《现代密码学》实验报告

实验名称：DLP 计算	**实验时间**：2025 年 12 月 13 日
学生姓名：谢润文	**学号**：23336262
学生班级：23 计科	**成绩评定**：

一、实验目的

通过 Pollard ρ 算法计算离散对数问题（Discrete Logarithm Problem，DLP），进而实现位数比较大的 DLP（p=192bit，n=64bit），理解 Floyd 判环、Pollard ρ 算法原理，利用蒙哥马利规约实现平方乘算法（模幂）的加速，提高大数运算的应用及处理能力，进一步为后面的数字签名实验奠定基础。

二、实验内容

用 C/C++实现 Z_p 中的 DLP 计算。输入由 p（大素数），n，alpha，beta 构成，n 是 alpha 在 Z_p 中的阶数（规定 n 为素数方便计算，否则要利用线性同余方程进行求解）。计算 x=log（alpha，beta），且 x<n。

最终目标是要计算一个比较大的离散对数：

p=376890152190840720115769119802971197287608764797082459653 3，

n=9993115456385501509，

alpha=3107382411142271813235322646657672922264748410711464860476 ** (2 * 2 * 23 * 8783 * 2419781956425763 * 192888768642311611 * 学号（这里为 23336262））

beta=212055387361243984541985869645154093639584450549686713371 1。

三、实验原理

0. 离散对数引言

实数中对数的定义 $\log_b a=x$，即 $b^x=a$。在任何群中可以为正整数 k 定义一个幂数 $b^k=a$，而 $\log_b a$ 定义为使得 $b^k=a$ 成立的正整数 k。

1976 年，Diffie 和 Hellman 实现了一种体现公钥密码体制思想，即基于离散对数问题的、在不安全的通道上进行密钥形成与交换的新技术，开启了公钥密码学的大门。遗憾的是，他们三人只是提出了一种关于公钥密码体制与数字签名的思想，而没有真正实现。直到 1977 年 RSA 的提出才是真正意义上的第一个公钥加密。

1. 离散对数的主流计算方法

1.1 BSGS 算法

对于实验给出的 DLP 的问题，暴力枚举每一个 x，x<n 显然是 O（n）的。有没有更快的做法呢？一种很经典的算法是大步小步法（Baby-Step Giant-Step，BSGS）。

以求解 $a^x=b$（mod p）为例。首先令 x=A⌈ sqrt(n)⌉ -B。其中 A，B 均小于⌈ sqrt(n)⌉ 。则有 $a^{A⌈ sqrt(n)⌉}=ba^B$（mod p）。已知 a,b，可以先算出等式右边的 ba^B的所有取值，用 hash 表存下来。然后逐一计算 $a^{A⌈ sqrt(n)⌉}$，寻找是否有与之相等的 ba^B。

时间复杂度为 O(sqrt(n))，空间复杂度为 O(sqrt(n))。

1.2 Pollard ρ 算法

BSGS 的复杂度相对可以接受，美中不足就是空间复杂度太高。比如 $n=2^{64}$时，需要 2^{32}条匹配（预计算）记录，这样庞大的开销显然是不能接受的。一个比较好的方法是约翰·波拉德在 1978 年提出的 ρ 算法。以 $a^x=b$（mod p）为例，找到 x1、x2、y1、y2 使得 $a^{x1}b^{y1}=a^{x2}b^{y2}$。整理可知 $x=(x1-x2)(y2-y1)^{-1}$(mod n)。使用 Floyd 判圈算法在数列 $t_i=a^{xi}b^{yi}$中寻找一个环，可以证明环的期望步数为 sqrt（π*n/8）。具体步骤如下：

```
输入 a: a 是 G 的生成元, b: G 的一个元素
输出：整数 x 使得 a^x = b, 或者失败
 初始化 a0 ← 0, b0 ← 0, x0 ← 1 ∈ G,
 i ← 1
 loop
     xi ← f(xi-1),
     ai ← g(xi-1, ai-1),
     bi ← h(xi-1, bi-1)
     x2i ← f(f(x2i-2)),
     a2i ← g(f(x2i-2), g(x2i-2, a2i-2)),
     b2i ← h(f(x2i-2), h(x2i-2, b2i-2))
     if xi = x2i then
         r ← bi - b2i
         if r = 0 return failure
         x ← r^-1(a2i - ai) mod p
         return x
     else
         i ← i+1
     end if
  end loop
```

整体时间复杂度仍为 O(sqrt(n))，但是空间复杂度降至了 O（1）。

1.3 Pohlig-Hellman 算法

有没有更快的算法呢？事实上，假如知道 n 的一种分解，$n=p_1p_2...p_n$，$p_1<p_2<...<p_n$，那么可以利用中国剩余定理和原根的性质来求解 DLP。具体地说，对于每一个质因子 pi，构建关于 x 的方程。假如 n 容易分解的话，那么称之为光滑群。在光滑群中求解 DLP 可以证明复杂度是 O（p_n）的，其中 p_n是最大的素因子。

1.4 Index calculus

还有没有更快的算法呢？Western 和 Miller 在 1968 年提出了指标演算法（Index Calculus，IC）的思想，后人进一步应用、优化。大体思想如下：

1. 找一个值 S，将小于 S 的所有素数构成基组 A。

2. 计算 ind_bp_i，具体计算通过解线性方程组完成。随机选取 t，希望 b^t 写作 $p_1^{e1}p_2^{e2}...p_n^{en}$，其中 1<=t<=p-1，如果不能被完全分解，则丢弃分解结果。

3. 直到分解了有效的 k 个分解结果，才中止。

4. 基于上述，求解任意数的离散对数，再随机选 t 计算 $y'=yb^t$。判定 y’是否能在 A 上分解。如果不能，重新选择 t。

在【1】中给出了关于 IC 的时间复杂度为亚指数级的证明。

2. 本次编程采用的算法

2.1 算法排查

首先确定算法：由于空间有限（只有 32MB，哪怕在本地跑也没有特别大的空间，对于 n=64 的情况仍然不能很好地存储），故不能采用 BSGS。再注意到 n 为素数，Pohlig-Hellman 算法退化为 pollard ρ 算法。可以考虑的就是 ρ 算法的变形（此处不考虑 Index calculus 因为一是随机选取可能导致常数较大，二是解线性方程组的开销不小。）

