



# 公钥密码学数学基础第四次实验报告

周家熠、潘子豪、王煜涵、刘一童

2024 年 11 月 10 日

## 目录

<b>1 摘要</b>	<b>2</b>
<b>2 实验目的</b>	<b>3</b>
<b>3 实验原理</b>	<b>3</b>
<b>4 实验内容</b>	<b>3</b>
4.1 试除法 . . . . .	3
4.1.1 原理分析 . . . . .	3
4.1.2 实验代码 . . . . .	3
4.1.3 实验结果 (sagemath) . . . . .	4
4.1.4 算法评价 . . . . .	4
4.2 Pollard $\rho$ 算法 . . . . .	4
4.2.1 原理分析 . . . . .	4
4.2.2 实验代码 . . . . .	5
4.2.3 实验结果 (NTL) . . . . .	7
4.2.4 算法评价 . . . . .	8
4.3 Pollard $p - 1$ 算法 . . . . .	8
4.3.1 原理分析 . . . . .	8
4.3.2 实验代码 . . . . .	8
4.3.3 实验结果 (NTL) . . . . .	11
4.3.4 算法评价 . . . . .	12
4.4 其他算法 . . . . .	12
4.4.1 Fermat's factorization . . . . .	12
4.4.2 基于连分数的分解方法 . . . . .	12
<b>5 实验反思</b>	<b>12</b>

## 1 摘要

本次实验为公钥密码学数学基础实验第四次实验，由周家熠、潘子豪、王煜涵、刘一童小组完成。实验内容为用试除法、Pollard  $\rho$  算法、Pollard  $p - 1$  算法完成大整数的素分解问题。王煜涵完成试除法部分；潘子豪完成 Pollard  $\rho - 1$  算法；周家熠完成 Pollard  $\rho$  算法；刘一童（学号：202300460117）负责完成实验报告的撰写于 10 月 25 日完成实验，11 月 10 日撰写实验报告。

## 2 实验目的

- 理解并掌握整数分解的基本原理，掌握常用的整数分解方法，并且比较各方法的优缺点
- 编程实现至少 3 种整数分解方法，分析各方法的优缺点

## 3 实验原理

整数分解问题（算术基本定理的算法化）研究具有重要意义。RSA 是著名的公钥密码算法，1977 年由 Rivest, Shamir 和 Adleman 一起提出，在实际中有着广泛的应用。RSA 算法的安全性依赖于大整数分解困难，这使得整数分解问题成为现代密码学非常关注的问题之一。关于整数分解方法的综述可以参见 [1]。整数分解为给定正整数  $N$ ，求  $N$  的素因子，即求整除  $N$  的素数

## 4 实验内容

### 4.1 试除法

#### 4.1.1 原理分析

设  $N$  为一个正整数，试除法看成是用小于等于  $\sqrt{N}$  的每个素数去试除待分解的整数。如果找到一个数能够整除除尽，这个数就是待分解数的因子。试除法是最初等的整数分解算法，该方法思想简单，但能够快速分解出  $N$  中的小素因子。

#### 4.1.2 实验代码

```
1 from sage.all import *
2
3 def trial_division(N):
4     factors = []
5     limit = floor(sqrt(N))
6     primes = prime_range(limit + 1)
7     for p in primes:
8         while N % p == 0:
9             factors.append(p)
10            N //= p
11        if N == 1:
12            break
13        if N > 1:
14            factors.append(N)
15    return factors
16 N = 16095650737563753533
17 factors = trial_division(N)
18 print("N 的素因子分解结果:", factors)
```

### 4.1.3 实验结果 (sagemath)

N 的素因子分解结果: [4279209601]

N 的素因子分解结果: [100547, 115637]

N 的素因子分解结果: [2, 3, 11, 4919, 592358293]

N 的素因子分解结果: [2299484981, 6999676393]

图 1: 试除法实验结果

### 4.1.4 算法评价

时间复杂度分析: 试除法的时间复杂度是  $O(\sqrt{N})$  在实际运行中, 随着  $n$  的增大, 试除法的效率会变得非常低下, 因为它需要逐个尝试每个因子, 直到  $\sqrt{n}$ 。

