模板

1. 数学

1.1. GCD 、LCM

【定理】$k|m$和$k|n \Leftrightarrow k|gcd(m,n)$

```
static long GCD(long a, long b) {
        return b == 0 ? a : GCD(b, a % b);
    }

    static long lcm(long a, long b) {
        return a / GCD(a, b) * b;
    }
```

1.1.1. 求n个数的最小公倍数

多个数的最小公倍数，先将两个数的最小公倍数求出，再与后面的数求最小公倍数。

```
static long all_lcm(long arr[]) {
        int n=arr.length;
        long e,gcd,k;
        k=gcd=1;
        for(int i=0;i<n;i++) {
            e=arr[i];
            gcd=k/GCD(e, k)*e;
            k=gcd;
        }
        return gcd;
    }
```

1.1.2. 性质

1. GCD是积性函数。
2. 交换律$gcd(a,b)=gcd(b,a)$
3. $gcd(-a,b)=gcd(a,b)$
4. $gcd(a,a)=|a|$
5. $gcd(a,0)=|a|$
6. $gcd(a,b)=gcd(b,a$

7. $gcd(a,b)=gcd(b,a-b)$;
8. $gcd(a+mb,b)=gcd(a,b)$
9. 对任意自然数m, $gcd(ma,mb)=m\cdot gcd(a,b)$(分配律)
10. 如果m是a和b的最大公约数 $gcd(a/m,b/m)=1$
11. 对任意自然数m, $gcd(ab,m)=gcd(a,m)*gcd(b,m)$
12. $gcd(a,lcm(b,c))=lcm(gcd(a,b),gcd(a,c))$
13. $lcm(a,gcd(b,c))=gcd(lcm(a,b),lcm(a,c))$
14. 设$a>b,gcd(a,b)=1$,那么$gcd(a^m-b^m,a^n-b^n)=a^{gcd(m,n)}-b^{gcd(m,n)}$
15. 设$a>1,m,n>0$;那么有$gcd(a^m-1,a^n-1)=a^{gcd(m,n)}-1$
16. 对任意自然数m,n；$gcd(a^m,b^n)=gcd(a,b)^{min(n,m)}$
17. 对于$\forall a>1$，$\forall a$的约数$c(c\neq a)$，令$b=a-c$,则有$c=gcd(a,b)$;
18. 设F_n为Fib数，那么有$gcd(F_m,F_n)=F_{gcd(m,n)}$

1.2. 对数运算

- $log10()$, 以10为底，$log()$以e为底.
- 换底公式：$\log_a b=\dfrac{\log_c b}{\log_c b}$
- 运算公式：$\log_a(b\cdot c)=\log_a b+\log_a c$

$$\log_a\frac{b}{c}=\log_a b-\log_a c$$

$$10aa^{b^c}=c\log_a b$$

1.3. 扩展GCD

什么是拓展欧几里得？简单的说，就是求关于x,y的方程 $ax+by=gcd(a,b)$ 的所有整数解

```
    static int[] exGCD(int a,int b)
    {   //扩展欧几里得
        //triple[0]=d,triple[1]=x,triple[2]=y
        int[]triple=new int[3];
        if(b==0)
        {
            triple[0]=a;
            triple[1]=1;
            triple[2]=0;
            return triple;
        }
        int[] t=exGCD(b,a%b);
        triple[0]=t[0];
        triple[1]=t[2];
        triple[2]=t[1]-(a/b)*t[2];
        return triple;
```

```
    }
```

应用：

1.3.1. 解模线性方程

输入正整数a,b,n,解方程$ax \equiv b(modn)$ 等价于$ax+by-c=0$

1.当且仅当d|b时，方程ax = b(mod n)有解。d=gcd(a,n)

2.ax = b(mod n) 或者有d个不同解，或者无解。

3.令d=gcd(a,n) 假定对整数x', y', 有d = ax' + ny'， 如果d | b, 则方程ax = b(mod n)有一个解的值为x0, 满足:

x0=x`(b/d)(mod n)

