



# 公钥密码学数学基础第五次实验报告

周家熠、潘子豪、王煜涵、刘一童

2024 年 11 月 21 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>
3.1 穷举法 . . . . .	3
3.2 Pohlig-Hellman 算法 . . . . .	4
3.3 小步大步法 (Baby-step Giant-step,BSGS) . . . . .	4
3.4 Pollard $\rho, \lambda$ 算法 . . . . .	4
3.5 指标求解算法 . . . . .	5
3.6 数域筛法 (Number Field Sieve, NFS) . . . . .	5
3.7 Shor 算法 . . . . .	5
<b>4 实验内容</b>	<b>5</b>
4.1 Pohlig-Hellman 算法 . . . . .	5
4.1.1 实验代码 . . . . .	5
4.1.2 实验结果 (sagemath) . . . . .	7
4.2 小步大步法 (Baby-step Giant-step,BSGS) . . . . .	7
4.2.1 实验代码 . . . . .	7
4.2.2 实验结果 (NTL) . . . . .	10
4.3 Pollard $\rho, \lambda$ 算法 . . . . .	10
4.3.1 实验代码 . . . . .	12
4.3.2 实验结果 (NTL) . . . . .	14
<b>5 实验反思</b>	<b>15</b>

## 1 摘要

本次实验为公钥密码学数学基础实验第五次实验，由周家熠、潘子豪、王煜涵、刘一童小组完成。实验内容为编程实现至少三种离散对数求解方法，分析各种方法的优缺点并对比同等规模的整数分解和离散对数问题求解所用的时间；其中潘子豪采用小步大步法；周家熠采用 Pollard  $\rho, \lambda$  算法；王煜涵为 Pohlig-Hellman 算法；刘一童（学号：202300460117）负责完成实验报告的撰写。于 11 月 17 日完成实验，11 月 20 日撰写实验报告。本实验报告借助于 overleaf 的在线 LaTeX 平台完成。

## 2 实验目的

- 理解并掌握离散对数求解的基本原理，掌握常用的离散对数求解方法，并且比较各方法的优缺点。
- 编程实现至少 3 种离散对数的求解方法，分析各方法的优缺点。
- 对比同等规模的整数分解和离散对数问题求解所用的时间。

## 3 实验原理

Diffie-Hellman 密钥交换协议是第一个公钥密码算法，1976 年由 Diffie 和 Hellman 一起提出，在 SSH, TLS 等安全套件中有着广泛的应用。DH 算法的安全性依赖于离散对数问题的困难性，这使得离散对数问题成为现代密码学非常关注的问题之一。关于离散对数问题求解的综述可以参考[裴定一, 祝跃飞. 算法数论 (第二版), 科学出版社, 2002]。模素数  $p$  的离散对数问题为给定正素数  $p$ , 元素  $g$  (通常为模  $p$  的原根) 和目标元素  $h$ , 求解  $h$  相对于  $g$  模  $p$  的离散对数。即求整数  $a$ , 使得  $g^a \equiv h \pmod{p}$ 。

在这里我们会介绍几种常见的求解离散对数的方法以及基本原理，稍后我们会在实验内容中选取三个进行编程实现。

### 3.1 穷举法

穷举法是最初等的离散对数求解算法，该方法依次计算  $g^x \equiv p$ , 并检验每个值是否等于  $h$ , 穷举法的思想简单，但复杂度高。现给出一穷举法的合理但非完全可行性的 python 代码 (面对较大的数不可行)

```
1 def discrete_log(g, h, p):
2     for x in range(p):
3         if pow(g, x, p) == h: # 计算  $g^x \bmod p$ 
4             return x
5     return None # 无解
6
7 # 测试
8 g = 3
9 h = 13
10 p = 17
11 result = discrete_log(g, h, p)
12 if result is not None:
13     print(f"The discrete logarithm is x = {result}")
14 else:
15     print("No solution found")
```

