gcd(2^m − 1, n) = n then
7:             factorization(m, n, 0)
8:         else
9:             if gcd(2^m + 1, n) = n then
10:                factorization(m, n, 1)
11:            else
12:                return n = n/gcd(2^m − 1, n) ∗ gcd(2^m − 1, n)
13:            end if
14:        end if
15:    else
16:        k ← the smallest odd prime number of m
17:        if gcd(2^(m/k) − 1, n) = n then
18:            factorization(m/k, n, 0)
19:        else
20:            if gcd(2^(m/k) − 1, n)! = 1 then
21:                n ← n/gcd(2^(m/k) − 1, n) ∗ gcd(2^(m/k) − 1, n)
22:            else
23:                return failure
24:            end if
25:        end if
26:    end if
27: end function
```

Algorithm	Time complexity	$\log(avg(count))$	$\log(avg(N))$
trial division	$O(\sqrt{n})$	7.27207	17.84485674
Pollard Rho	$O(\sqrt[4]{n})$	3.658396	17.84485674
Pollard p-1	$O(n^{0.3/\log n})$	2.609123	17.84485674
Fermat	$O(n)$	15.10109	17.84485674

Table 1: result of the experiment

and q are both strong primes, the time complexity of Pollard p - 1 to crash RSA will be very large, and the reliability and security of RSA is guarantee. However, this mechanism will also return a failure because in the formula 7 may not be factorized further, and the gcd(,n)=n.. In this case, this algorithm still returns a failure.

4 Result and Analysis

Here is the result of our experiment

To make the experiment closer to the reality, we construct p and q as strong prime numbers. From the table 1, we observe that the $\log(avg(count))$ of Pollard p - 1 is more than 0.3/0.25 times than the Pollard rho algorithm. This is because that the digit length of p and q is the same and both $p-1$ and $q-1$ has large prime factors which are more than $n^{0.3}$. In this situation, the time complexity

of Pollard p - 1 is the same as trial division. And utilize strong prime numbers to execute RSA algorithm will increase the security of RAS against Pollard p - 1 algorithm.

5 Conclusion

By studying, analyzing, and implementing the four prime factorization algorithm to break RSA algorithm, we conclude that the Pollard rho algorithm is the most efficient algorithm.

References

[1] R. A. Mollin, "A brief history of factoring and primality testing bc (before computers)," *Mathematics magazine*, vol. 75(1), pp. 18–29, 2002.

[2] J. E. Hardy, G. H. Littlewood, "Contributions to the theory of the riemann zeta-function and the theory of the distribution of primes," *Acta Mathematica*, vol. 4(1), p. 119–196, 1916.

[3] Maeher, "how much computing resource is required to brute force rsa.," 2012.

[4] C. Barnes, "Integer factorization algorithms," *Acta Mathematica*, 2004.

[5] W. LeVeque, *Fundamentals of Number Theory*. Dover Publications, 2014.

[6] H. Riesel, "Prime numbers and computer methods for factorization," *Springer Science Business Media*, vol. 126, 2012.

[7] S. D. Miller and R. Venkatesan, "Spectral analysis of pollard rho collisions.," *International Algorithmic Number Theory Symposium*, 2006.

[8] J. M. Pollard, "Theorems of factorization and primality testing," *Proceedings of the Cambridge Philosophical Society*, vol. 76(3), p. 521–528, 1974.

[9] J. D. Dixon, "Asymptotically fast factorization of integers," *Math. Comp.*, vol. 36(152), p. 255–260, 1981.