4.假设方程ax = b(mod n)有解， x0是方程的任意一个解， 则方程对模n恰有d个不同的解， 分别为:

xi = x0 + i * (n / d), 其中 i = 1,2,3.....d - 1

定理：如果任意整数a,b不都为0，那么$gcd(a,b)$是a,b的线性组合集$\{ax+by,\ (x,y \in Z)\}$中的最小正元素。

```java
public static void main(String[] args) {
        int a = 3, b = 4, c = 5;
        // ax+by=c
        int ans[]=exGCD(a, b);
        int d=ans[0],x=ans[1],y=ans[2];
        if (c % d != 0) {
            System.out.println("无解");
            return;
        }
        System.out.println("有解");
        System.out.println("最大公约数: " + d);
        // 特解
        y = y * (c / d);
        x = x * (c / d);
        System.out.println("特解:x= " + x + " y=" + y);

        // 全解
        int r, t;
        r = a / d;
        t = b / d;
```

```
        System.out.println("全解:");
        for (int i = 0; i < d; i++)
            System.out.println("x=" + (x - i * t) + " y=" + (y + i * r));

        // 最小整数解
        x = (x % t + t) % t;
        System.out.println("最小整数解: x=" + x + " y=" + (c - (a * x)) / b);

    }
```

1.3.2. exgcd求逆元$O(logn)$

$ax \equiv 1(modn)$有**唯一解**当且仅当a与n**互质**。

等价于$ax - ny = 1$

```
static int inv(int a,int n)
{//计算a关于n的逆元
    int triple[]=exGCD(a, n);
    return triple[0]==1?(triple[1]+n)%n:0;
}
```

1.4. 素数

1.4.1. 线性筛$O(logN)$

```
static int maxn=(int)1e5+5;
    static boolean vis[]=new boolean[maxn];
    static int prime[]=new int[maxn];//也可以用ArrayList代替
    static int tot=0;
    static void sieve() {
        for(int i=2;i<maxn;i++) {
            if(!vis[i]) prime[tot++]=i;
            for(int j=0;j<tot&&i*prime[j]<maxn;j++) {
                vis[i*prime[j]]=true;
                if(i%prime[j]==0) break;
            }
        }
    }
```

1.4.2. 埃式筛$O(n * lglgn)$

当数据量小于10^5时使用埃式筛，否则用线性筛。

```
    static void sieve2() {
```

```
        // 打表1~maxn的素数
        vis[0] = vis[1] = true;
        for (int i = 2; i < maxn; i++) {
            if (!vis[i]) {
                prime[tot++] = i;
                if (i > maxn / i)
                    continue;
                for (int j = i * i; j < maxn; j += i) {
                    if (!vis[j])
                        vis[j] = true;
                }
            }

        }
    }
```

1.4.3. 唯一分解定理

唯一分解式：$n = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k} = \prod_{i=1}^{k} p_i^{\alpha_i} \qquad (n > 1)$

将一个分数分解为若干素数整数幂的乘积 分别将分子分母的`exponent`对应项求和。

求n的唯一分解式中d的幂

```
static void addInteger(int n,int d,int exponent[]) {
        //可以计算一个分式的 唯一分解式,首先用素数打表
        // d=1时表示n为分子，d=-1时表示n为分母
        //exponent存储各素数的指数

        for(int i=0;i<prime.length;i++) {
            while(n%prime[i]==0) {
                n/=prime[i];
                exponent[i]+=d;
            }
            if(n==1) break;
        }
    }

//利用唯一分解式，求出原来的数
double ans=1;       //可能是小数
for(int i=0;i<prime.length;i++)
    ans*=Math.pow(prime[i],e[i]);
```

1.4.4. 快速质因数分解$O(n^{\frac{1}{4}})$

求解整数n的唯一分解式

```java
static int divide(int n, int d, ArrayList<Integer> list) {
        //计算n中有多少个素因子d并将其存入list中
        while(n%d==0) {
            n/=d;
            list.add(d);
        }
        return n;
    }
    static ArrayList<Integer>divideAll(int n){
        //求n的唯一分解式
        ArrayList<Integer> list=new ArrayList<>();//存储唯一分解式
        int m=(int)Math.floor(Math.sqrt(n)+0.5);
        for(int i=2;i<m;i++) {
            if(n%i==0)
              n=divide(n, i, list);
            if(n==0) break;
        }
        if(n>1) list.add(n);
        return list;
    }
```