容易得到，穷举法的时间复杂度为  $O(p)$  因为其需要测试所有可能的  $x$  的值，并不是最优解法。

### 3.2 Pohlig-Hellman 算法

Pohlig-Hellman 算法利用孙子定理给出的已知  $p-1$  分解时的求解算法, 假设  $p-1 = \prod_{i=1}^n l_i^{e_i}$ 。Pohlig-Hellman 算法的主要思想是对  $i$  从 1 到  $n$ , 计算

$$a_i = k_0 + k_1 l_1 + \dots + k_{e_i} l_i^{e_i-1}$$

其中,  $a_i \equiv a \pmod{l_i^{e_i}}$ , 再用孙子定理计算  $a$ 。

Pohlig-Hellman 算法其实是将该问题分解成了许多子问题, 先将待进行离散对数分解的  $p$  进行质因子分解, 然后在子群模  $q_i^{e_i}$  内求解  $x_i$ , 然后利用递归方法求解每一个  $x$ , 最后使用孙子定理将所有的解组合为一个模  $n$  的解。所以, 该方法的时间复杂度要分布求解, 对于质因数分解来说为  $O(\sqrt{n})$ , 但实际通常非常小, 然后对于每一个子问题。其复杂度为  $O(q_i^{e_i})$  或更低, 然后对其求和, 有

$$= O\left(\sum_{i=1}^k q_i^{e_i}\right)$$

由于  $n$  为因数分解的结果, 当对于质因子  $q_i$  很小时该算法非常高效。

### 3.3 小步大步法 (Baby-step Giant-step, BSGS)

Shancke 利用了时间/空间权衡的思想, 提出了 BSGS 法, 将暴力枚举的复杂度从  $O(p)$  降低到  $O(\sqrt{p})$ , 效率较穷举法有所改进, 但占用内存较大。

它适用于离散对数问题的一般情况:

$$g^x \equiv h \pmod{p}$$

其中  $g$  是群  $G$  的生成元,  $h$  是目标值,  $p$  是模数, 求  $x$ 。

利用离散对数的性质, 将  $x$  分为两部分:

$$x = m * k + j, \quad m = \lceil \sqrt{p} \rceil$$

于是:

$$h \equiv g^x \pmod{p} \implies h * g^{-j} \equiv g^{m*k} \pmod{p}$$

这里  $m$  为步长的大小, 算法分两步完成:

- 小步阶段 (Baby-step): 枚举所有可能的  $g^j \pmod{p}$ , 存入一个哈希表, 其中  $j \in [0, m-1]$ 。
- 大步阶段 (Giant-step): 枚举  $h * g^{-m*k} \pmod{p}$ , 其中  $k \in [0, m-1]$ , 并在哈希表中查找匹配项。

如果找到匹配项  $g^j$ , 则有:

$$x = m * k + j.$$

相较于暴力穷举法来说, 其时间复杂度为  $O(\sqrt{p})$  显著降低了计算时间, 并且其通用性强, 适用于任何离散对数问题。但其缺点也非常显著, 其空间需求较高, 对于  $p$  非常大 (如 256 位素数) 时可能难以实现。

### 3.4 Pollard $\rho, \lambda$ 算法

Pollard 提出著名的  $\rho, \lambda$  (也称 kangaroo) 算法, 基本思想是通过概率方法寻找整数对  $x_1, x_2, y_1, y_2$  满足  $g^{x_1} t^{x_2} \equiv g^{y_1} t^{y_2} \pmod{p}$ , 从而求解离散对数。两种算法都利用了伪随机游走。

### 3.5 指标求解算法

针对有限域 (有限域即元素有限的集合, 其中可以任意进行加减乘, 以及分母非零的除运算) 离散对数问题, 研究者设计了指标求解算法, 其中利用了分解基、光滑性的概念和线性代数求解方程. 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. 该算法依赖这样的命题:

如果  $x^a y^b z^c = 1$ , 那么  $a \log_g x + b \log_g y + c \log_g z = \log_g 1 = 0$ . 因此, 如果我们能找到  $x_i$ , 且  $L_i = \log_g x_i$ ,  $h = x_1^{a_1} \dots x_n^{a_n}$ , 那么有:

$$\log_g h = a_1 L_1 + \dots + a_n L_n$$

指标求解法的时间复杂度为  $O(p)(\ ) + O(1)(\ )$ .