应用与局限性: 优点是实现简单, 适合用于分解小整数或用于理解分解算法的基本原理。而其缺点是对大整数试除法的效率非常低下, 在实际应用中通常使用效率更高的算法如 Pollard  $\rho$  算法以及 Fermat's factorization 算法等。

## 4.2 Pollard $\rho$ 算法

### 4.2.1 原理分析

Pollard 于 1975 年提出著名的  $\rho$  算法. 该方法的基本思想是通过多项式迭代产生数列, 从中寻找整数  $x_1, x_2$  满足  $\gcd(x_1 - x_2, N)$  是  $N$  的一个非平凡因子。其步骤为:

a) 设  $p$  是  $N$  的一个素因子,  $p < \sqrt{N}$ 。找到两个整数  $x, x'$ , 满足  $x \equiv x' \pmod{p}$ , 则可通过求最大公因子  $\gcd(x - x', N)$  来分解  $N$ 。

b) 如果致使想随机选一个子集  $X$  来进行碰撞, 那么这个子集的大小大约为  $1.17\sqrt{p}$ , 此时碰撞概率为 50

c) 但我们事先不知道  $p$  是多少, 所以要计算  $\binom{|X|}{2} > p/2$  次 GCD。

事实上, 我们可以不那么随机。选取一个整系数多项式  $f(x)$ , 例如  $f(x) = x^2 + 1$ 。

随机选取  $x_1$ , 考虑序列  $x_1, x_2, \dots$ , 其中

$$x_k \equiv f(x_{k-1}) \pmod{N}$$

某一时刻必有

$$x_i \equiv x_j \pmod{p}$$

则

$$x_{i+1} \equiv f(x_i) \equiv f(x_j) \equiv x_{j+1} \pmod{p}$$

一般的,

$$x_{i+k} \equiv x_{j+k} \pmod{p}$$

更一般的, 令  $j - i = l$ , 只要  $j' \equiv i' \pmod{p}$ , 则

$$x_{i'} \equiv x_{j'} \pmod{p}$$

这个算法需要计算  $\binom{j}{2}$  次 GCD。

#### 4.2.2 实验代码

```
1 #include<iostream>
2 #include<list>
3 #include<NTL/ZZ.h>
4 using namespace std;
5 using namespace NTL;
6 void Pollard_(ZZ N, list<ZZ>& table)
7 {
8     ZZ x = ZZ(1);
9     ZZ y = (x * x + 1) % N;
10    while (true)
11    {
12        if (x == y)
13        {
14            break;
15        }
16        if (GCD(abs(x - y), N) > 1 && GCD(abs(x - y), N) < N)
17        {
18            table.push_back(GCD(abs(x - y), N));
19            Pollard_(N / GCD(abs(x - y), N), table);
20            return;
21        }
22        x = (x * x + 1) % N;
23        y = (y * y + 1) % N;
24        y = (y * y + 1) % N;
25    }
26    table.push_back(N);
27 }
28 int main()
29 {
```

```
30 ZZ N;
31 cin >> N;
32 list<ZZ> table;
33 Pollard_(N, table);
34 if (table.size() == 1)
35 {
36     cout << "该数可能是素数" << endl;
37 }
38 else
39 {
40     cout << "分解结果：";
41     for (list<ZZ>::iterator it = table.begin(); it != table.end(); it++)
42     {
43         cout << *it << " ";
44     }
45 }
46 }
```

### 4.2.3 实验结果 (NTL)



图 2: Pollard  $\rho$  算法结果



#### 4.2.4 算法评价

时间复杂度分析：从期望上来说  $j$  大概差不多是  $\sqrt{p}$  (根据生日碰撞)，又因为  $p < \sqrt{N}$ ，所以整个计算的复杂度为  $O(N^{1/4})$ 。

