数论算法详解

1. 整数的取余运算

定义 带余除法 设 a,b 是整数，且 $b>0$，则存在非负整数 q,r，使得

$$a=bq+r$$

且 $0\le r<b$，称 q 为商，r 为余数

显然带余除法中的商和余数都是唯一的，在下文中将商记为 a/b，将余数记为 $a\%b$，“/”与“%”的运算优先级与乘除法相同，当然，在C语言中二者分别对应 `a/b` 与 `a%b`

1.1 整数的加减乘在 $(\mathbb{Z}_m,+,\cdot)$ 中的运算

1.1.1 定理 整数的加、减、乘对于取余的右分配律

$$(a+b)\%c=(a\%c+b\%c)\%c$$

$$(a-b)\%c=(a\%c-b\%c+c)\%c$$

$$(ab)\%c=(a\%c)(b\%c)\%c$$

注意，在计算减法时，通常需要加 c，防止变成负数

在计算乘法时，如果 c 较大（但不超过64位整数范围），可以使用快速乘法进行计算，原理与快速幂运算类似，复杂度为 $O(\log b)$

```
ll fastMul(ll a,ll b,ll p){
    a%=p;
    ll ans=0;
    while(b>0){
        if(b&1)ans=(ans+a)%p;
        b>>=1;
        a=(a+a)%p;
    }
    return ans;
}
```

如果需要更快的快速乘法，可以使用 long double 数据类型进行计算，复杂度为 $O(1)$

```
ll modMul(ll a,ll b,ll p){
    if (p<=1000000000) return a*b%p;
    else if (p<=100000000000000ll) return (((a*(b>>20)%p)<<20)+(a*(b&
((1<<20)-1))))%p;
    else {
        ll d=(ll)floor(a*(long double)b/p+0.5);
        ll ret=(a*b-d*p)%p;
        if (ret<0) ret+=p;
        return ret;
    }
}
```

1.2 整数的除法在 $(\mathbb{Z}_p,+,\cdot)$ 中的运算(p 为素数)

虽然取余运算对于“+”、“-”、“×”不难，但通常情况下

$$\frac{a}{b}\%c\neq\frac{a\%c}{b\%c}\%c$$

如何计算 $\frac{a}{b}\%c$？我们有时可以找一个乘法逆元 b^{-1}，使得 $bb^{-1}\equiv 1(mod\ c)$，那么就有

$$\frac{a}{b}\%c=ab^{-1}\%c$$

如果 c 是素数，有下面的定理

1.2.1 定理 费马小定理 设 b 是一个整数，c 是一个素数，二者互质，那么

$$b^{c-1}\equiv 1(mod\ c)$$

将上式改写一下，得到

$$bb^{c-2}\equiv 1(mod\ c)$$

因此取 $b^{-1}=b^{c-2}$ 即可，一般需要用快速幂计算，但要注意，与除数不能为0类似，要保证 $b\%c\neq 0$

```
ll inv(ll a,ll p){
    return fpow(a,p-2);
}
```

1.3 整数的除法在 $(\mathbb{Z}_m,+,\cdot)$ 中的运算

上面的方法给出了模数为素数的解决方案，如果模数不是素数，可以用下面的方法

1.3.1 定理 若 $b|a$，则 $\frac{a}{b}\%c=\frac{a\%(bc)}{b}$

这样可以不使用逆元来求 $(\mathbb{Z}_m,+,\cdot)$ 中的除法

其他逆元相关内容将在下文中介绍

（如果你像我一样无聊，可以把整环 $(\mathbb{Z}_m,+,\cdot)$ 中的数写成这样一个类）

```
class mint{
    static const ll mo=1e9+7;
    inline ll fpow(ll a,ll b){ll c=1;while(b)
{if(b&1)c=c*a%mo;b>>=1;a=a*a%mo;}return c;}
    inline ll inv(ll a){return fpow(a,mo-2);}
    inline ll norm(ll a){return a<0?(a%mo+mo):a%mo;}
public:
    ll v;
    mint(){}
    mint(ll x):v(x){}
    mint &operator =(const ll b){v=norm(b);return *this;}
    mint &operator +=(const mint &b){v+=b.v;if(v>=mo)v-=mo;return *this;}
    mint &operator -=(const mint &b){v-=b.v;if(v<0)v+=mo;return *this;}
    mint &operator *=(const mint &b){v=v*b.v%mo;return *this;}
    mint &operator /=(const mint &b){v=v*inv(b.v)%mo;return *this;}
    mint operator + (const mint &b)const{mint a{*this};return a+=b;}
    mint operator - (const mint &b)const{mint a{*this};return a-=b;}
    mint operator * (const mint &b)const{mint a{*this};return a*=b;}
    mint operator / (const mint &b)const{mint a{*this};return a/=b;}
```

```
    mint operator ^ (const mint &b){return mint(fpow(v,b.v));}
    mint &operator++(){v++;if(v==mo)v-=mo;return *this;}
    mint &operator--(){v--;if(v==-1)v+=mo;return *this;}
    mint operator -()const{return mint(v?(mo-v):0);}
    bool operator ==(const mint &b)const{return v==b.v;}
    bool operator <(const mint &b)const{return v<b.v;}
    bool operator >(const mint &b)const{return v<b.v;}
    bool operator !=(const mint &b)const{return v!=b.v;}
    explicit operator ll()const{return v;}
    friend ostream &operator << (ostream &o,const mint & b){o<<b.v;return o;}
    friend istream &operator >> (istream &in,mint &a){in>>a.v;a.v%=mo;return
in;}
};
```