2.2 算法优化

一个很普通的优化是将 mod 3 运算单独算，将 2*a、2*b 的运算改为 a+a、b+b 的运算，取模改为减。这样能够降低常数。

mod 3 代码片段如下：

```
int mod3() const {
        uint64_t sum=0;
        for(int i=0;i<len;++i) sum+=d[i];
        return sum%3;
    }
```

即通过 uint64_t 的模运算替代大整数除法所需的开销。

此外，还有一个容易想到的优化：所有关于 mod p 的模乘、模幂，用蒙哥马利规约实现而不是大数乘法再模。

如此，实验表明可以快速地提升 4 倍左右的性能（如图 1 所示，其中最下面一次是没有优化的，中间那次是经过乘改加，模改减的，最上面那次是基于蒙哥马利规约的。）

2695	23336262	2025-12-04 09:16:40	C++	13965	1715.758 ms / 512 KB / Accept 7/7
2693	23336262	2025-12-04 08:37:50	C++	13147	3725.633 ms / 512 KB / Accept 7/7
2674	23336262	2025-12-03 15:27:04	C++	12890	6580.587 ms / 512 KB / Accept 7/7

图 1 OJ 上的提交-DLP-初始

2.3 算法改进

2.3.1 CIOS

一种比普通蒙哥马利规约更快而且被很好应用（比如，在【2】中给出了进一步的 CPU 架构）的算法是粗粒度集成操作数扫描算法（Coarsely Integrated Operand Scanning，CIOS）。普通的 Montgomery 算法要一次性计算出 M、N、T=A*B。CIOS 的思想是基于拆分，在 b 进制下，利用存储位宽进行合并，对 Montgomery 算法的 for 循环进行展开。

算法伪代码如下：

```
CIOS (a,b,n,n')
for i = 0 to s − 1 loop
    C := 0
    for j = 0 to s − 1 loop
        (C,S) := t[j] + a[j] * b[i] + C
        t[j] := S
    end loop
    (C,S) := t[s] + C
    t[s] := S
    t[s + 1] := C
    C := 0
    m := t[0] * n'[0] mod W
    (C,S) := t[0] + m * n[0]
    for j = 1 to s − 1 loop
        (C,S) := t[j] + m * n[j] + C
        t[j − 1] := S
    end loop
    (C,S) := t[s] + C
    t[s − 1] := S
    t[s] := t[s + 1] + C
end loop
```

图 2 CIOS 算法伪代码

应用在大数运算上，可以看到大约提供了 140ms 的加速（id=2822 对比 id=2576，中间那次是尝试用 karatsuba 乘法但是位移运算开销过大导致的）。

图 3 OJ 上的提交-大数运算

2.3.2 20-adding 与 tag tracing

2008 年三位韩国学者提出了 20-adding 和 tag tracing 算法（发表在 2008 年的亚密会上），用以改进和加速 pollard ρ 算法。其中 20-adding walk 是 r-adding walk 中 r 取 20 的情况，实验证明其性能比较好。基本思想是将群 G 划分为 r 个大致相等的子集，并为每个子集预定义一个随机元素 $M_s=g^{ms}h^{ns}$，通过迭代计算。而 tag 算法大致是在 r-adding walk 的基础上，不完全计算每个节点，而是利用预计算的乘法器乘积表和部分 tag 来推断下一个索引。

具体参考【3】中算法的实现。

在原论文中给出了 20-adding 和 tag tracing 的时间复杂度对比，可以看到即使是 20-adding 相比于原始的 pollard ρ 算法，也有大约 1.25 倍的加速。

Table 2. Full running time comparison of tag tracing and 20-adding walks

	$\|q\| = 35$	$\|q\| = 40$	$\|q\| = 45$	$\|q\| = 50$	$\|q\| = 55$
Pollard rho	1.103 sec	6.272 sec	38.738 sec	203.138 sec	1185.578 sec
20-adding	0.940 sec	5.174 sec	29.653 sec	159.977 sec	959.027 sec
tag tracing	0.093 sec	0.441 sec	2.634 sec	13.481 sec	80.785 sec
Pollard rho/tag tracing	11.89	14.24	14.70	15.07	14.68
20-adding/tag tracing	10.14	11.75	11.26	11.87	11.87

图 4 原论文中的算法时间复杂度对比

在实际编程中采取了 20-adding 的方式，因为 tag 的预计算处理逻辑较为复杂（~~实在是没看懂~~）。

代码片段如下：

```
int r_branches=20,bit=0;
vector<bign> M_mont(r_branches);
vector<bign> u_val(r_branches);
```

```
        vector<bign> v_val(r_branches);
        for(int i=0;i<r_branches;++i){
            u_val[i].len=n.len;
            v_val[i].len=n.len;
            for(int k=0;k<n.len;++k){
                u_val[i].d[k]=0;
                for(bit=0;bit<64;bit+=15)
u_val[i].d[k]^=((uint64_t)rand()<<bit);
                v_val[i].d[k]=0;
                for(bit=0;bit<64;                                    bit+=15)
v_val[i].d[k]^=((uint64_t)rand()<<bit);
            }
            u_val[i].clean();
            v_val[i].clean();
            u_val[i]=u_val[i]%n;
            v_val[i]=v_val[i]%n;
            bign t1=mont_pow(alpha,u_val[i],p);
            bign t2=mont_pow(beta,v_val[i],p);
            bign t1_mont=mont_mul(t1,R2_mod_p,p,p_prime);
            bign t2_mont=mont_mul(t2,R2_mod_p,p,p_prime);
            M_mont[i]=mont_mul(t1_mont,t2_mont,p,p_prime);
        }
        long long attempt=0;
        while(1){
            bign a,b,x_mont;
            a.len=n.len;
            b.len=n.len;
            for(int k=0;k<n.len;++k){
```

```
                a.d[k]=0;
                                                    for(bit=0;bit<64;bit+=15)
a.d[k]^=((uint64_t)rand()<<bit);
                b.d[k]=0;
                                                    for(bit=0;bit<64;bit+=15)
b.d[k]^=((uint64_t)rand()<<bit);
            }
            a.clean();
            b.clean();
            a%=n;
            b%=n;
            bign t1=mont_pow(alpha,a,p);
            bign t2=mont_pow(beta,b,p);
            bign t1_mont=mont_mul(t1,R2_mod_p,p,p_prime);
            bign t2_mont=mont_mul(t2,R2_mod_p,p,p_prime);
            x_mont=mont_mul(t1_mont,t2_mont,p,p_prime);
            bign x_prime_mont=x_mont;
            bign a_=a;
            bign b_=b;
            auto f=[&](bign& x_m,bign& a_val,bign& b_val){
                int tag=x_m.d[0]%r_branches;
                x_m=mont_mul(x_m,M_mont[tag],p,p_prime);
                a_val=a_val+u_val[tag];
                if(a_val>=n) a_val=a_val-n;
                b_val=b_val+v_val[tag];
                if(b_val>=n) b_val=b_val-n;
            };
```