成功率分析：算法有可能会失败。即  $x_i \equiv x_j \pmod{p}$  会导致  $x_i \equiv x_j \pmod{N}$ 。这个概率差不多是  $p/N$ 。

### 4.3 Pollard $p-1$ 算法

#### 4.3.1 原理分析

1974 年, Pollard 基于费马小定理, 提出了  $p-1$  算法. 虽然该方法不是一个具有一般性的分解算法, 但是其思想却被应用到一些现代的分解算法中. 比如, 基于 Pollard  $p-1$  算法的思想, Lenstra 提出了椭圆曲线分解方法。

设  $p$  是  $n$  的一个素因子, 并且假定对任意素数幂  $q \mid (p-1)$ , 有

$$q \leq B$$

于是

$$(p-1) \mid B!$$

$$a \equiv 2^{B!} \pmod{n}$$

$$a \equiv 2^{B!} \equiv 2^{p-1} \equiv 1 \pmod{p}$$

此时  $p \mid \gcd(a-1, n)$ 。

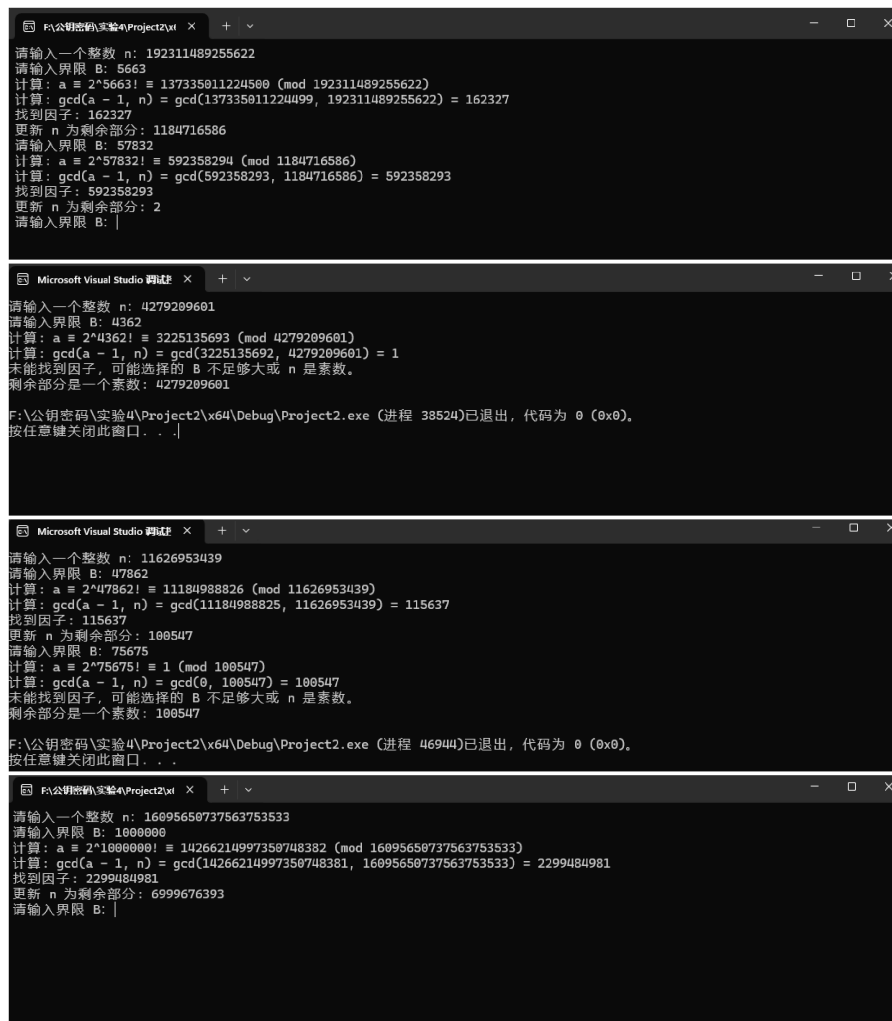
#### 4.3.2 实验代码

```
1 #include <NTL/ZZ.h>
2 #include <iostream>
3
4 using namespace std;
5 using namespace NTL;
6
7 // 计算阶乘 B!
8 ZZ factorial(long B) {
9     ZZ fact = ZZ(1);
10    for (long i = 2; i <= B; ++i) {
11        fact *= i;
12    }
13    return fact;
14 }
15
16 // Pollard p-1 算法实现
17 ZZ Pollard_p_minus_1(const ZZ& n, long B) {
18     ZZ a = ZZ(2); // 选择基础值 a = 2
19     ZZ B_factorial = factorial(B); // 计算 B!
```

```
20
21 // 计算  $a^{B!} \bmod n$ 
22 a = PowerMod(a, B_factorial, n);
23 cout << "计算: a  2^" << B << " !    " << a << "(mod " << n << ")" << endl;
24
25 // 计算 gcd(a - 1, n)
26 ZZ d = GCD(a - 1, n);
27 cout << "计算: gcd(a-1, n)= gcd(" << a-1 << ", " << n << ")=" << d << endl;
28
29 // 如果找到非平凡因子
30 if (d > 1 && d < n) {
31     return d;
32 }
33
34 return ZZ(0); // 如果没有找到因子, 返回 0
35 }
36
37 int main() {
38     ZZ n;
39     long B;
40
41     cout << "请输入一个整数 n: ";
42     cin >> n;
43
44     // 继续分解直到 n 为 1
45     while (n > 1) {
46         cout << "请输入界限 B: ";
47         cin >> B;
48
49         // 使用 Pollard p-1 算法寻找因子
50         ZZ factor = Pollard_p_minus_1(n, B);
51
52         if (factor != 0) {
53             cout << "找到因子: " << factor << endl;
54             n /= factor; // 更新 n 为剩余部分
55             cout << "更新 n 为剩余部分: " << n << endl;
56         }
57         else {
58             cout << "未能找到因子, 可能选择的 B 不够大或 n 是素数。" << endl;
59             break; // 如果没有找到因子, 则结束分解
60         }
61     }
```

```
62
63     if (n > 1) {
64         cout << "剩余部分是一个素数：" << n << endl;
65     }
66
67     return 0;
68 }
```