2. 最大公因数与最小公倍数

2.1 最大公因数

顾名思义，最大公因数就是公因数中最大的那个，我们记 a,b 的最大公因数为 $gcd(a,b)$，有如下性质

2.1.1 性质 $gcd(a,b)=gcd(b,a)$

2.1.2 性质 $gcd(a,b)=gcd(a-b,b)(a\geq b)$

2.1.3 性质 $gcd(a,b)=gcd(a\%b,b)$

2.1.4 性质 $gcd(a,b,c)=gcd(gcd(a,b),c)$

2.1.5 性质 $gcd(ka,kb)=k\ gcd(a,b)$

2.1.1 性质是显然的，2.1.2 性质是辗转相减法的原理，2.1.3 性质可以视为2.1.2 性质的“一步到位”版本，2.1.4 性质指出多个数的最大公因数可以递推地进行求解，2.1.5 性质说明 gcd 对乘法有分配律

2.2 辗转相除法

根据2.1.3 性质，得到辗转相除法的参考代码模板

```
typedef long long ll;
ll gcd(ll a, ll b){
    return b?gcd(b,a%b):a;
}
```

注意当 $b\neq 0$ 时，返回值为 `gcd(b,a%b)` 而不是 `gcd(a%b,b)`，否则会不断递归导致栈溢出

2.3 最小公倍数

最小公倍数就是公倍数中最小的那个，我们记 a,b 的最小公倍数为 $lcm(a,b)$，有如下性质

2.3.1 性质

$$lcm(a,b)=\frac{ab}{gcd(a,b)}$$

下面是最小公倍数的参考代码模板

```
ll lcm(ll a, ll b){
    return a/gcd(a,b)*b;
}
```

注意是先除后乘，避免在中间过程中数据超出64位整数的范围

2.4 扩展欧几里得算法

考虑二元一次不定方程

$$ax + by = c$$

其中 a, b, c 是已知的正整数，如何求出方程的解呢?

2.4.1 定理 上述方程有解的充要条件是 $gcd(a, b)|c$ （c 是 $gcd(a, b)$ 的倍数）

可以理解为，$gcd(a, b)$ 是 $ax + by$ 可以表示出的最小正整数

2.4.2 定理 方程 $ax + by = d, d = gcd(a, b)$ 的所有解为

$$\begin{cases} x = x_0 + \frac{b}{d}t \\ y = y_0 - \frac{a}{d}t \end{cases}$$

其中 x_0, y_0 是一组特解

2.4.3 定理 方程 $ax + by = c, gcd(a, b)|c$ 的所有解为

$$\begin{cases} x = \frac{c}{d}x_0 + \frac{b}{d}t \\ y = \frac{c}{d}y_0 - \frac{a}{d}t \end{cases}$$

其中 x_0, y_0 是方程 $ax + by = d, d = gcd(a, b)$ 的一组特解

下面是参考代码模板:

```
ll ext_gcd(ll a,ll b,ll& x,ll& y){
    ll d = a;
    if (!b){
        x = 1;y = 0;
    }else{
        d = ext_gcd(b,a%b,y,x);
        y -= a/b*x;
    }
    return d;
}
```

在求逆元时，要找到 b^{-1} 使得 $bb^{-1} \equiv 1(mod\ c)$ ，实质上是求解方程 $bx + cy = 1$ 中的 x ，因此可以用扩展欧几里得算法来求逆元，当然只有 $gcd(b, c) = 1$ 时才有解，否则逆元不存在

2.4.4 推论 逆元的存在性 存在 $b^{-1} \in \mathbb{Z}, s.t.\ bb^{-1} \equiv 1(mod\ c)$ 的充要条件是 $gcd(b, c) = 1$

3. 同余方程组

有方程就有方程组

$$\begin{cases} x \equiv a_1(mod\ m_1) \\ x \equiv a_2(mod\ m_2) \\ \cdots \\ x \equiv a_n(mod\ m_n) \end{cases}$$

其中 $a_1, a_2, \ldots, a_n$ 是整数， $m_1, m_2, \ldots, m_n$ 是正整数

3.1.1 中国剩余定理（孙子定理） 设上述方程组中 $m_1, m_2, \ldots, m_n$ 两两互质，则方程组的通解为

$$x = k\prod_{i=1}^{n} m_i + \sum_{i=1}^{n} a_i M_i M_i^{-1}$$

其中 $M_i = \prod_{j\neq i} m_j$

下面是参考代码模板，需要调用前面的扩展欧几里得算法模板

```
ll Sunzi(ll *m,ll *a,int len){
    ll lcm = 1;
    ll ans = 0;
    for (int i=0;i<len;i++){
        ll k0,ki;
        ll d = ext_gcd(lcm,m[i],k0,ki);
        if ((a[i]-ans)%d!=0) return -1;
        else {
            ll t = m[i]/d;
            k0 = ( k0*(a[i]-ans)/d%t + t)%t;
            ans = k0*lcm + ans;
            lcm = lcm/d*m[i];
        }
    }
    return ans;
}
```

4. 素数

素数是只有1和本身两个因数的数，1不是素数

4.1 素数的判断

```
bool isPrime(ll n){
    if(n==1)return false;
    for(ll i=2;i*i<=n;i++)
        if(n%i==0)return false;
    return true;
}
```

这是最简单的素数的判断的参考代码模板，复杂度为 $O(\sqrt{n})$

原理其实很简单，对于一个大于1的整数，如果 x 是它的一个大于 $\sqrt{n}$ 的因子，那么 $\frac{n}{x}$ 是它的小于 $\sqrt{n}$ 的因子

在大多数情况下，这种判断方式的复杂度已经足够小了，如果要追求更高的效率，可以考虑 $kn+i$ 法

一个大于1的整数如果不是素数，那么一定有素因子，因此在枚举因子时只需要考虑可能为素数的因子即可。 $kn+i$ 法即枚举形如 $kn+i$ 的数，例如取 $k=6$ ，那么 $6n+2, 6n+3, 6n+4, 6n+6$ 都不可能为素数，只需要枚举形如 $6n+1, 6n+5$ 的数即可，这样复杂度降低了 $\frac{2}{3}$ 。

下面的模板是 $kn+i$ 法 $k=30$ 的版本

```
bool isPrime(ll n){
    if(n==2||n==3||n==5)return 1;
    if(n%2==0||n%3==0||n%5==0||n==1)return 0;
    ll c=7,a[8]={4,2,4,2,4,6,2,6};
    while(c*c<=n)for(auto i:a){if(n%c==0)return 0;c+=i;}
    return 1;
}
```