### 3.6 数域筛法 (Number Field Sieve, NFS)

数域筛法是基于指标求解算法框架设计的算法, 目前求模  $p$  离散对数渐近时间复杂度最低的经典计算算法。

数域筛法的复杂度是已知离散对数和大整数分解算法中最低的一个, 为亚指数级:

$$L_N\left[\frac{1}{3}, \left(\frac{64}{9}\right)^{\frac{1}{3}}\right]$$

其中:

$$L_N[s, c] = \exp((c + o(1)) * (\log N)^s * (\log \log N)^{1-s})$$

对于  $N$  的大小在 100 到 200 之间时, NFS 通常优于其他算法, 对于更小的规模的问题, NFS 算法可能不如简单的试除法或分圆筛法。

### 3.7 Shor 算法

Shor 算法, 是 Shor 提出的在量子计算机上面运作的算法, 也可高效地求解离散对数问题。

## 4 实验内容

### 4.1 Pohlig-Hellman 算法

#### 4.1.1 实验代码

```
1 #!/usr/bin/env python
2 # coding: utf-8
3 # In[4]:
4 from sage.all import *
5 import math
6 # 计算素因数分解
7 def fenjie(n):
8     # 创建元组存储因子
9     fs = []
10    # 检查偶数因子
```

```
11     count = 0
12     while n % 2 == 0:
13         n //= 2
14         count += 1
15     if count > 0:
16         fs.append((2, count))
17     # 检查奇数因子
18     f = 3 #从2之后的最小因子3开始
19     while f * f <= n:
20         count = 0
21         while n % f == 0:
22             n //= f
23             count += 1
24         if count > 0:
25             fs.append((f, count))
26         f += 2
27     if n > 1:
28         fs.append((n, 1))
29     return fs
30 # range(p)暴力搜索计算离散对数
31 def find(g, h, p):
32     for x in range(p):
33         if pow(g, x, p) == h: #g**x mod p == h
34             return x
35     return None
36 # 实现Pohlig-Hellman算法
37 def pohlig_hellman(g, h, p):
38     # 1. 获取群的阶p-1的素因数分解
39     fs = fenjie(p - 1)
40     # 存储每个素因子阶的离散对数解
41     s = []
42     # 2. 处理每个素因子阶
43     for q, i in fs:
44         q_i = q ** i
45         # 3. 计算当前素因子阶的问题: g**x == h (mod p), 但是阶为q_i
46         # 通过暴力搜索计算离散对数
47         s_i = find(g, h, p)
48         if s_i is None:
49             raise ValueError(f"子问题无解, 检查输入参数")
50         s.append(s_i)
51     # 4. 使用孙子定理合并解
52     x = crt(s, [q ** i for q, i in fs])
```

```
53     return x
54 p = 17
55 g = 3
56 h = 10
57 # Pohlig-Hellman 算法
58 result = pohlig_hellman(g, h, p)
59 print("离散对数结果:", result)
60 # In[ ]:
```

#### 4.1.2 实验结果 (sagemath)

```
p - 635906251
g - 2
h - 465843315

# Pohlig-Hellman 算法
result - pohlig_hellman(g, h, p)
print("离散对数结果:", result)
```

离散对数结果: 257172463

```
p - 1563714751
g - 7
h - 1779858

# Pohlig-Hellman 算法
result - pohlig_hellman(g, h, p)
print("离散对数结果:", result)
```