实际跑结果如图 5，其中 id=2774 是 20-adding，id=2777 是 4-adding。

图 5 OJ 上的评测-DLP-adding 算法

结合之前的 pollard ρ 算法表现（约 1700ms），可以看到结果大致符合预期，说明 20-adding 的加速是有效的。（理论上如果结合 tag tracing 算法，预计时间在 100ms 左右）

2.3.3 Pollard's Kangaroo Algorithm

本来采用这个算法只是为了测试对比的，没想到跑出来的结果却远远地比预期好。袋鼠算法是一种基于“占坑”的算法，有点类似显著点（可区分点）。在【4】中作者给出了算法成功的概率，以及更丰富的例子。

算法流程如下：

1. 选取跳跃步长跳跃表生成：生成 32 个（由于 n<=64bit，取 bit(sqrt n)个即 32 个）跳跃步长 Si，步长为 2^i（上限为 n），并预计算对应的乘数 $Mi=a^{Si}(\mathrm{mod}\ p)$。

2. Tame Kangaroo：从 $x_T=n$ 开始，进行随机跳跃。当到达“显著点”（Distinguished Point，不妨定义为低 16 位为 0 的点）时，记录位置(y,x_T)到哈希表中。

3. Wild Kangaroo：从 $x_W=0$（即值为 b）开始，进行相同的随机跳跃。当到达显著点时，检查哈希表。

4. 如果 Wild Kangaroo 遇到了 Tame Kangaroo 留下的记录，则计算 $x=x_T-x_W$ 并返回

在【5】中认为时间复杂度为 O（sqrt(n)）。可能是不用多次蒙哥马利规约计算幂的原因，反而常数小了，比 20-adding 的效率还高。运用上述袋鼠算法在 OJ 上的评测结果如下：

图 6 OJ 上评测的 Pollard's Kangaroo 算法

然后是一点小巧思。Wild Kangaroo 的随机跳跃的时间比较长，我们希望能够降低时间，又增加成功率，那怎么办呢。实际上，我们可以增加显著点的数量（通过 mask 动态调整），利用空间换时间，二分循环次数以逼近最佳成功率，实验表明，循环次数的下限大概在 1.25*sqrt(n)。

代码片段如下：

```
    uint64_t mask=0xFF;
   uint64_t target=m_val/512;
```

```
       while(mask<target) mask=(mask<<1)|1;
       bign xT=n;
       bign yT=mont_pow(a,n,p);
       bign yT_mont;
       mont_mul(yT,R2_mod_p,p,p_prime,yT_mont);
       int limit=std::max((uint64_t)(1.25*m_val),(uint64_t)1114514);
        for(int i=0;i<limit;++i){
            int j=yT_mont.d[0]%k;
            xT=xT+S[j];
            mont_mul(yT_mont,M[j],p,p_prime,yT_mont);
            if((yT_mont.d[0]&mask)==0){
                table[yT_mont]=xT;
                      if(table.size()>std::max((uint64_t)4000,m_val/512))
break;
            }
        }
        bign xW=0;
        bign yW_mont;
        mont_mul(b,R2_mod_p,p,p_prime,yW_mont);
        for(int i=0;i<limit;++i){
            int j=yW_mont.d[0]%k;
            xW=xW+S[j];
            mont_mul(yW_mont,M[j],p,p_prime,yW_mont);
            if((yW_mont.d[0]&mask)==0){
                if(table.count(yW_mont)){
                    bign target_x=table[yW_mont];
                    if(target_x>xW) return (target_x-xW)%n;
                }
```

```
            }
        }
```

2843	23336262	2025-12-11 12:28:40	C++	17440	439.734 ms / 1.75 MB / Accept 7/7
2842	23336262	2025-12-11 11:51:09	C++	17440	511.211 ms / 767.99 KB / Accept 7/7
2841	23336262	2025-12-11 11:46:40	C++	17440	846.961 ms / 512 KB / Accept 6/7
2840	23336262	2025-12-11 11:42:47	C++	17439	655.337 ms / 611.99 KB / Accept 7/7