如果 n 极大，可以使用素数测试算法，素数测试算法可以通过控制迭代次数来间接控制正确率，常用的有下面的Miller-Rabin方法

```
ll Rand(){
    static ll x=(srand((int)time(0)),rand());
    x+=1000003;
    if(x>1000000007)x-=1000000007;
    return x;
}
bool Witness(ll a,ll n){
    ll t=0,u=n-1;
    while(!(u&1))u>>=1,t++;
    ll x=fpow(a,u,n),y;
    while(t--){
        y=x*x%n;
        if(y==1 && x!=1 && x!=n-1)return true;
        x=y;
    }
    return x!=1;
}
bool MillerRabin(ll n,ll s){
    if(n==2||n==3||n==5)return 1;
    if(n%2==0||n%3==0||n%5==0||n==1)return 0;
    while(s--){
        if(Witness(Rand()%(n-1)+1,n))return false;
    }
    return true;
}
```

当然，`Rand()` 怎么写可以自由发挥，这会影响其性能

4.2 素数筛

如果要求出不超过 n 的所有素数，素数筛是最好的选择，下面是一种朴素的筛法

```
void getPrime(bool p[],int n){
    for(int i=1;i<=n;i++)p[i]=true;
    p[1]=false;
    for(int i=2;i<=n;i++){
        if(p[i]){
            for(int j=i+i;j<=n;j+=i)p[j]=false;
        }
    }
}
```

这种方法的原理是从小到大将素数的倍数筛掉，复杂度为 $O(n\log n)$，注意到每个合数如果有多个素因子，那么就会被重复筛掉，造成复杂度的浪费，因此，用下面的方法可以保证**每个合数只被它最小的素因子筛掉一遍**，以 $O(n)$ 的复杂度解决上述问题

```
ll getPrime(ll n,bool vis[],ll prime[]){
    ll tot=0;
    for(ll i=1;i<=n;i++)vis[i]=0;
    for(ll i=2;i<=n;i++){
        if(!vis[i])prime[tot++]=i;
        for(ll j=0;j<tot;j++){
            if(prime[j]*i>n)break;
            vis[prime[j]*i]=1;
            if(i%prime[j]==0)break;
        }
    }
    return tot;
}
```

5. 组合数取余

5.1 组合数

5.1.1 定义 组合数 在 $n(n\geq 0)$ 个不同元素中选取 $m(0\leq m\leq n)$ 个元素，不同的取法记为

$$C_n^m=\frac{n!}{m!(n-m)!}$$

5.2 杨辉三角

组合数与杨辉三角中的数字是一一对应的

杨辉三角的自然数形式

$$\begin{matrix}1&1&1&1&1\\1&2&3&4&5\\1&3&6&10&15\\1&4&10&20&35\\1&5&15&35&70\end{matrix}$$

杨辉三角的组合数形式

$$\begin{matrix}C_0^0&C_1^1&C_2^2&C_3^3&C_4^4\\C_1^0&C_2^1&C_3^2&C_4^3&C_5^4\\C_2^0&C_3^1&C_4^2&C_5^3&C_6^4\\C_3^0&C_4^1&C_5^2&C_6^3&C_7^4\\C_4^0&C_5^1&C_6^2&C_7^3&C_8^4\end{matrix}$$

按照上面的写法，杨辉三角的第 n 行第 m 列即为 C_{n+m-2}^{m-1}

注意到上图中每个数等于其左边的数与上边的数（如果有的话）之和，这就是**杨辉恒等式**

$$C_n^m=C_{n-1}^{m-1}+C_{n-1}^m$$

5.3 组合数取余的求法

在ACM竞赛中，我们常常需要计算 $C_n^m\%p$，可以参考下面几种方法

1. 如果 n,m 很小（不超过50），可以用C++的库函数 `double tgamma(double x)` ，这是一个欧拉积分

$$\Gamma(s)=\int_0^{+\infty}x^{s-1}e^{-x}dx$$

在整数点处的取值满足

$$\Gamma(n+1)=n!$$

因此代码可以这么写

```
ll C(ll n,ll m){
    return (ll)round(tgamma(n+1)/tgamma(m+1)/tgamma(n-m+1));
}
```

效率并不高，但是对于追求手速来说足够了

2. 如果 n,m 不大，可以开 $O(n^2)$ 的空间，可以利用杨辉恒等式来预处理组合数表

```
const ll mo=1e9+7;
ll c[1005][1005];
void getC(int n){
    for(int i=0;i<=n;i++){
        for(int j=0;j<=i;j++){
            if(j==0 || j==i)
                c[i][j]=1;
            else
                c[i][j]=(c[i-1][j-1]+c[i-1][j])%mo;
        }
    }
}
```

3. 如果 n,m 比较大，可以开 $O(n)$ 的空间，可以利用前文所述的逆元来求解，当然，要保证 p 是素数

```
const ll mo=1e9+7;
ll C(ll n,ll m){
    static ll M=0,inv[N],mul[N],invMul[N];
    while(M<=n){
        if(M){
            inv[M]=M==1?1:(mo-mo/M)*inv[mo%M]%mo;
            mul[M]=mul[M-1]*M%mo;
            invMul[M]=invMul[M-1]*inv[M]%mo;
        }
        else mul[M]=1,invMul[M]=1;
        M++;
    }
    return mul[n]*invMul[m]%mo*invMul[n-m]%mo;
}
```