1.4.5. 勒让德定理

定理定义

在正数n!的素因子标准分解式中，素数p的最高指数记作 $L_p(n!)$，则 $L_p(n!)=\sum_{k\geq 1}\left[\frac{n}{p^k}\right]$.

```java
static int f(int n, int p) {
        // 求解n! 的唯一分解式中p的指数
        int exp = 0;
        while (n != 0) {
            exp += n / p;
            n = n / p;
        }
        return exp;
    }

```

1.4.6. 快速素数检测

```
boolean is_prime( int num )
{
    //快速判断一个数是否为素数

    //两个较小数另外处理
    if(num ==2|| num==3 )
        return true ;
    //不在6的倍数两侧的一定不是质数
    if(num %6!= 1&&num %6!= 5)
        return false ;
    int tmp =(int)Math.sqrt(num);
    //在6的倍数两侧的也可能不是质数
    for(int i= 5; i <=tmp; i+=6 )
        if(num %i== 0||num %(i+ 2)==0 )
            return false ;
    //排除所有，剩余的是质数
    return true ;
}
```

1.5. 欧拉函数

对正整数n，*欧拉函数*是小于n的正整数中与n互质的数的数目。

1.5.1. 性质

1. $\phi(1) = 1$
2. 除了$n = 2$，$\phi(n)$都是偶数。
3. 设m和n是互素的正整数,那么$\phi(mn) = \phi(m)\phi(n)$，特殊的，当$m = 2$，$n$为奇数时，$\varphi(2 * n) = \varphi(n)$
4. 设p为素数
    1. $\phi(p) = p - 1$
    2. 若已知$phi(x)$，且p能整除x：$phi(x * p) = phi(x) * p$;
    3. 若已知$phi(x)$，且p不能整除x：$phi(x * p) = phi(x) * (p - 1)$;
5. 如果p是**素数**，a是一个正整数，那么$\phi(p^a) = p^a - p^{a-1}$
6. $n!$ 的欧拉函数就是小于等于n的所有素数。既：$\phi(n!) = n!(1 - 1/p_1)(1 - 1/p_2)(1 - 1/p_3)\dots(1 - 1/p_k), p_k$为小于等于n的所有素数
7. 对于两个正整数m和n,如果n是m的倍数，那么[1..n]中与m互素的数的个数为$\frac{n}{m}\phi(m)$
8. 小于n的数中，与n互质的数的总和为：$\varphi(n) * n/2(n > 1)$。

9. $n=\sum_{d|n}\varphi(d)$，即n的因数（包括1和它自己）的欧拉函数之和等于n。
10. **欧拉定理：** $a^{\phi(n)}\equiv 1(modn)$，要求$a$和$n$互质。可以用来求a模n的乘法逆元：$a^{-1}\equiv a^{\phi(n)-1}(modn)$

前十个欧拉函数值

1、1、2、2、4、2、6、4、6、4

1.5.2. 单值欧拉函数

```
static int euler(int n) {
    //计算n的欧拉函数值
        int m = (int) Math.sqrt(0.5 + n);
        int ans = n;
        for (int i = 2; i <= m; i++) {
            if (n % i == 0) {
                ans = ans / i * (i - 1);
                while (n % i == 0)
                    n /= i;
            }
        }

        if (n > 1)
            ans = ans / n * (n - 1);
        return ans;
    }
```

1.5.3. 线性欧拉筛$O(N)$

```
    static int maxn = (int) 1e5 + 5;
    static int phi[] = new int[maxn];
    static int prime[] = new int[maxn];
    static boolean vis[] = new boolean[maxn];

    static int table(int n) {
        // 计算1~n的所有欧拉函数,返回1~n以内素数的个数
        // prime存储1~n的所有素数，phi存储1~n的所有欧拉值
        phi[1] = 1;
        int tot = 0;
        for (int i = 2; i <= n; i++) {
            if (!vis[i]) {
                prime[tot++] = i;
                phi[i] = i - 1;
            }
            for (int j = 0; j < tot; j++) {
```

```
                if (i * prime[j] > n)
                    break;
                vis[i * prime[j]] = true;
                if (i % prime[j] == 0) {
                    phi[i * prime[j]] = phi[i] * prime[j];
                    break;
                } else
                    phi[i * prime[j]] = phi[i] * (prime[j] - 1);
            }
        }
        return tot;
    }
```