离散对数结果: 341881522

图 1: 实验结果

## 4.2 小步大步法 (Baby-step Giant-step, BSGS)

### 4.2.1 实验代码

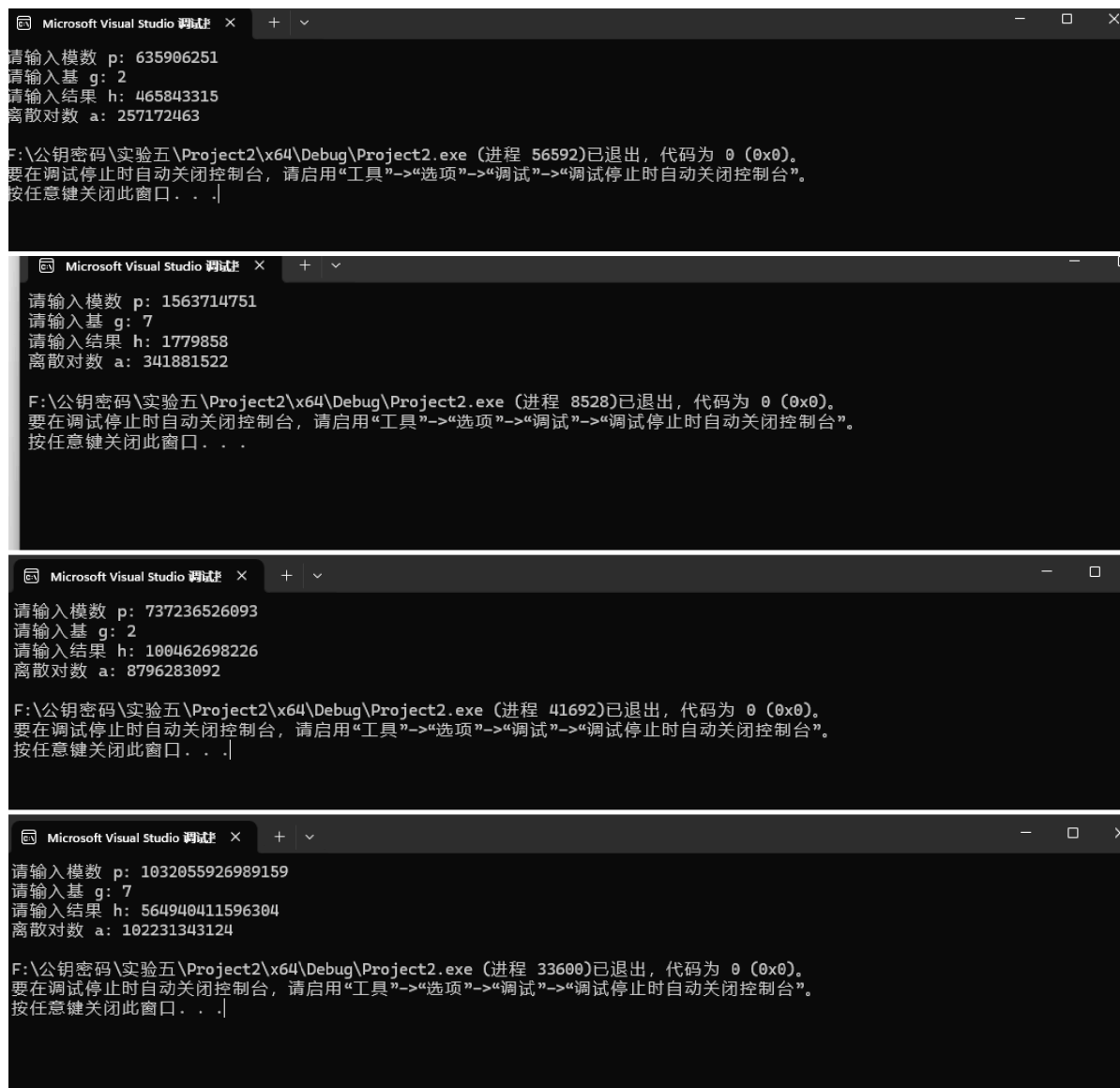
```
1 #include <NTL/ZZ.h>
2 #include <vector>
3 #include <cmath>
4 #include <iostream>
5
6 using namespace NTL;
7 using namespace std;
8
```



```
9  ZZ BabyStepGiantStep(ZZ g, ZZ h, ZZ p) {
10     // 首先计算  $m = \sqrt{N}$  , 其中  $N$  为模数  $p$  的阶
11     ZZ m = SqrRoot(p) + 1;
12
13     // 使用 vector 存储小步表
14     vector<pair<ZZ, ZZ>> baby_steps;
15
16     // 然后计算所有小步值:  $g^{a0} \pmod p$ 
17     ZZ cur = ZZ(1); // 初始值  $g^0 = 1$ 
18     for (ZZ a0 = ZZ(0); a0 < m; a0++) {
19         baby_steps.push_back(make_pair(cur, a0)); // 存储  $(g^{a0}, a0)$ 
20         cur = (cur * g) % p; // 递推计算  $g^{(a0+1)}$ 
21     }
22
23     // 计算  $g^{(-m)}$  (模逆元的  $m$  次幂)
24     ZZ g_m = PowerMod(g, -m, p);
25
26     // 进行大步搜索:  $h * (g^{(-m)})^{a1} \pmod p$ 
27     cur = h; // 初始值为  $h$ 
28     for (ZZ a1 = ZZ(0); a1 < m; a1++) {
29         // 线性查找匹配的小步值
30         for (const auto& entry : baby_steps) {
31             if (entry.first == cur) {
32                 // 找到匹配的小步值
33                 ZZ a0 = entry.second;
34                 return a0 + a1 * m; //  $a = a0 + a1 * m$ 
35             }
36         }
37         cur = (cur * g_m) % p; // 更新为  $h * (g^{(-m)})^{(a1+1)}$ 
38     }
39
40     // 如果未找到解
41     return ZZ(-1);
42 }
43
44 int main() {
45
46     ZZ g, h, p;
47     cout << "请输入模数 p: ";
48     cin >> p;
49     cout << "请输入基 g: ";
50     cin >> g;
```

```
51     cout << "请输入结果 h: ";
52     cin >> h;
53     ZZ result = BabyStepGiantStep(g, h, p);
54     if (result != -1) {
55         cout << "离散对数 a: " << result << endl;
56     }
57     else {
58         cout << "未找到解。" << endl;
59     }
60
61     return 0;
62 }
```