图 7 二分法试探下限（主要是最后一个点）

四、实验步骤（源代码）

此处主要展示 20-adding 下的 pollard ρ 算法。

代码如下：

```
    #include<iostream>
    #include<vector>
    #include<string>
    #include<algorithm>
    #include<cstring>
    #include<fstream>
    #include<cstdint>
    #include<fcntl.h>
    #include<cstdlib>
    #include<ctime>
    using namespace std;
    const int MAXN=130;
    const uint64_t BASE10_19=10000000000000000000ull;
    inline uint8_t addcarry_u64(uint8_t c_in,uint64_t a,uint64_t b,uint64_t
*out){
        unsigned __int128 res=(unsigned __int128)a+b+c_in;
        *out=(uint64_t)res;
        return (uint8_t)(res>>64);
    }
    inline   uint8_t   subborrow_u64(uint8_t   b_in,uint64_t   a,uint64_t
b,uint64_t *out) {
        unsigned __int128 res=(unsigned __int128)a-b-b_in;
        *out=(uint64_t)res;
        return (uint8_t)(res>>64)&1;
    }
    inline uint64_t mulx_u64(uint64_t a,uint64_t b,uint64_t *hi){
        unsigned __int128 res=(unsigned __int128)a*b;
        *hi=(uint64_t)(res>>64);
```

```
    return (uint64_t)res;
}

struct bign{
    uint64_t d[MAXN];
    int len;
    bool sign;
    bign(){
        memset(d,0,sizeof(d));
        len=1;
        sign=false;
    }
    bign(int v){
        memset(d,0,sizeof(d));
        if(v<0){
            sign=true;
            v=-v;
        }
        else sign=false;
        d[0]=(uint64_t)v;
        len=1;
    }
    bign(long long v){
        memset(d,0,sizeof(d));
        if(v<0){
            sign=true;
            v=-v;
        }
        else sign=false;
        d[0]=(uint64_t)v;
        len=1;
    }
    bign(unsigned long long v){
        memset(d,0,sizeof(d));
        d[0]=v;
        len=1;
        sign=false;
    }
    void clean(){
        while(len>1 && d[len-1]==0) len--;
        if(len==1 && d[0]==0) sign=false;
    }
    int abs_cmp(const bign& b) const {
        if(len!=b.len) return len<b.len?-1:1;
```

```
            for(int i=len-1;i>= 0;--i){
                if(d[i]!=b.d[i]) return d[i]<b.d[i]?-1:1;
            }
            return 0;
        }
        bool operator<(const bign& b) const {
            if(sign!=b.sign) return sign;
            if(sign) return abs_cmp(b)>0;
            return abs_cmp(b)<0;
        }
        bool operator>(const bign& b) const { return b < *this; }
        bool operator<=(const bign& b) const { return !(*this > b); }
        bool operator>=(const bign& b) const { return !(*this < b); }
        bool operator==(const bign& b) const { return sign==b.sign &&
abs_cmp(b)==0; }
        bool operator!=(const bign& b) const { return !(*this == b); }

        static bign abs_add(const bign& a,const bign& b){
            bign res;
            res.len=max(a.len,b.len);
            uint8_t carry=0;
            for(int i=0; i<res.len;++i){
                uint64_t ai=(i<a.len)?a.d[i]:0ull;
                uint64_t bi=(i<b.len)?b.d[i]:0ull;
                carry=addcarry_u64(carry,ai,bi,&res.d[i]);
            }
            if(carry){
                if(res.len<MAXN) res.d[res.len++]=carry;
            }
            return res;
        }

        static bign abs_sub(const bign& a,const bign& b){
            bign res;
            res.len=a.len;
            uint8_t borrow=0;
            for (int i=0;i<res.len;++i){
                uint64_t ai=(i<a.len)?a.d[i]:0ull;
                uint64_t bi=(i<b.len)?b.d[i]:0ull;
                borrow=subborrow_u64(borrow,ai,bi,&res.d[i]);
            }
            res.clean();
            return res;
        }
```

```
       static bign mul(const bign& a,const bign& b){
           if((a.len==1 && a.d[0]==0) || (b.len==1 && b.d[0]==0)) return
bign(0);
           bign res;
           res.len=a.len+b.len;
           if(res.len>MAXN) res.len=MAXN;
           memset(res.d,0,sizeof(uint64_t)*res.len);
           for(int i=0;i<a.len;++i){
               uint64_t r_c=0,hi,lo,v;
               for(int j=0;j<b.len;++j){
                   if(i+j>=MAXN) break;
                   lo=mulx_u64(a.d[i],b.d[j],&hi);
                   v=res.d[i + j];
                   uint8_t k1=addcarry_u64(0,v,lo,&v);
                   uint8_t k2=addcarry_u64(0,v,r_c,&res.d[i+j]);
                   r_c=hi+k1+k2;
               }
               if(i+b.len<MAXN) res.d[i+b.len]=r_c;
           }
           res.clean();
           res.sign=a.sign^b.sign;
           return res;
       }

       static pair<bign, bign> div_mod(const bign& u, const bign& v){
           if(v.len==1 && v.d[0]==0) return {bign(0),bign(0)};
           if(u.abs_cmp(v)<0) return {bign(0),u};
           if(v.len==1){
               bign q;
               q.len=u.len;
               uint64_t rem=0;
               for(int i=u.len-1;i>=0;--i){
                   unsigned            __int128            cur=((unsigned
__int128)rem<<64)|u.d[i];
                   q.d[i]=(uint64_t)(cur/v.d[0]);
                   rem=(uint64_t)(cur%v.d[0]);
               }
               q.clean();
               q.sign=u.sign^v.sign;
               bign r((unsigned long long)rem);
               r.sign=u.sign;
               return {q,r};
           }
```

```
            int s=__builtin_clzll(v.d[v.len-1]);
            bign vn=v,un=u;
            if(s>0){
                for(int                                    i=vn.len-1;i>0;--i)
vn.d[i]=(vn.d[i]<<s)|(vn.d[i-1]>>(64-s));
                vn.d[0]<<=s;
                if(un.len<MAXN) un.d[un.len]=0;
                for(int            i=un.len;i>0;--i)            if(i<MAXN)
un.d[i]=(un.d[i]<<s)|(un.d[i-1]>>(64-s));
                un.d[0]<<=s;
                if(un.len<MAXN && un.d[un.len]) un.len++;
            }
            if(un.len==u.len && un.len<MAXN) un.d[un.len++]=0;
            bign q;
            q.len=un.len-vn.len;
            for(int j=un.len-vn.len-1;j>=0;--j){
                unsigned              __int128              num=((unsigned
__int128)un.d[j+vn.len]<<64)|un.d[j+vn.len-1];
                unsigned __int128 q_=num/vn.d[vn.len-1];
                unsigned __int128 r_=num%vn.d[vn.len-1];
                while(1){
                    if(q_>=((unsigned
__int128)1<<64)||(q_*vn.d[vn.len-2]>((r_<<64)|un.d[j+vn.len-2]))){
                        q_--;
                        r_+=vn.d[vn.len-1];
                        if(r_>=((unsigned __int128)1<<64)) break;
                    }
                    else break;
                }
                unsigned __int128 k=0;
                for(int i=0;i<vn.len;++i){
                    unsigned __int128 p=(unsigned __int128)q_*vn.d[i]+k;
                    uint64_t sub=(uint64_t)p;
                    k=p>>64;
                    if(un.d[i+j]<sub) k++;
                    un.d[i+j]-=sub;
                }
                if(un.d[j+vn.len]<k){
                    un.d[j+vn.len]-=k;
                    q_--;
                    uint8_t c=0;
                    for(int                                  i=0;i<vn.len;++i)
c=addcarry_u64(c,un.d[i+j],vn.d[i],&un.d[i+j]);
                    un.d[j+vn.len]+=c;
```

```
                }
                else un.d[j+vn.len]-=k;
                q.d[j]=(uint64_t)q_;
            }
            q.clean();
            q.sign=u.sign^v.sign;
            if(s>0)               for(int               i=0;i<vn.len;++i)
un.d[i]=(un.d[i]>>s)|(un.d[i+1]<<(64-s));
            un.len=vn.len;
            un.clean();
            un.sign=u.sign;
            return {q,un};
        }

        bign operator+(const bign& b) const {
            if(sign==b.sign) {
                bign res=abs_add(*this,b);
                res.sign=sign;
                return res;
            }
            else{
                if(abs_cmp(b)>=0){
                    bign res=abs_sub(*this,b);
                    res.sign=sign;
                    return res;
                }
                else{
                    bign res=abs_sub(b,*this);
                    res.sign=b.sign;
                    return res;
                }
            }
        }
        bign operator-(const bign& b) const{
            bign t=b;
            t.sign=!t.sign;
            return *this+t;
        }
        bign operator*(const bign& b) const{ return mul(*this, b); }
        bign  operator/(const  bign&  b)  const{  return  div_mod(*this,
b).first; }
        bign operator%(const bign& b) const{
            bign res=div_mod(*this,b).second;
            if(res.sign) res=res+b;
```

```
        return res;
    }
    bign operator+=(const bign& b) { *this = *this + b; return *this; }
    bign operator-=(const bign& b) { *this = *this - b; return *this; }