## 4.3.3 实验结果 (NTL)



```
F:\公钥密码\实验4\Project2\vt x + v
请输入一个整数 n: 192311489255622
请输入界限 B: 5663
计算:  $a \equiv 2^{5663} \pmod{192311489255622}$ 
计算:  $\gcd(a-1, n) = \gcd(137335911224580, 192311489255622) = 162327$ 
找到因子: 162327
更新 n 为剩余部分: 1184716586
请输入界限 B: 57832
计算:  $a \equiv 2^{57832} \pmod{1184716586}$ 
计算:  $\gcd(a-1, n) = \gcd(592358294, 1184716586) = 592358293$ 
找到因子: 592358293
更新 n 为剩余部分: 2
请输入界限 B: |

Microsoft Visual Studio 调试 x + v
请输入一个整数 n: 4279209601
请输入界限 B: 4362
计算:  $a \equiv 2^{4362} \pmod{4279209601}$ 
计算:  $\gcd(a-1, n) = \gcd(3225135693, 4279209601) = 1$ 
未能找到因子, 可能选择的 B 不够大或 n 是素数。
剩余部分是一个素数: 4279209601

F:\公钥密码\实验4\Project2\x64\Debug\Project2.exe (进程 38524)已退出, 代码为 0 (0x0)。
按任意键关闭此窗口。 . . |

Microsoft Visual Studio 调试 x + v
请输入一个整数 n: 11626953439
请输入界限 B: 47862
计算:  $a \equiv 2^{47862} \pmod{11626953439}$ 
计算:  $\gcd(a-1, n) = \gcd(11184988826, 11626953439) = 115637$ 
找到因子: 115637
更新 n 为剩余部分: 100547
请输入界限 B: 75675
计算:  $a \equiv 2^{75675} \pmod{100547}$ 
计算:  $\gcd(a-1, n) = \gcd(0, 100547) = 100547$ 
未能找到因子, 可能选择的 B 不够大或 n 是素数。
剩余部分是一个素数: 100547

F:\公钥密码\实验4\Project2\x64\Debug\Project2.exe (进程 46944)已退出, 代码为 0 (0x0)。
按任意键关闭此窗口。 . . |

F:\公钥密码\实验4\Project2\vt x + v
请输入一个整数 n: 16095650737563753533
请输入界限 B: 1000000
计算:  $a \equiv 2^{1000000} \pmod{16095650737563753533}$ 
计算:  $\gcd(a-1, n) = \gcd(14266214997350748382, 16095650737563753533) = 2299484981$ 
找到因子: 2299484981
更新 n 为剩余部分: 6999676393
请输入界限 B: |
```