1.5.4. 欧拉函数打表$O(Nlog(logN))$

```
static int phi[] = new int[maxn];
static void table2(int n) {
    //计算1~n的欧拉函数
        phi[1]=1;
        for(int i=2;i<=n;i++) {
            if(phi[i]==0) {
                for(int j=i;j<=n;j+=i) {
                    if(phi[j]==0)
                        phi[j]=j;
                    phi[j]=phi[j]/i*(i-1);
                }
            }
        }
    }
```

当数据范围超过10^6需要使用线性筛,否则使用普通打表。

1.6. 逆元

1.6.1. 逆元的计算

> 对于正整数 a 和 m，如果有 $ax \equiv 1(modm)$，那么把这个同余方程中 x 的最小正整数解叫做 a 模 m 的逆元。

最常见的是扩展欧几里得算法与费马小定理,要求a与m互质。

扩展欧几里得

```
    static int[] exGCD(int a,int b)
```

```
    {   //扩展欧几里得
        //triple[0]=d,triple[1]=x,triple[2]=y
        int[]triple=new int[3];
        if(b==0)
        {
            triple[0]=a;
            triple[1]=1;
            triple[2]=0;
            return triple;
        }
        int[] t=exGCD(b,a%b);
        triple[0]=t[0];
        triple[1]=t[2];
        triple[2]=t[1]-(a/b)*t[2];
        return triple;
    }
static int inv(int a,int m)          //扩展欧几里得求逆元
{//计算a关于n的逆元
    int triple[]=exGCD(a, m);
    return triple[0]==1?(triple[1]+m)%m:0;
}
```

欧拉定理

```
    static int powMod(int a, int b, int m) {
        int ans = 1;
        for (; b != 0; b >>= 1) {
            if (b % 2 == 1)
                ans = (ans * a) % m;
            a = (a * a) % m;
        }
        return ans;
    }
static int inv(int a,int m) //欧拉定理计算逆元，要求a与m互质
{
    return powMod(a,m-2,m);
}
```

1.6.2. 逆元递推$O(P)$

```
//在模质数p的情况下，求1~P的逆元
inv[1] = 1;
for (int i=2; i<=n; ++i) {
   inv[i] = (long) (p - p / i) * inv[p%i] % p;
}
```

逆元递推只能求$[1..p]$的逆元，但是对于$i > p$的逆元，有：$inv[i] = inv[i$;

1.6.3. 阶乘逆元

```
static int invf[ ]=new int[N], factor[ ]=new int[N];
void get_factorial_inverse(int n, int p) {
    factor[0] = 1;
    for (int i = 1; i <= n; ++i) {
        factor[i] = i * factor[i - 1] % p;
    }
    invf[n] = powMod(factor[n], p-2,p);          //快速幂取模
    for (int i = n-1; i >= 0; --i) {
        invf[i] = invf[i + 1] * (i + 1) % p;
    }
}
```

1.7. 组合数

$$
\begin{aligned}
&\text{定义公式：} C_n^m = \frac{n!}{m!(n-m)!} \qquad (1)\\
&\text{杨辉三角递推：} C_n^m = C_{n-1}^m + C_n^{m-1}\\
&\text{递推} C_{n+1}^m = C_n^m + n\\
&C_n^m + C_{m+1}^m + \ldots + C_{m+n}^m = C_{m+n+1}^{m+1}\\
&C_n^0 + C_{n+1}^1 + \ldots + C_{n+k}^k = C_{n+k+1}^k\\
&C_{2n}^2 = 2C_n^2 + n^2
\end{aligned}
$$

1.7.1. 杨辉三角递推$O(n^2)$

(1) $1 \le m \le n \le 1000$和$1 \le p \le 10^9$

杨辉三角递推：$C_n^n = C_{n-1}^m + C_n^{m-1}$

```
static int M = 10007;
static int MAXN = 1000;
static int C[][]=new int[MAXN+1][MAXN+1];
static void Initial()
{
    int i, j;
    for(i = 0; i <= MAXN; ++i)
    {
        C[0][i] = 0;
```

```
        C[i][0] = 1;
    }
    for(i = 1; i <= MAXN; ++i)
    {
        for(j = 1; j <= MAXN; ++j)
            C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % M;
    }
}
```