    bign operator*=(const bign& b) { *this = *this * b; return *this; }
    bign operator/=(const bign& b) { *this = *this / b; return *this; }
    bign operator%=(const bign& b) { *this = *this % b; return *this; }

    static bign from_string(const uint8_t* str){
        bign res(0);
        int i=0;
        bool neg=false;
        if(str[0]=='-'){ neg = true; i++; }
        for(;str[i]; ++i){
            bign ten(10);
            bign digit(str[i]-'0');
            res=res*ten+digit;
        }
        res.sign=neg;
        return res;
    }
    void print(){
        if(len==1 && d[0]==0){
            putchar('0');
            putchar('\n');
            return;
        }
        if(sign) putchar('-');
        bign tmp=*this;
        tmp.sign=false;
        bign base((unsigned long long)BASE10_19);
        vector<uint64_t> c;
        while(tmp.len>1 || tmp.d[0]>=BASE10_19){
            pair<bign,bign> qr=div_mod(tmp,base);
            c.push_back(qr.second.d[0]);
            tmp=qr.first;
        }
        c.push_back(tmp.d[0]);
        uint64_t val=c.back();
        if(val==0) putchar('0');
        else{
            char buf[32];
            int idx=0;
            while(val){
```

```
                buf[idx++]=val%10+'0';
                val/=10;
            }
            while(idx--) putchar(buf[idx]);
        }
        for(int i=c.size()-2;i>=0;--i){
            val=c[i];
            char buf[32];
            for(int j=0;j<19;++j){
                buf[j]=val%10+'0';
                val/=10;
            }
            for(int j=18;j>=0;--j) putchar(buf[j]);
        }
        putchar('\n');
    }
    int mod3() const {
        uint64_t sum = 0;
        for(int i=0; i<len; ++i){
            sum += d[i];
        }
        return sum % 3;
    }
};

void mont_mul(const bign& a, const bign& b, const bign& n, uint64_t
n_prime, bign& res){
    uint64_t t[2*MAXN+2]={0};
    int k=n.len;
    for(int i=0;i<k;++i){
        uint64_t ai=(i<a.len)?a.d[i]:0;
        uint64_t carry=0;
        for(int j=0;j<k;++j){
            uint64_t bj=(j<b.len)?b.d[j]:0;
            uint64_t hi,lo;
            lo=mulx_u64(ai,bj,&hi);
            uint64_t old_t=t[i+j];
            uint8_t c1=addcarry_u64(0,old_t,lo,&t[i+j]);
            uint8_t c2=addcarry_u64(0,t[i+j],carry,&t[i+j]);
            carry=hi+c1+c2;
        }
        uint64_t old_t=t[i+k];
        uint8_t c1=addcarry_u64(0,old_t,carry,&t[i+k]);
        t[i+k+1]+=c1;
```

```
            uint64_t m=t[i]*n_prime;
            carry=0;
            for(int j=0;j<k;++j){
                uint64_t nj=n.d[j];
                uint64_t hi,lo;
                lo=mulx_u64(m,nj,&hi);
                old_t=t[i+j];
                c1=addcarry_u64(0,old_t,lo,&t[i+j]);
                uint8_t c2=addcarry_u64(0,t[i+j],carry,&t[i+j]);
                carry=hi+c1+c2;
            }
            old_t=t[i+k];
            c1=addcarry_u64(0,old_t,carry,&t[i+k]);
            t[i+k+1]+=c1;
        }
        res.len=k;
        for(int i=0;i<k;++i) res.d[i]=t[i+k];
        bool fl=(t[2*k]!=0);
        if(!fl){
            for(int i=k-1;i>=0;--i){
                if(res.d[i]>n.d[i]) { fl = true; break; }
                if(res.d[i]<n.d[i]) { break; }
                if(i==0) fl=true;
            }
        }
        if(fl){
            uint8_t borrow=0;
            for(int i=0;i<k;++i){
                uint64_t ni=n.d[i];
                borrow=subborrow_u64(borrow,res.d[i],ni,&res.d[i]);
            }
        }
        res.clean();
    }

    bign mont_mul(const bign& a, const bign& b, const bign& n, uint64_t
n_prime){
        bign res;
        mont_mul(a,b,n,n_prime,res);
        return res;
    }

    bign zero(0),one(1);
    bign mont_pow(bign a,bign b,bign n){
```

```
        if(n.len==1 && n.d[0]==1) return zero;
        if(b.len==1 && b.d[0]==0) return one;
        uint64_t x = 1;
        uint64_t n0 = n.d[0];
        for (int i = 0; i < 6; ++i) x = x * (2 - n0 * x);
        uint64_t n_prime = -x;
        bign R;
        R.len=n.len+1;
        R.d[n.len]=1;
        bign R_mod_n=R%n;
        bign R2_mod_n=(R_mod_n*R_mod_n)%n;

        bign a_mont;
        mont_mul(a, R2_mod_n, n, n_prime, a_mont);

        bign ans = R_mod_n;
        int bit_len=b.len*64-__builtin_clzll(b.d[b.len-1]);
        for(int i=bit_len-1;i>=0;--i){
            mont_mul(ans, ans, n, n_prime, ans);
            if((b.d[i/64]>>(i%64))&1) mont_mul(ans, a_mont, n, n_prime,
ans);
        }
        bign result;
        mont_mul(ans, one, n, n_prime, result);
        return result;
    }
    bign ex_gcd(bign a, bign b, bign& x, bign& y){
        bign r0=a,r1=b;
        bign s0(1),s1(0),t0(0),t1(1);
        while(r1.len>1 || r1.d[0]!=0){
            pair<bign,bign> qr=bign::div_mod(r0, r1);
            bign q=qr.first,r2=qr.second;
            bign s2=s0-q*s1;
            bign t2=t0-q*t1;
            r0=r1;
            r1=r2;
            s0=s1;
            s1=s2;
            t0=t1;
            t1=t2;
        }
        x=s0;
        y=t0;
        return r0;
```

```
    }
    bign rho(bign n,bign alpha,bign beta,bign p){
        uint64_t x=1,n0=p.d[0];
        for(int i=0;i<6;++i) x=x*(2-n0*x);
        uint64_t p_prime=-x;
        bign R;
        R.len=p.len+1;
        R.d[p.len]=1;
        bign R_mod_p=R%p;
        bign R2_mod_p=(R_mod_p*R_mod_p)%p;
        srand(time(0));
        int r_branches=20,bit=0;
        vector<bign> M_mont(r_branches);
        vector<bign> u_val(r_branches);
        vector<bign> v_val(r_branches);
        for(int i=0;i<r_branches;++i){
            u_val[i].len=n.len;
            v_val[i].len=n.len;
            for(int k=0;k<n.len;++k){
                u_val[i].d[k]=0;
                for(bit=0;bit<64;bit+=15)
u_val[i].d[k]^=((uint64_t)rand()<<bit);
                v_val[i].d[k]=0;
                for(bit=0;bit<64;bit+=15)
v_val[i].d[k]^=((uint64_t)rand()<<bit);
            }
            u_val[i].clean();
            v_val[i].clean();
            u_val[i]=u_val[i]%n;
            v_val[i]=v_val[i]%n;
            bign t1=mont_pow(alpha,u_val[i],p);
            bign t2=mont_pow(beta,v_val[i],p);
            bign t1_mont=mont_mul(t1,R2_mod_p,p,p_prime);
            bign t2_mont=mont_mul(t2,R2_mod_p,p,p_prime);
            M_mont[i]=mont_mul(t1_mont,t2_mont,p,p_prime);
        }
        long long attempt=0;
        while(1){
            bign a,b,x_mont;
            a.len=n.len;
            b.len=n.len;
            for(int k=0;k<n.len;++k){
                a.d[k]=0;
                                                    for(bit=0;bit<64;bit+=15)
```

```
a.d[k]^=((uint64_t)rand()<<bit);
                b.d[k]=0;
                                                    for(bit=0;bit<64;bit+=15)
b.d[k]^=((uint64_t)rand()<<bit);
            }
            a.clean();
            b.clean();
            a%=n;
            b%=n;
            bign t1=mont_pow(alpha,a,p);
            bign t2=mont_pow(beta,b,p);
            bign t1_mont=mont_mul(t1,R2_mod_p,p,p_prime);
            bign t2_mont=mont_mul(t2,R2_mod_p,p,p_prime);
            x_mont=mont_mul(t1_mont,t2_mont,p,p_prime);
            bign x_prime_mont=x_mont;
            bign a_=a;
            bign b_=b;
            auto f=[&](bign& x_m,bign& a_val,bign& b_val){
                int tag=x_m.d[0]%r_branches;
                x_m=mont_mul(x_m,M_mont[tag],p,p_prime);
                a_val+=u_val[tag];
                if(a_val>=n) a_val=a_val-n;
                b_val+=v_val[tag];
                if(b_val>=n) b_val=b_val-n;
            };
            f(x_mont,a,b);
            x_prime_mont=x_mont;
            a_=a;
            b_=b;
            f(x_prime_mont,a_,b_);
            while(x_mont!=x_prime_mont){
                f(x_mont,a,b);
                f(x_prime_mont,a_,b_);
                f(x_prime_mont,a_,b_);
            }
            bign delta_b=b_-b;
            if(delta_b.sign) delta_b+=n;
            bign delta_a=a-a_;
            if(delta_a.sign) delta_a+=n;
            bign binv,y;
            bign g=ex_gcd(delta_b,n,binv,y);
            if(g.len==1 && g.d[0]==1){
                if(binv.sign) binv=binv+n;
                bign ans=(delta_a*binv)%n;
```

```
            return ans;
        }
        else attempt++;
    }
}
uint8_t buf[8192];
int main(){
    bign p,n,a,b,res;
    scanf("%s",buf);
    p=bign::from_string(buf);
    scanf("%s",buf);
    n=bign::from_string(buf);
    scanf("%s",buf);
    a=bign::from_string(buf);
    scanf("%s",buf);
    b=bign::from_string(buf);
    res=rho(n,a,b,p);
    res.print();
    return 0;
}
```