上面的代码中用 $O(n)$ 的复杂度处理了 $[1,n]$ 的逆元，处理 Q 次 $n,m\le N$ 的询问的总复杂度为 $O(N+Q)$

4. 如果 n,m 更大， p 是素数，可以用Lucas定理来求解

5.3.1 Lucas定理

若 p 是素数，则

$$C_n^m = \prod_{i=0}^{k} C_{n_i}^{m_i} \, (mod \, p)$$

其中

$$n = \sum_{i=0}^{k} n_i p^i$$

$$m = \sum_{i=0}^{k} m_i p^i$$

即将 n, m 表示成 p 进制形式

5.3.2 推论

$$C_n^m \equiv \chi(n \& m = m) (mod \, 2)$$

```
ll Lucas(ll n,ll m,ll p){
    ll ans=1;
    while(n|m)ans=ans*C(n%P,m%P)%P,n/=P,m/=P;
    return ans;
}
```

6. $\mathbb{Z}^*$ 与 $(\mathbb{Z}_p^*, \cdot)$ 的结构

6.1 $\mathbb{Z}^*$ 的结构

6.1.1 算数基本定理 任何一个大于1的整数 n ，都可以唯一地表示成素数乘积的形式

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$$

其中 $p_1, p_2, \dots, p_k$ 是素数

对于一个较大的数，有用来分解其因数的 Pollard Rho 算法

```
typedef long long ll;
typedef pair<int,int> pii;
typedef pair<ll,ll> pll;
namespace pollard_rho{
    const int C=2307;
    const int S=10;
    typedef pair<ll,int> pli;
    mt19937 rd(time(0));
    vector<ll>ve;
    ll gcd(ll a,ll b){return b?gcd(b,a%b):a;}
    ll mul(ll a,ll b,ll mod){return (__int128)a*b%mod;}
    ll power(ll a,ll b,ll mod){
        ll res=1;a%=mod;
        while(b){
            if(b&1)res=mul(res,a,mod);
            a=mul(a,a,mod);
            b>>=1;
        }
        return res;
```

```
    }
    bool check(ll a,ll n){
        ll m=n-1,x,y;
        int j=0;
        while(!(m&1))m>>=1,j++;
        x=power(a,m,n);
        for(int i=1;i<=j;x=y,i++){
            y=mul(x,x,n);
            if(y==1&&x!=1&&x!=n-1)return 1;
        }
        return y!=1;
    }
    bool miller_rabin(ll n){
        ll a;
        if(n==1)return 0;
        if(n==2)return 1;
        if(!(n&1))return 0;
        for(int i=0;i<s;i++)if(check(rd()%(n-1)+1,n))return 0;
        return 1;
    }
    ll pollard_rho(ll n,int c){
        ll i=1,k=2,x=rd()%n,y=x,d;
        while(1){
            i++;x=(mul(x,x,n)+c)%n,d=gcd(y-x,n);
            if(d>1&&d<n)return d;
            if(y==x)return n;
            if(i==k)y=x,k<<=1;
        }
    }
    void findfac(ll n,int c){
        if(n==1)return ;
        if(miller_rabin(n)){
            ve.push_back(n);
            return ;
        }
        ll m=n;
        while(m==n)m=pollard_rho(n,c--);
        findfac(m,c);
        findfac(n/m,c);
    }
    vector<pli> solve(ll n){
        vector<pli>res;
        ve.clear();
        findfac(n,c);
        sort(ve.begin(),ve.end());
        for(auto x:ve){
            if(res.empty()||res.back().fi!=x)res.push_back({x,1});
            else res.back().se++;
        }
        return res;
    }
}
```

6.1.2 推论 若

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$$

$$m=p_1^{\beta_1}p_2^{\beta_2}\dots p_k^{\beta_k}$$

则

$$nm=p_1^{\alpha_1+\beta_1}p_2^{\alpha_2+\beta_2}\dots p_k^{\alpha_k+\beta_k}$$

$$gcd(n,m)=p_1^{min(\alpha_1,\beta_1)}p_2^{min(\alpha_2,\beta_2)}\dots p_k^{min(\alpha_k,\beta_k)}$$

$$lcm(n,m)=p_1^{max(\alpha_1,\beta_1)}p_2^{max(\alpha_2,\beta_2)}\dots p_k^{max(\alpha_k,\beta_k)}$$

6.1.3 定理 n 的阶乘中 p 的幂次为

$$\sum_{r=1}^{+\infty}\left[\frac{n}{p^r}\right]$$

6.1.4 定理 $(p-1)!+1\equiv 0(mod\ p)$

6.2 $(\mathbb{Z}_p^*,\cdot)$ 的结构

定理6.2.1 $(\mathbb{Z}_p^*,\cdot)$ 是循环群，即存在 $a\in\mathbb{Z}_p^*$，使得

$$\mathbb{Z}_p^*=\{a^n|n=1,2,\dots,p-1\}$$

这样的 a 称为 p 的原根

素数一定有原根，原根不唯一，部分合数也有**原根**

原根一般不大，暴力枚举即可

1000000007的原根为5，998244353的原根为3

7. 离散对数与BSGS算法

考虑求解方程

$$a^x\equiv b\ (mod\ p)$$

这样的 x 称为离散对数，可以写为 $\log_a b\ (mod\ p)$

Baby step giant step 算法

设 $x=kn+i$ （n 为某常正整数），则原方程可以写成 $(a^n)^k=b(a^{-1})^i$

将 $\left(b\left(a^{-1}\right)^i,i\right),i=0,1,\dots,n-1$ 存入表（`table`，C++中可以用`unordered_map`）中，然后枚举 k，在表中查找 $(a^n)^k$ 即可，复杂度为 $O(n+\frac{p}{n})$，取 $n=\sqrt{p}$，那么复杂度为 $O(\sqrt{p})$

```
ll bsgs(ll a,ll b,ll p){
    static unordered_map<ll,ll> tab;
    tab.clear();
    ll u=(ll)sqrt(p)+1;
    ll now=1,step;
    rep(i,0,u-1){
        ll tmp=b*inv(now,p)%p;
        if(!tab.count(tmp))tab[tmp]=i;
        (now*=a)%=p;
    }
    step=now;
    now=1;
```

```
    for(ll i=0;i<p;i+=u){
        if(tab.count(now))return i+tab[now];
        (now*=step)%=p;
    }
    return -1;
}
```