1.7.2. 组合数计算

预处理阶乘逆元，再利用定义式。：$C_n^m = \frac{n!}{m!(n-m)!}$

```
    static int maxn;
    static long inv[]=new long[maxn], fac[ ]=new long[maxn], facInv[ ]=new
long[maxn];
    static int n = 100;
    static long P;
    static void f()
    {
        inv[1] = 1; fac[0] = facInv[0] = 1;
        for(int i = 1; i <= n; i++)
        {
            if(i != 1) inv[i] = (P - P / i) * inv[(int)P % i] % P;
            fac[i] = fac[i - 1] * i % P;
            facInv[i] = facInv[i - 1] * inv[i] % P;
        }
    }
    long C(int n, int m)
    {
        return fac[n] * facInv[m] % P * facInv[n - m] % P;
    }
```

1.7.3. 质因子分解$O(logn)$

主要是要预处理素数打表较为花费时间，不用处理逆元。

$1 \le n \le m \le 10^8$

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

设$n!$分解因式后，素因子P的指数为a；$m!$分解质因数后P的指数为b；$(n-m)!$分解质因数后P的指数为c。则C_n^m分解之后P的指数为

```
    static int maxn=(int)1e5+5;
```

```
    static boolean vis[]=new boolean[maxn];
    static int prime[]=new int[maxn];//也可以用ArrayList代替
    static int tot=0;
    static void sieve() {                //素数打表
        for(int i=2;i<maxn;i++) {
            if(!vis[i]) prime[tot++]=i;
            for(int j=0;j<tot&&i*prime[j]<maxn;j++) {
                vis[i*prime[j]]=true;
                if(i%prime[j]==0) break;
            }
        }
    }
//计算n!中素因子p的指数
//勒让德定理
static int f(int n, int p) {
    // 求解n! 的唯一分解式中p的指数
        int exp = 0;
        while (n != 0) {
            exp += n / p;
            n = n / p;
        }
        return exp;
    }
    static int powMod(int a, int b, int m) {
        int ans = 1;
        for (; b != 0; b >>= 1) {
            if (b % 2 == 1)
                ans = (ans * a) % m;
            a = (a * a) % m;
        }
        return ans;
    }
    //计算C(n,m)
    static int C(int n, int m, int mod)
    {
        int ans = 1;
        for(int i = 0; i < tot & prime[i] <= n; i++)
        {
            int ret = fac[n] - fac[m] - fac[n - m];
            ans = 1ll * ans * (powMod(prime[i], ret, mod)) % mod;
        }
        return ans;
    }
```

1.7.4. Lucas定理

$$C_n^m \% p,\ 1 \le m \le n \le 10^{18}, 2 \le p \le 10^9 \tag{2}$$

$n = (a_k, a_{k-1}, \dots a_0)_p$

$m = (b_k, b_{k-1}, \dots, b_0)_p$

$\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{a_i}{b_i} \mod p$

```
static int lucas(int n, int m, int mod)
{//C(n,m,mod)计算组合数取模
    if(n < m) return 0;
    int ans = 1;
    for(; m; n /= mod, m /= mod) ans = 1L * ans * C(n % mod, m % mod, mod) %
mod;
    return ans;
}
```

1.8. 整数分块

可以用到整除分块的形式，大致是这样的：

$\sum_{i=1}^{n} f(i)\lfloor \frac{n}{i} \rfloor$　　$f(i)$为积性函数

对于每一个$\lfloor \frac{n}{i} \rfloor$我们可以通过打表可以发现：有许多$\lfloor \frac{n}{i} \rfloor$的值是一样的，而且它们呈一个**块状分布**；再通过打表之类的各种方法,我们惊喜的发现对于每一个值相同的块，它的最后一个数就是$n/(n/i)$。得出这个结论后，我们就可以做$O(\sqrt{n})$次的处理了。

一维整除分块$\sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor$的算法实现

```
for(int l=1,r;l<=n;l=r+1)
{
    r=n/(n/l);
    ans+=(r-l+1)*(n/l);
}
```

二维整数分块$\sum_{i=1}^{n} \sum_{j=1}^{m} \lfloor \frac{n}{i} \rfloor \lfloor \frac{m}{j} \rfloor$的算法实现

```
int f()
{
    int ans = 0;
    for(int l = 1, r; l <= n; l = r + 1)
    {
        for(int x = 1, y; x <= m; x = y + 1)
        {
            r = n / (n / l);
```

```
            y = m / (m / x);
            ans += (r - l + 1) * (n / l) * (y - x + 1) * (m / x);
        }
    }
    return ans;
}
```