五、实验结果

这里以第三组样例数据为例，运行 rho 算法，可以看到正确地输出了结果。

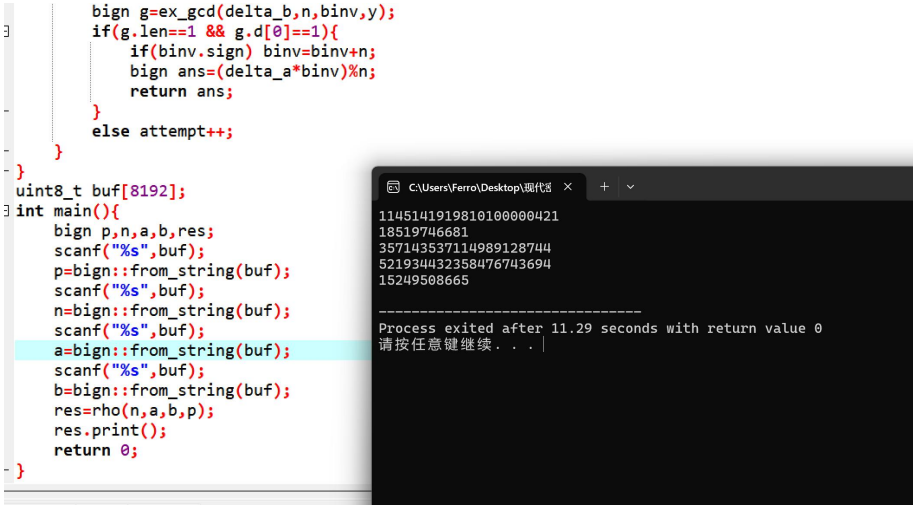

图 8 样例#3 本地测试

前面实验原理部分也证明了在 OJ 上跑没有问题。可以验证 rho 算法的设计与实现是正确的。下面进行大数部分（即开头提到的 p=192bit 素数，n=64bit 素数的情况）的计算。