### 4.2.2 实验结果 (NTL)



```

Microsoft Visual Studio 调试
请输入模数 p: 635906251
请输入基 g: 2
请输入结果 h: 465843315
离散对数 a: 257172463

F:\公钥密码\实验五\Project2\x64\Debug\Project2.exe (进程 56592)已退出, 代码为 0 (0x0)。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .

Microsoft Visual Studio 调试
请输入模数 p: 1563714751
请输入基 g: 7
请输入结果 h: 1779858
离散对数 a: 341881522

F:\公钥密码\实验五\Project2\x64\Debug\Project2.exe (进程 8528)已退出, 代码为 0 (0x0)。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .

Microsoft Visual Studio 调试
请输入模数 p: 737236526093
请输入基 g: 2
请输入结果 h: 100462698226
离散对数 a: 8796283092

F:\公钥密码\实验五\Project2\x64\Debug\Project2.exe (进程 41692)已退出, 代码为 0 (0x0)。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .

Microsoft Visual Studio 调试
请输入模数 p: 1032055926989159
请输入基 g: 7
请输入结果 h: 564940411596304
离散对数 a: 102231343124

F:\公钥密码\实验五\Project2\x64\Debug\Project2.exe (进程 33600)已退出, 代码为 0 (0x0)。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .

```

图 2: 实验结果

### 4.3 Pollard $\rho, \lambda$ 算法

问题: 给定  $p, g, h$ , 求  $\gamma$  满足:

$$g^\gamma \equiv h \pmod{p}$$

算法思想: 想办法找到整数  $a, b, A, B$ , 使得

$$g^a h^b \equiv g^A h^B \pmod{p}$$

整理得

$$g^{a-A} \equiv h^{B-b} \pmod{p}$$

由于

$$g^\gamma \equiv h \pmod{p}$$

得

$$g^{a-A} \equiv g^{(B-b)\gamma} \pmod{p}$$

即

$$(B-b)\gamma \equiv a-A \pmod{p-1}$$

利用 Euclid 算法可以轻易的算出  $\gamma$ 。

算法具体实现：

设模  $p$  的一组既约剩余系为群  $G$ ，对  $G$  进行划分：

$$G = S_0 \cup S_1 \cup S_2, S_i \cap S_j = \phi$$

定义映射  $f: G \rightarrow G$ :

$$f(x) \equiv \begin{cases} gx & \pmod{p}, x \in S_0 \\ hx & \pmod{p}, x \in S_1 \\ x^2 & \pmod{p}, x \in S_2 \end{cases}$$

定义映射  $m: G * Z \rightarrow Z$ :

$$m(x, k) \equiv \begin{cases} k+1 & \pmod{p-1}, x \in S_0 \\ k & \pmod{p-1}, x \in S_1 \\ 2k & \pmod{p-1}, x \in S_2 \end{cases}$$