图 3: 实验结果

#### 4.3.4 算法评价

Pollard  $p-1$  算法的缺陷:

$n$  有一个素因子  $p$  使得  $p-1$  只有小的素因子

$n = pq$ , 其中  $p = 2q' + 1, q = 2q' + 1$ , 且  $p', q'$  均为大素数。

#### 4.4 其他算法

##### 4.4.1 Fermat's factorization

费马曾提出一个基于二次同余的想法, 即如果可以找到正整数  $s, t$  满足  $s^2 \equiv t^2 \pmod{N}$ , 则  $\gcd(s+t, N)$  可能是  $N$  的一个非平凡因子。为了提高搜索满足条件的整数  $s, t$  的效率, 人们引入了分解基的概念。一个分解基是不超过某个上界  $B$  的素数集合, 若一个正整数的素因子均在该分解基中, 则称为  $B$ -光滑的。现代分解算法大都基于分解基的方法, 比如连分数方法, 二次筛法和数域筛法等。数域筛法是目前分解大整数最有效的算法。

##### 4.4.2 基于连分数的分解方法

该方法特别用于攻击 RSA 低解密指数的情况, 即 Weiner 攻击, 下面是对于该方法的分析:

RSA 算法的安全性基于大整数分解的困难性, 特别是  $N$  的两个大素因子 (通常是两个质数  $p$  和  $q$ ) 的分解。Weiner 提出的连分数方法用于攻击低解密指数  $d$  的 RSA 算法, 适用于某些特定条件下的加密系统, 其中解密指数  $d$  非常小。

Weiner 攻击的关键条件时: 模数  $N$  是由两个质数  $p, q$  的乘积  $N = p * q$ ;  $p, q$  的二进制长度接近使得  $p < q < 2p$ ; 解密指数  $d$  满足  $d < \frac{1}{3}N^{1/4}$ 。

攻击方法概述: Weiner 方法的基本步骤是利用连分数展开将分数  $\frac{k}{d}$  作为近似, 满足  $\frac{k}{d} \approx \frac{e}{N}$ , 其中  $e$  是公钥指数,  $N$  是模数,  $d$  是解密指数

展开  $\frac{e}{N}$  为连分数并找到其逐步逼近; 检查这些逼近是否能够满足  $ed \equiv 1 \pmod{\phi(N)}$ ; 如果找到了符合的近似  $\frac{k}{d}$ , 则  $d$  是解密指数。

该方法利用了连分数的性质: 对于一个给定的有理数, 连分数展开能给出它的最佳有理近似。Weiner 证明了, 当满足上述条件时, RSA 的解密指数  $d$  的连分数逼近是其中之一。

## 5 实验反思

在实验给定的 7 个测试用例中, 后三个大整数的分解较为困难, 其中, 试除法和 Pollard  $p-1$  算法均未成功分解后三个大整数 (时间消耗超过 24 小时), 而 Pollard 算法经过算法的优化, 成功在半小时之内跑出了第 5 个大整数, 但对于最后两个大整数仍无能为力 (时间消耗超过 12 小时)。其中既有客观原因, 如待分解的数过大和设备算力有限, 也有主观原因, 如代码优化不到位等。