8. 高次同余方程

考虑求解方程

$$x^a \equiv b \ (mod \ p)$$

先求 p 的原根 g，设 $x \equiv g^u \ (mod \ p)$，$b \equiv g^t \ (mod \ p)$，用BSGS算法求出 t，方程可写成

$$g^{au} \equiv g^t \ (mod \ p)$$

进而有

$$au + (p-1)v = t$$

用扩展欧几里得算法求出 u，也就求出了 x

```
ll ModEquationSolve(ll a,ll y,ll p){
    a%=p-1;
    ll g=primitiveRoot(p),t=bsgs(g,y,p),z,z_,d=ext_gcd(a,p-1,z,z_);
    if(t%d!=0)return -1;
    ll tmp=(p-1)/d;
    z=(t/d*z%tmp+tmp)%tmp;
    return fpow(g,z,p);
}
```

9. 积性函数

9.1 积性函数

9.1.1 定义 积性函数 映射 $f:\mathbb{Z}^* \to \mathbb{R}$ 如果满足：任意 $a,b \in \mathbb{Z}^*$ 且 $gcd(a,b)=1$，有

$$f(ab) = f(a)f(b)$$

那么称 f 为积性函数

最简单的积性函数 $I(n) = \chi(n=1)$ 与 $E(n)=1$

其中 $\chi(a)$ 为示性函数，当条件 a 成立时取1，否则取0

9.1.2 定理 积性函数必满足

$$f(1) = 1$$

9.2 欧拉函数

9.2.1 定义 欧拉函数

$$\phi(n) = \sum_{i=1}^{n} \chi\left(gcd\left(i,n\right)=1\right)$$

欧拉函数是积性函数，且有

$$\phi(n)=n\prod_{p|n}\left(1-\frac{1}{p}\right)$$

```
vector<ll> factors(ll x){
    vector<ll> fac;
    ll y=x;
    for(ll i=2;i*i<=x;i++){
        if(y%i==0){
            fac.push_back(i);
            while(y%i==0)y/=i;
            if(y==1)return fac;
        }
    }
    if(y!=1)fac.push_back(y);
    return fac;
}
ll Euler(ll n){
    vector<ll> fac=factors(n);
    ll ans=n;
    for(auto p:fac)ans=ans/p*(p-1);
    return ans;
}
```

9.2.2 欧拉定理 若 m 是大于1的整数，$gcd(a,m)=1$，则

$$a^{\phi(m)}\equiv 1\ (mod\ m)$$

这又是一个求逆元的方法，其结果与扩展欧几里得算法相同

9.3 莫比乌斯函数

9.3.1 定义 莫比乌斯函数

$$\mu(n)=\begin{cases}1, n=1\\ (-1)^k, n=p_1p_2\dots p_k\\ 0, otherwise\end{cases}$$

```
ll Mobius(ll n){
    vector<ll> fac=factors(n);
    for(auto p:fac)n/=p;
    return n>1?0:(fac.size()&1)?-1:1;
}
```

9.3.2 定理 莫比乌斯反演

$$g(n)=\sum_{d|n}f(d)\Longleftrightarrow f(n)=\sum_{d|n}\mu(d)\,g(\frac{n}{d})$$

9.3.2' 定理 莫比乌斯反演的另一种形式

$$g(n)=\sum_{n|d}f(d)\Longleftrightarrow f(n)=\sum_{n|d}\mu\left(\frac{d}{n}\right)g(d)$$

9.4 积性函数筛

积性函数可以通过前文中的线性素数筛在线性时间复杂度内求出，某些取值依赖于自变量的素因子的数论函数也能通过其求出

```
#define rep(i,a,b) for(ll i=a;i<=b;i++)
const ll N=10005;
//积性数论函数筛
void Number_Theory_Function_Solve(vector<ll> &a,ll n,string function_name){
    static bool vis[N];
    static ll prime[N];
    a.resize(n+1);
    ll tot=0;
    rep(i,1,n)vis[i]=0;
    a[1]=1;
    if(function_name=="IsPrime"){
/*
       { 1, if x is a prime
f(x) = {
       { 0, otherwise

素数判断函数不是积性函数，但可以用积性函数筛求解
*///
        rep(i,2,n){
            if(!vis[i])prime[tot++]=i,a[i]=1;
            rep(j,0,tot-1){
                if(prime[j]*i>n)break;
                vis[prime[j]*i]=1;
                if(i%prime[j]==0){
                    a[i*prime[j]]=0;
                    break;
                }
                a[i*prime[j]]=0;
            }
        }
        a[1]=0;
    }else if(function_name=="E"){
/*
                { 1, x=1
f(x) = I(x=1) = {
                { 0, otherwise

积性函数群中的单位元
*///
        rep(i,2,n)a[i]=0;
    }else if(function_name=="Euler"){
/*
        n
f(x) = sum I(gcd(i,x)=1) = x product (1-1/p)
       i=1                   p|x, p is Prime

欧拉函数
*///
        rep(i,2,n){
            if(!vis[i])prime[tot++]=i,a[i]=i-1;
            rep(j,0,tot-1){
                if(prime[j]*i>n)break;
                vis[prime[j]*i]=1;
                if(i%prime[j]==0){
```

```
                    a[i*prime[j]]=a[i]*prime[j];
                    break;
                }
                a[i*prime[j]]=a[i]*a[prime[j]];
            }
        }
    }else if(function_name=="Mobius"){
/*
       { 1, x=1
f(x) = { (-1)^k, n=p1 p2 ... pk, pi is prime
       { 0, otherwise

莫比乌斯函数
*///
        rep(i,2,n){
            if(!vis[i])prime[tot++]=i,a[i]=-1;
            rep(j,0,tot-1){
                if(prime[j]*i>n)break;
                vis[prime[j]*i]=1;
                if(i%prime[j]==0){
                    a[i*prime[j]]=0;
                    break;
                }
                a[i*prime[j]]=a[i]*a[prime[j]];
            }
        }
    }else if(function_name=="FactorCounter"){
/*
        n
f(x) = sum I(i|x)
       i=1

因子个数
*///
        static ll t[N];
        rep(i,2,n){
            if(!vis[i])prime[tot++]=i,a[i]=2,t[i]=1;
            rep(j,0,tot-1){
                if(prime[j]*i>n)break;
                vis[prime[j]*i]=1;
                if(i%prime[j]==0){
                    a[i*prime[j]]=a[i]/(t[i]+1)*(t[i]+2);
                    t[i*prime[j]]=t[i]+1;
                    break;
                }
                a[i*prime[j]]=a[i]*a[prime[j]];
                t[i*prime[j]]=1;
            }
        }
    }else if(function_name=="FactorSum"){
/*
        n
f(x) = sum I(i|x)*i
       i=1

因子和
*///
        static ll pw[N];
```

```
        rep(i,2,n){
            if(!vis[i])prime[tot++]=i,a[i]=i+1,pw[i]=i;
            rep(j,0,tot-1){
                if(prime[j]*i>n)break;
                vis[prime[j]*i]=1;
                if(i%prime[j]==0){
                    a[i*prime[j]]=
(pw[i]*prime[j]*prime[j]-1)/(prime[j]-1)*a[i/pw[i]];
                    pw[i*prime[j]]=pw[i]*prime[j];
                    break;
                }
                a[i*prime[j]]=a[i]*a[prime[j]];
                pw[i*prime[j]]=prime[j];
            }
        }
    }else if(function_name=="MinFactor"){
        rep(i,2,n){
            if(!vis[i])prime[tot++]=i,a[i]=i;
            rep(j,0,tot-1){
                if(prime[j]*i>n)break;
                vis[prime[j]*i]=1;
                if(i%prime[j]==0){
                    a[i*prime[j]]=prime[j];
                    break;
                }
                a[i*prime[j]]=prime[j];
            }
        }
    }else{
        assert(false);
    }
}
```