把 main 函数改为如下，用 result.txt 保存计算结果：

```cpp
int main(){
    bign p,n,a,b,res,a_1,a_2;
    p=bign::from_string("37689015219084072011576911980297119728760876479708245965333");
    n=bign::from_string("9993115456385501509");
    a_1=bign::from_string("3107382411142271813235322646657672922264748410711464860476");
    a_2=bign::from_string("192888768642311611");
    long long a_20=2419781956425763;
    freopen("result.txt", "w", stdout);
    a_2*=a_20;
    a_2*=23336262;
    a_2*=4*23*8783;
    a=mont_pow(a_1,a_2,p);
    a.print();
    b=bign::from_string("212055387361243984541985869645154093639584450549686713371 1");
    res=rho(n,a,b,p);
    cout<<"find:";
    res.print();
    return 0;
}
```

图 9 跑大数 DLP 的本地测试

有一个小细节就是直接拿 a 去跑的话会增加很多运算量，可以先做 a%p 的规约。

跑了大约 6h（程序运行完的时候手贱点了两下退出了，没有看到精确的计时），得到结果如下：

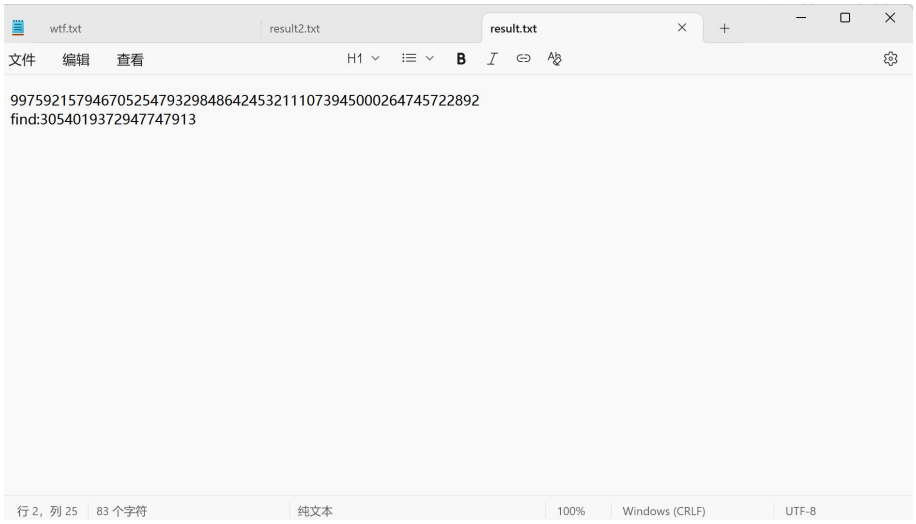

图 10 运行结果（找到的解）

下面进行验证，利用 Argon2 Hash Generator, Validator & Verifier 验证找到的解的正确性。

在文档中给出了 Argon2 的 hash 值：

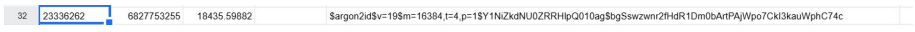

图 11 助教计算得到的标准 hash 值

将解（3054019372944747913）与这个 hash 值试图匹配：

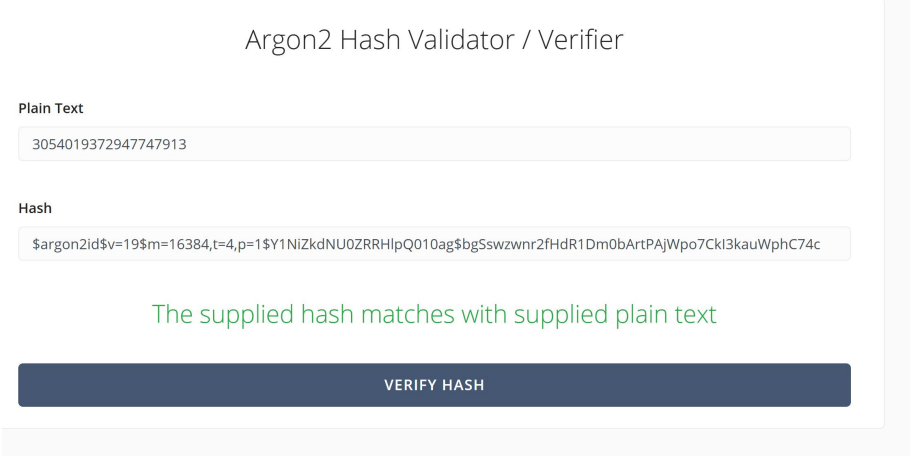

图 12 验证结果

可以看到它们是匹配的，说明没有算错。

另外，这个 6h 左右的速度还是比较慢，既然 p 的分解已知，考虑用 Pohlig-Hellman 算法的思想进行优化，可以从直接模 p 变为模 p-1 的因子，减小模运算开销。main 函数如下：

```
int main(){
    bign p,n,a,b,res,a_1,a_2;
    p=bign::from_string((const uint8_t*)"376890152190840720115769119802971197287608764797082459653 3
    n=bign::from_string((const uint8_t*)"9993115456385501509");
    a_1=bign::from_string((const uint8_t*)"3107382411142271813235322646657672922264748410711464860 4
    a_2=bign::from_string((const uint8_t*)"192888768642311611");
    long long a_20=2419781956425763;
    freopen("result2.txt", "w", stdout);
    a_2*=a_20;
    a_2*=23336262;
    a_2*=4*23*8783;
    a=mont_pow(a_1,a_2,p);
    a.print();
    b=bign::from_string((const uint8_t*)"212055387361243984541985869645154093639584450549686713371 1

    res = pohlig_hellman(a, b, p);

    cout<<"find:";
    res.print();
    return 0;
}
```