1.9. 快速幂$O(logN)$

```
    static int powMod(int a, int b, int m) {
        int ans = 1;
        for (; b != 0; b >>= 1) {
            if (b % 2 == 1)
                ans = (ans * a) % m;
            a = (a * a) % m;
        }
        return ans;
    }
```

1.9.1. 费马小定理降幂

求$a^x\%p$，降幂时要求P为质数：：既$a^x \equiv a(mod \quad p)^{x\%(p-1)}(mod \quad p)$

$\forall a$与p互质，且p是质数有$a^p \equiv a \mod p$

既$a^{p-1} \equiv 1(mod \quad p) \equiv a^0$,$a^x(mod \quad p)$有循环节，长度为$p-1$

```
powMod(a%p,x%(p-1),p);        //降幂公式
```

1.9.2. 扩展欧几里得降幂

$$a^b \equiv \begin{cases} a^{b\%\phi(p)} & gcd(a,p)=1 \\ a^b & gcd(a,p)\neq 1, b<\phi(p) \\ a^{b\%\phi(p)+\phi(p)} & gcd(a,p)\neq 1, b\geq \phi(p) \end{cases} \quad (mod\ p) \tag{3}$$

1.9.3. 大数快速乘

于计算机底层设计的原因，做加法往往比乘法快的多，因此将乘法转换为加法计算将会大大提高（**大数**，比较小的数也没必要）乘法运算的速度，除此之外，当我们计算a*b%mod的时候，**往往较大的数计算a * b会超出long long int的范围**，这个时候使用快速乘法方法也能解决上述问题.

快速乘法的原理就是利用乘法分配率来将a * b转化为多个式子相加的形式求解（注意这时使用乘法分配率的时候后面的一个乘数转化为二进制的形式计算）.

```java
 //大数相乘取模
static long multi(long x, long y, long mod)
 {
     long t=0;
     x %= mod;
     for(t = 0; y!=0; x = (x << 1) % mod, y >>= 1)
         if (y & 1)
             t = (t + x) % mod;
     return t;
 }
```

用`multi`方法替换乘法，可以计算大数的快速幂

```java
    static long powMod(long a, long b, long m) {
        long ans = 1;
        for (; b != 0; b >>= 1) {
            if (b % 2 == 1)
                ans = multi(ans, a, m);
            a = multi(a, a, m);
        }
        return ans;
    }
```

1.10. 矩阵快速幂

考虑

$$\begin{cases} F_1 = A \\ F_2 = B \\ F_n = C * F_{n-2} + D * F_{n-1} \end{cases} \quad A, B, C, D\text{为常数} \tag{4}$$

可以构造如下**矩阵递推**式

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} D & C \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} \tag{5}$$

边界 是

$$\begin{bmatrix} F_3 \\ F_2 \end{bmatrix} = \begin{bmatrix} D & C \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = \begin{bmatrix} D & C \\ 1 & 0 \end{bmatrix} \begin{bmatrix} B \\ A \end{bmatrix} \tag{6}$$

得出结论:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} D & C \\ 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} B \\ A \end{bmatrix} \qquad n > 2 \tag{7}$$

再考虑

$$\begin{cases} F_1 = A \\ F_2 = B \\ F_n = C * F_{n-2} + D * F_{n-1} + P \end{cases} \qquad n > 2, P\text{为常数} \tag{8}$$

同样有

$$\begin{bmatrix} F_n \\ F_{n-1} \\ P \end{bmatrix} = \begin{bmatrix} D & C & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{n-2} \begin{bmatrix} B \\ A \\ P \end{bmatrix} \qquad n > 2 \tag{9}$$