9.5 数论函数的狄利克雷乘积

9.5.1 定义 数论函数的狄利克雷乘积

$$(f*g)(n)=\sum_{d|n}f(d)g\left(\frac{n}{d}\right)$$

9.5.2 定理 所有积性数论函数构成一个群，该群有单位元

$$I(x)=\chi(x=1)$$

任意积性数论函数 f ，都存在唯一的积性数论函数 g ，使得

$$f*g=g*f=E$$

且

$$g(1)=\frac{1}{f(1)},g(n)=\frac{-1}{f(1)}\sum_{d|n,d<n}f\left(\frac{n}{d}\right)g(d)$$

称 g 为 f 的狄利克雷逆函数

9.5.3 推论 数论函数间的关系 设 ϕ 为欧拉函数，μ 为莫比乌斯函数，I 为单位数论函数，E 为常数函数（ $E(n)=1$ ），i 为恒等函数（ $i(n)=n$ ），那么

$$\mu * E = I$$

$$\phi * E = i$$

$$\mu * i = \phi$$

```
bool mulFunCheck(const vector<ll> &val){
    ll n=val.size()-1;
    static bool vis[N];
    static ll prime[N],pw[N],t[N],tot=0;
    if(val[1]!=1)return false;
    rep(i,1,n)vis[i]=0;
    rep(i,2,n){
        if(!vis[i])prime[tot++]=i,pw[i]=i,t[i]=1;
        rep(j,0,tot-1){
            if(prime[j]*i>n)break;
            vis[prime[j]*i]=1;
            if(i%prime[j]==0){
                if(val[i*prime[j]]!=val[i/pw[i]]*val[pw[i]*prime[j]])return
false;
                pw[i*prime[j]]=pw[i]*prime[j],t[i*prime[j]]=t[i]+1;
                break;
            }
            if(val[i*prime[j]]!=val[i]*val[prime[j]])return false;
            pw[i*prime[j]]=prime[j],t[i*prime[j]]=1;
        }
    }
    return true;
}
//狄利克雷乘积 h(n)=\sum_{d|n}f(d)g(n/d)
vector<ll> DirichletProduct(const vector<ll> &f,const vector<ll> &g){
    ll n=f.size()-1;
    vector<ll> h(n+1,0);
    rep(i,1,n)h[i]=0;
    rep(i,1,n)rep(j,1,n/i)h[i*j]+=f[i]*g[j];
    return h;
}
//狄利克雷逆函数 f*g=g*f=[n==1]
vector<ll> DirichletInversion(const vector<ll> &f){
    ll n=f.size()-1;
    vector<ll> g(n+1,0);
    g[1]=1;
    rep(i,1,n)rep(j,2,n/i)g[i*j]-=f[j]*g[i];
    return g;
}
//Mobius反演 f(n)=sum(g(d),d|n) g(n)=sum(f(d)mu(n/d),d|n)
vector<ll> MobiusInversion(const vector<ll> &f){
    ll n=f.size()-1;
    static vector<ll> mu;
    Number_Theory_Function_Solve(mu,n,"Mobius");
    return DirichletProduct(mu,f);
}
```

10. 多项式乘积算法

试想这样一个问题，求两个多项式

$$f(x)=\sum_{i=0}^{n-1}a_ix^i$$

$$g(x)=\sum_{i=0}^{m-1}b_ix^i$$

的乘积

$$f(x)g(x)=\sum_{i=0}^{n+m-2}\sum_{j+k=i}(a_j+b_k)x^i$$

使用传统的方法至少需要 $O(n^2)$ 的复杂度，如何加速上述过程?

10.1 快速傅里叶变换

首先考虑如何用其他方式表示多项式 $f(x)=\sum_{i=0}^{n-1}a_ix^i$.

任取n个不同的数(可以是整数、实数，甚至是复数)

$$x_0,x_1,\cdots,x_{n-1}$$

将其代入 $f(x)$ 中，就得到一个线性方程组

$$\begin{cases}f(x_0)=y_0\\f(x_1)=y_1\\\cdots\\f(x_{n-1})=y_{n-1}\end{cases}$$

只要 n 足够大，就能够唯一地确定一个多项式，换言之，上述方程组可以表示一个多项式，将这两种多项式的表示方法分别称为系数表示和点值表示.