图 13 优化后的 main 函数

得到运行时间如下：

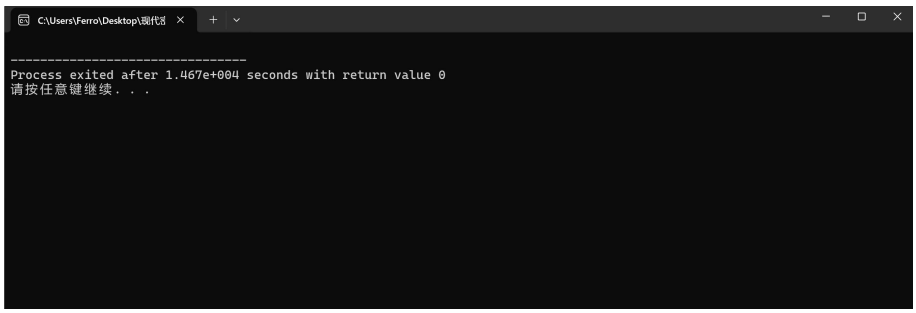

图 14 计时结果

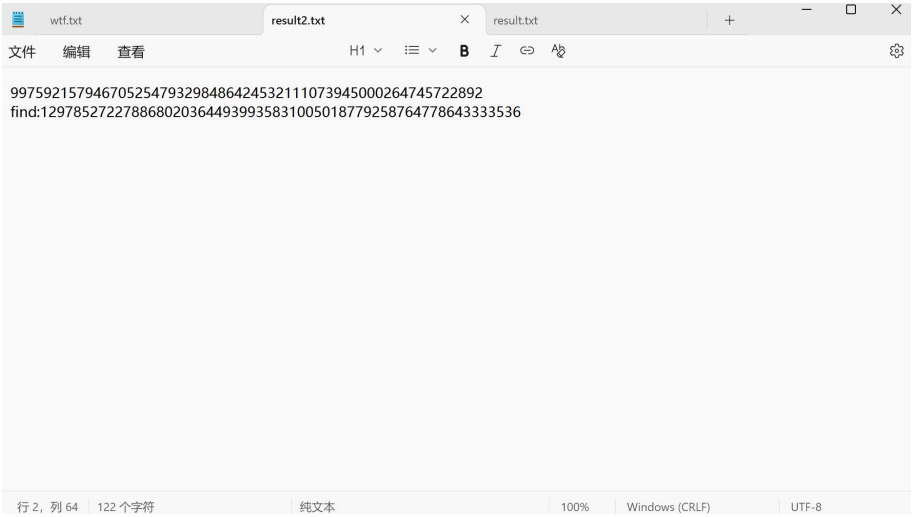

图 15 结果（找到的解）

哎呀好像忘了取模了，没关系，再验证一下是这两个解相等（本次实验有约束 x<n）就可以了：

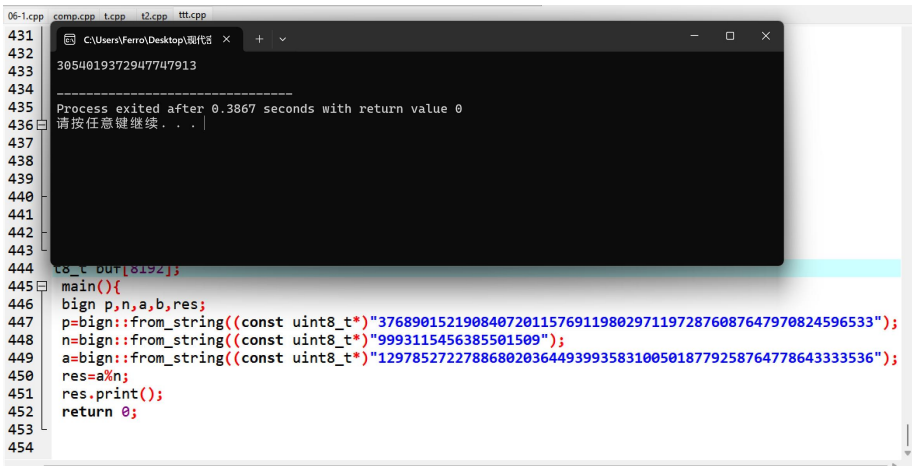

图 16 二次验证

至此完结撒花。

从图 14 可以看到跑了 $1.467*10^4$s 即 14670s（大约 4.08h），而参考用时为 18435s，大约快了一个钟头的运行时间。这说明优化起到了效果，整体运行时间还是可观的。

六、思考题

1. 还有什么加速求解这个大数 DLP 的方法？

一种方法是利用 CPU 并行。【6】中给出了利用分布式 Pollard rho 算法求解的方法。即找到显著点（可区分点），每个处理器独立地运行 Pollard rho 算法，但是每当它计算到一个特征点时，它就将这个点及对应的指数信息发送到一个中央服务器。中央服务器存储这

些特征点。当同一个特征点被两个不同的处理器发送，或者被同一个处理器两次发送时，说明检测到了一个循环，从而求解出离散对数。在本地机也许可以尝试多线程处理，但是比较复杂，所以没进行加速。

七、实验总结

本次离散对数实验要求计算一些比较小的 DLP 和一个大数（p=192bit，n=64bit）的 DLP。同样通过前置的大数运算包括模幂、通过 exgcd 的求逆等，考察了对 Z_p 上基本运算的理解和 n 阶循环群的性质，为下一次的 DSA 数字签名实验做铺垫。实验中通过 CIOS 对蒙哥马利规约的加速和 20-adding 等优化技巧，实现了对基本 Pollard rho 算法的加速。通过改进的 Pollard Kangroo 算法得到了更好的性能。对于大数 DLP，相比于参考时间提升了约 1h，经过 Argon2id hash 检验正确。

八、参考文献

【1】LectureNotes10.pdf，
https://math.mit.edu/classes/18.783/2021/LectureNotes10.pdf

【2】Coarsely integrated operand scanning (CIOS) architecture for high-speed Montgomery modular multiplication | IEEE Conference Publication | IEEE Xplore，
https://ieeexplore.ieee.org/document/1393267

【3】Speeding Up the Pollard Rho Method on Prime Fields，
https://link.springer.com/content/pdf/10.1007/978-3-540-89255-7_29.pdf

【4】S0025-5718-1978-0491431-9.pdf，
https://www.ams.org/journals/mcom/1978-32-143/S0025-5718-1978-0491431-9/S0025-5718-1978-0491431-9.pdf

【5】Pollard's Kangaroo Algorithm Pollard 袋鼠算法 - 知乎，
https://zhuanlan.zhihu.com/p/603786377

【6】ECC2-131 的并行 Pollard rho 算法实现分析*，
http://www.jcr.cacrnet.org.cn/CN/10.13868/j.cnki.jcr.000522#18