注意矩阵乘法$A * B \neq B * A$

```java
public class Matrix {
     int n;                                  // 矩阵阶数，不能是静态的
     long matrix[][] ;       //不能是静态的
     public Matrix(int n) {
        this.n=n;
         matrix=new long[n][n];
    }
     Matrix multi(Matrix rhs) {              //不用是静态的
        Matrix ans = new Matrix(n);
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++)
                if (matrix[i][j] != 0) {
                    for (int k = 0; k < n; k++)
                        ans.matrix[i][k] += matrix[i][j] * rhs.matrix[j][k];
                }
        }
        return ans;
    }

    Matrix pow(Matrix M, long m) {  //不是静态的
        // 计算矩阵M的m次幂
        Matrix ans = new Matrix(n);
        for (int i = 0; i < M.n; i++) // 构造单位矩阵
            ans.matrix[i][i] = 1;
        for (; m > 0; m >>= 1) {
            if ((m & 1) == 1)
                ans = ans.multi(M);
            M = M.multi(M);
        }
```

```
        return ans;
    }
    public static void main(String args[]) {
        //构造全1矩阵
        Matrix input=new Matrix(2);
        for(int i=0;i<input.n;i++)
            for(int j=0;j<input.n;j++)
                input.matrix[i][j]=1;
        //计算3次幂
        Matrix ans=input.pow(input, 3);
        for(int i=0;i<input.n;i++)
            for(int j=0;j<input.n;j++)
                System.out.println(ans.matrix[i][j]);
    }
}
```

2. 单调队列

正向单调队列可以维护固定区间长度的的递增序列，要求每次区间最右边元素必须在队列中。

```
static int N = 10000010;
    public static void main(String args[])
    {
        Scanner cin=new Scanner(new BufferedInputStream(System.in));
        int h, t;
        int a[]=new int[100], q[]=new int[100];
        int n, m;
        n=cin.nextInt();
        m=cin.nextInt();
        for(int i = 1; i <= n; i++)
           a[i]=cin.nextInt();
        h = 1;
        t = 0;
        for(int i = 1; i <= n; i++)
        {
            while(t >= h && a[q[t]] >= a[i]) //从队首开始抛出所有大于等于a[i]的元素下标，使得队首元素最大
                t--;                                //将>号改为<实现递减单调栈
            q[++t] = i;                             //a[i]入队
            if(i >= m)                  //区间长度达到m
            {
               if(q[h] == i - m )
                    h++;    //队尾元素不再序列内，要抛出
                for(int j = h; j <= t; j++)     //输出单调队中的元素
                    System.out.printf("%d ", a[q[j]]);
```

```
                System.out.println();
            }
        }
    }
/*input:
7 4
5 2 6 8 10 7 4
Output:
2 6 8
2 6 8 10
6 7
4
*/
```

反向维护单调队列！！！不仅可以维护区间最大（小）值，还能维护区间最大（小）值的改变次数（队列元素个数）队列中的元素下标并不连续。

下面以反向维护为例讲解：

假设题目序列为

> 3 2 2 1 5 7 6 8 2 9
>
> 序列长度n=10,区间长度为m=6，问每个区间的最大值改变次数。

我们从$i=n$到$n-m+1$维护一个单调递减的单调队列$q[h.....t]$，其中h是队尾下标，t是队首下标。**队列中存储在队列中的元素下标** 。加入“9”在队列中，“9”的下标为10，则有
$q[h]=10;$(队列中的第一个元素)

维护原则：$a[i]$,必然要放入队列中，**放入之前**更新队列，从队首开始**抛出所有小于等于**a[i]的元素；这样就保证了队列的单调递减性（自队尾至队首），然后**将a[i]放入队首**；注意随着区间的移动，如果**队尾元素不在区间内就要抛出队尾**，更新完成后，**队尾元素是区间的最大值，并且队列的元素个数是区间最大值改变次数。**

区间从$[5,10].....[1,6]$过程中单调队列中的元素, 从队尾到队首！！。

> [5, 10]: 9 8 7 5
>
> [4, 9]: 8 7 5 1
>
> [3, 8]: 8 7 5 2
>
> [2, 7]: 7 5 2
>
> [1, 6]: 7 5 3