利用快速傅里叶变换来求多项式乘积的总体思路是

1. 选取合适的 n 个不同的数 $x_0,x_1,\cdots,x_{n-1}$
2. 将多项式 $f(x)$ 与 $g(x)$ 转化为点值表示(称为离散傅里叶变换，简称DFT)
3. 计算 $f(x)g(x)$ 的点值表示
4. 将 $f(x)g(x)$ 转化为系数表示(称为逆离散傅里叶变换，简称 DFT^{-1})

1. 选取合适的 n 个不同的数 $x_0,x_1,\cdots,x_{n-1}$

我们选取复数域上 $\sqrt[n]{1}$ 的 n 个不同的值(或称 n 个 n 次单位复根)作为 $x_0,x_1,\cdots,x_{n-1}$ 的值，即

$$x_k=\omega_n^k=e^{\frac{2k\pi i}{n}},k=0,2,\cdots,n-1$$

至于指数形式的复数 $e^{\frac{2k\pi i}{n}}$ ，用大家所熟知的欧拉公式即可求得其代数形式

$$e^{i\theta}=\cos\theta+i\sin\theta$$

经过简单计算可知

$$\omega_n^{k+mn}=\cos\left(\frac{2k\pi}{n}+2\pi m\right)+i\sin\left(\frac{2k\pi}{n}+2\pi m\right)=\cos\frac{2k\pi}{n}+i\sin\frac{2k\pi}{n}=\omega_n^k,m\in\mathbb{Z}$$

当 n 为偶数时

$$\left(\omega_n^k\right)^2=\left(e^{\frac{2k\pi i}{n}}\right)^2=e^{\frac{2k\pi i}{n/2}}=\omega_{n/2}^k=\omega_{n/2}^{k\ mod\ n/2}$$

其中 $a\ mod\ b$ 为 a 除以 b 的余数，上述两等式将在后文中使用.

我们为什么要费尽周折选取如此复杂的点呢?是为了使用快速傅里叶变换.

2. 将多项式 $f(x)$ 与 $g(x)$ 转化为点值表示

考虑多项式 $f(x)=\sum_{i=0}^{n-1}a_ix^i$ ，当 $n=2^m,m\in\mathbb{Z}^+$ 时(当不满足该条件时，向 $f(x)$ 补充系数为0的高次项来扩大 n 使其满足该条件)，将其化为两个多项式

$$f^{[0]}(x)=a_0+a_2x+\cdots+a_{n-2}x^{\frac{n-2}{2}}$$

$$f^{[1]}(x)=a_1+a_3x+\cdots+a_{n-1}x^{\frac{n-2}{2}}$$

则有

$$f(x)=f^{[0]}(x^2)+xf^{[1]}(x^2)$$

进而

$$f(\omega_n^k)=f^{[0]}(\omega_{n/2}^{k\ mod\ n/2})+\omega_n^kf^{[1]}(\omega_{n/2}^{k\ mod\ n/2})$$

也就是说，要求 $f(x)$ 在 n 个不同点处的值，只需要求 $f^{[0]}(x)$ 与 $f^{[1]}(x)$ 在 $\frac{n}{2}$ 个不同点处的值，由于 $n=2^m,m\in\mathbb{Z}^+$ ，可对 $f^{[0]}(x)$ 与 $f^{[1]}(x)$ 重复进行上述过程，最终经过 m 步后得到 n 个函数

$$f^{[0]}(x)=a_0,f^{[1]}(x)=a_1,\cdots,f^{[n-1]}(x)=a_{n-1}$$

之后回推得到 $f(x)$ 的点值表示，上述过程就是快速傅里叶变换的过程，复杂度为 $O(nm)$ 即 $O(n\log n)$.

当然，还需要对 $g(x)$ 进行同样的变换.

3. 计算 $f(x)g(x)$ 的点值表示

点值表示的优点是可以快速地求出两个选取了相同点值的多项式的乘积，例如多项式

$$\begin{cases}f(x_0)=y_0\\f(x_1)=y_1\\\cdots\\f(x_{n-1})=y_{n-1}\end{cases}$$

与多项式

$$\begin{cases}g(x_0)=z_0\\g(x_1)=z_1\\\cdots\\g(x_{n-1})=z_{n-1}\end{cases}$$

的乘积

$$\begin{cases}f(x_0)g(x_0)=y_0z_0\\f(x_1)g(x_1)=y_1z_1\\\cdots\\f(x_{n-1})g(x_{n-1})=y_{n-1}z_{n-1}\end{cases}$$

只需要 $O(n)$ 的复杂度即可求得.

4. 将 $f(x)g(x)$ 转化为系数表示

下面以 $f(x)$ 为例，讲解如何将多项式从点值表示转化为系数表示，此过程又称多项式的插值.

将 $f(x)$ 的点值表示写成矩阵形式 $Y=V_nA$ 即

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

此处矩阵 V_n 中的1可视为 ω_n^0 .

现在我们已知的是 Y 和 V_n ，要求的是 A ， V_n 是一范德蒙德矩阵，可求得其逆矩阵

$$V_n^{-1} = \frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \omega_n^{-3} & \cdots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \omega_n^{-6} & \cdots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \omega_n^{-3(n-1)} & \cdots & \omega_n^{-(n-1)(n-1)} \end{pmatrix}$$

因此 $A = V_n^{-1} Y$ 即

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \omega_n^{-3} & \cdots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \omega_n^{-6} & \cdots & \omega_n^{-2(n-1)} \\ 1 & \omega_n^{-3} & \omega_n^{-6} & \omega_n^{-9} & \cdots & \omega_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \omega_n^{-3(n-1)} & \cdots & \omega_n^{-(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

也就是说，只需将 Y 与 A 对换，将 ω_n^k 换成 ω_n^{-k} ，再乘上系数 $\frac{1}{n}$ ，进行类似步骤2的变换，即可进行逆快速傅里叶变换，算法复杂度同样为 $O(n \log n)$.