定义映射  $n: G * Z \rightarrow Z$ :

$$n(x, k) \equiv \begin{cases} k & \pmod{p-1}, x \in S_0 \\ k+1 & \pmod{p-1}, x \in S_1 \\ 2k & \pmod{p-1}, x \in S_2 \end{cases}$$

初始化  $a_i, a_j, b_i, b_j = 0; x_i, x_j = 1$ ; ( $i$  和  $j$  是快慢指针,  $i$  每移动一步,  $j$  移动两步)

执行下面的循环:

```

1 Loop:
2
3 xi=f(xi) ai=m(xi, ai) bi=n(xi, bi)
4
5 xj=f(xj) aj=m(xj, aj) bj=n(xj, bj)
6
7 xj=f(xj) aj=m(xj, aj) bj=n(xj, bj)
8
9 if xi==xj break
    
```

使用 Euclid 算法解方程

$$(b_i - b_j)\gamma \equiv (a_j - a_i) \pmod{p-1}$$

将方程所有解  $\gamma$  依次带入

$$g^\gamma \equiv h \pmod{p}$$

中验证, 找到满足该式子的唯一  $\gamma$ , 即为所求。

## 4.3.1 实验代码

```
1 #include<iostream>
2 #include<NTL/ZZ.h>
3 #include<NTL/ZZ_p.h>
4 using namespace std;
5 using namespace NTL;
6 ZZ Pollard_rho(ZZ p, ZZ g, ZZ h)
7 {
8     ZZ ai, aj, bi, bj = ZZ(0);
9     ZZ xi = ZZ(1); ZZ xj = ZZ(1);
10    while (true)
11    {
12        switch (xi % 3)
13        {
14        case 0: xi = xi * g % p; ai = (ai + 1) % (p - 1);
15        break;
16        case 1: xi = xi * h % p; bi = (bi + 1) % (p - 1);
17        break;
18        case 2: xi = xi * xi % p; ai = ai * 2 % (p - 1); bi = bi * 2 % (p - 1);
19        break;
20        }
21        switch (xj % 3)
22        {
23        case 0: xj = xj * g % p; aj = (aj + 1) % (p - 1);
24        break;
25        case 1: xj = xj * h % p; bj = (bj + 1) % (p - 1);
26        break;
27        case 2: xj = xj * xj % p; aj = aj * 2 % (p - 1); bj = bj * 2 % (p - 1);
28        break;
29        }
30        switch (xj % 3)
31        {
32        case 0: xj = xj * g % p; aj = (aj + 1) % (p - 1);
33        break;
34        case 1: xj = xj * h % p; bj = (bj + 1) % (p - 1);
35        break;
36        case 2: xj = xj * xj % p; aj = aj * 2 % (p - 1); bj = bj * 2 % (p - 1);
37        break;
38        }
39        if (xi == xj)break;
40    }
41    ZZ a = (bi - bj) % (p - 1);
```

```
42     if (a == 0)
43     {
44         cout << "failed" << endl;
45         return ZZ(0);
46     }
47     ZZ b = (aj - ai) % (p - 1);
48     ZZ gcd = GCD(a, p - 1);
49     ZZ r = b / gcd * InvMod(a / gcd, (p - 1) / gcd) % ((p - 1) / gcd);
50     ZZ_p g_mod, h_mod;
51     conv(g_mod, g);
52     conv(h_mod, h);
53     for (int i = 0; i < gcd; i++)
54     {
55         if (power(g_mod, r + i * (p - 1) / gcd) == h_mod)
56             return (r + i * (p - 1) / gcd);
57     }
58 }
59 int main()
60 {
61     ZZ p, g, h;
62     cout << "请依次输入p g h" << endl;
63     cin >> p >> g >> h;
64     ZZ_p::init(p);
65     cout << Pollard_rho(p, g, h) << endl;
66     return 0;
67 }
```

### 4.3.2 实验结果 (NTL)



图 3: 实验结果

## 5 实验反思

在实验给定的 5 个测试用例中，只有周家熠负责部分的方法经过编程以及代码优化成功完成 5 个测试用例，潘子豪以及王煜涵负责的部分方法只可完成其中的 2 到 3 个测试用例，第 4 以及第 5 个测试用例因为方法缺陷以及代码优化不良好导致运行时间过长导致计算机报错。