按照上述方法将 $f(x)g(x)$ 转化为系数表示，本题得解.

```
#include<bits/stdc++.h>
using namespace std;
const int maxn=1<<18;
//typedef complex<double> C;
struct C
{
    double a,b;
    C(){}
    C(double a,double b):a(a),b(b){}
    C operator = (int a){*this=C(a*1.0,0);return *this;}
    C operator + (const C &t){return C(a+t.a,b+t.b);}
    C operator - (const C &t){return C(a-t.a,b-t.b);}
    C operator * (const C &t){return C(a*t.a-b*t.b,a*t.b+b*t.a);}
};
C wn(int n,int f)
{
    return C(cos(acos(-1.0)/n),f*sin((acos(-1.0))/n));
}
C inv(int n)
{
    return C(1.0/n,0);
}
```

```
C a[maxn],b[maxn],c[maxn];
int g[maxn];
void FFT(C *a,int n,int f)
{
    for(int i=0;i<n;i++)if(i>g[i])swap(a[i],a[g[i]]);
    for(int i=1;i<n;i<<=1)
    {
        C w=wn(i,f),x,y;
        for(int j=0;j<n;j+=i+i)
        {
            C e;e=1;
            for(int k=0;k<i;e=e*w,k++)
            {
                x=a[j+k];
                y=a[j+k+i]*e;
                a[j+k]=x+y;
                a[j+k+i]=x-y;
            }
        }
    }
    if(f==-1)
    {
        C Inv=inv(n);
        for(int i=0;i<n;i++)a[i]=a[i]*Inv;
    }
}
void conv(C *a,int n,C *b,int m,C *c)
{
    int k=0,s=2;
    while((1<<k)<max(n,m)+1)k++,s<<=1;
    for(int i=1;i<s;i++)g[i]=(g[i/2]/2)|((i&1)<<k);
    FFT(a,s,1);
    FFT(b,s,1);
    for(int i=0;i<s;i++)c[i]=a[i]*b[i];
    FFT(c,s,-1);
}
int main()
{
    int n,m;
    scanf("%d%d",&n,&m);
    for(int i=0;i<=n;i++)scanf("%lf",&a[i]);
    for(int i=0;i<=m;i++)scanf("%lf",&b[i]);
    conv(a,n,b,m,c);
    for(int i=0;i<=n+m;i++)printf("%d ",(int)(c[i].a+0.5));
    return 0;
}
```

10.2 快速数论变换

快速傅里叶变换可以用于计算两个实系数多项式的乘积，但毕竟有精度问题，对于整系数多项式和**模M意义下的乘法**，是否有无损精度的算法呢?

FFT中精度损失的关键出在第一步——$x_0, x_1, \ldots, x_{n-1}$的选取，算法的实现依赖一个基本的等式

$$\left(\omega_n^k\right)^2 = \omega_{n/2}^{\mathrm{k\ mod\ n/2}}$$

这个等式导致， $x_0, x_1, \ldots, x_{n-1}$有$n$个不同的取值，但它们的平方只有$\frac{n}{2}$个不同的取值，这$\frac{n}{2}$个值的平方只有$\frac{n}{4}$个不同的取值，以此类推

若M满足某些条件时，是可以找到整数$x_0, x_1, \ldots, x_{n-1}$的，例如当
$M = 998244353 = 2^{23} \cdot 7 \cdot 17 + 1$，$n = 2^k, k \leq 23$时，取

$$\omega_n = 3^{\frac{M-1}{n}} \text{ mod M}$$

此时恰好满足类似的性质，后续的算法步骤类似，但这种变换就不是离散傅里叶变换（DFT）了，而是**数论变换（NTT）**

```
const ll mo=998244353;
ll fpow(ll a, ll b){
    ll ans=1;
    while(b>0){if(b&1)ans=ans*a%mo;b>>=1;a=a*a%mo;}
    return ans;
}
ll D(ll x) {
    ((x>=mo) && (x-=mo)) || ((x<0) && (x+=mo));
    return x;
}
void NTT(ll a[],ll n,ll op) {
    for(ll i=1,j=n>>1;i<n-1;++i) {
        if(i<j)
            swap(a[i],a[j]);
        ll k=n>>1;
        while(k<=j) {
            j-=k;
            k>>=1;
        }
        j+=k;
    }
    for(ll len=2;len<=n;len<<=1) {
        ll rt=fpow(3,(mo-1)/len);
        for(ll i=0;i<n;i+=len) {
            ll w=1;
            for(ll j=i;j<i+len/2;++j) {
                ll u=a[j],t=1LL*a[j+len/2]*w%mo;
                a[j]=D(u+t),a[j+len/2]=D(u-t);
                w=1LL*w*rt%mo;
            }
        }
    }
    if(op==-1) {
        reverse(a+1,a+n);
        ll in=fpow(n,mo-2);
        for(ll i=0;i<n;++i)
            a[i]=1LL*a[i]*in%mo;
    }
}
vector<ll> Conv(vector<ll> const &A,vector<ll> const &B,ll N) {
    static ll a[2000005],b[2000005];
    auto Make2=[](ll x)->ll {
        return 1<<((32-__builtin_clz(x))+((x&(-x))!=x));
    };
    ll n=Make2(A.size()+B.size()-1);
    for(ll i=0;i<n;++i) {
```

```
        a[i]=i<A.size()?A[i]:0;
        b[i]=i<B.size()?B[i]:0;
    }
    NTT(a,n,1);NTT(b,n,1);
    for(ll i=0;i<n;++i)
        a[i]=1LL*a[i]*b[i]%mo;
    NTT(a,n,-1);
    vector<ll> C(N);
    for (ll i=0;i<N;i++)
        C[i]=a[i];
    return C;
}
```