可以看出这和每个区间的最长单调子序列对应，并且队尾元素就是区间最值。

```java
static int N = 10000010;
    public static void main(String args[])
    {
        Scanner cin=new Scanner(new BufferedInputStream(System.in));
        int h, t;
        int a[]=new int[100], q[]=new int[100];
        int n, m;
        n=cin.nextInt();
        m=cin.nextInt();
        for(int i = 1; i <= n; i++)
           a[i]=cin.nextInt();
        h = 1;
        t = 0;
        for(int i = n; i >0; i--)
        {
            while(t >= h && a[q[t]] <= a[i]) //从队首开始抛出所有大于等于a[i]的元素下标，使得队首元素最大
                t--;                                //将>号改为<实现递减单调栈
            q[++t] = i;                             //a[i]入队
            if(i+m-1<=n)                 //区间长度达到m
            {
                if(q[h] == i + m )
                    h++;                 //队尾元素不再序列内，要抛出
                for(int j = h; j <= t; j++)      //输出单调队中的元素
                    System.out.printf("%d ", a[q[j]]);
                System.out.printf("max:%d  len:%d\n",a[q[h]],t-h+1);
            }
        }
    }
/*
10 6
3 2 2 1 5 7 6 8 2 9
9 8 7 5 max:9  len:4
8 7 5 1 max:8  len:4
8 7 5 2 max:8  len:4
7 5 2 max:7  len:3
7 5 3 max:7  len:3
*/
```

从题目可以看出如果求的是区间最大值，区间最大值改变次数！！子序列长度，我们就反向维护一个单调递减的单调队列。

反之我们写一个反向维护单调递增的单调队列，求区间最小值，区间最小值改变次数，只需将

```
while(t >= h && a[q[t]] >= a[i])

    t--;       //将<号改为>号就维护了一个单调递降队列。


```

3. 二分查找

```
//注意查找区间[l,r)!!!!
    static int search(int A[], int l, int r, int val) {
        // 查找元素val的下标，不存在返回-1
        int m;
        while (l < r) {
            m = l + (r - l) / 2;

            if (A[m] == val)
                return m;
            else if (A[m] > val)
                r = m;
            else
                l = m + 1;
        }
        return -1;
    }

    static int lowerBound(int A[], int l, int r, int val) {
        // 查找第一个大于等于val的下标，若不存在输出r，r不属于A的下标
        int m;
        while (l < r) {
            m = l + (r - l) / 2;
            if (A[m] >= val)
                r = m;
            else
                l = m + 1;
        }
        return l;
    }

    static int upperBound(int A[], int l, int r, int val) {
        // 查找第一个大于val的下标，若不存在输出r，r不属于A的下标
        int m;
        while (l < r) {
            m = l + (r - l) / 2;
            if (A[m] > val)

```

```
                r = m;
            else
                l = m + 1;
        }
        return l;
    }
```

4. 日期计算

```
/*
 * 模板有风险,使用需谨慎 ,
 * 在获取某年的天数时,可以打表
 *
 */
// 声明
struct Time ;
inline bool IsrunYear( int const & year );
// 输入一个日期, 返回是这年的第几天
int Cnt_day_of_year( Time const& time);
// 得到这年的第几天这个日期
Time Time_of_nTH_day(int const&year,int days);
//获取某年的天数
inline int getdayOFyear( int y );
//计算两个日期相差的天数
int  Cnt_day_between_Times(Time t1,Time t2);
// 一个日期n天后的日期
Time next_Days_after_Time( Time time,int days );
// 1.是闰年 0.是平年
extern int dayOFmonth[2][13] ;


#include <cstdio>
#include <iostream>
using namespace std;
struct Time{
    int year,month,day;
    Time(int y = 0,int m = 0,int d = 0) : year(y),month(m),day(d){}
};
inline bool IsrunYear( int const & year ){

    return ( (year % 4 == 0 && year % 100 != 0 ) || ( year % 400 == 0 ) );
}
// 1.是闰年 0.是平年
int dayOFmonth[2][13] = {
    // 1  2  3  4  5  6  7  8  9  10 11 12
```

```
{0,    31,28,31,30,31,30,31,31,30, 31,30,31},
{0,    31,29,31,30,31,30,31,31,30, 31,30,31}
};


// 输入一个日期，返回是这年的第几天
int Cnt_day_of_year( Time const& time){
    int days(0);
    bool flag = IsrunYear(time.year);
    for (int i = 1;i < time.month;++i )
        days += dayOFmonth[flag][i];
    days += time.day;
    return days ;
}

// 得到这年的第几天这个日期
Time Time_of_nTH_day(int const&year,int days){
    Time time(year,1,1);
    bool flag = IsrunYear( year );
    for ( int i = 1; i <= 12;++i ){
        if( days <= dayOFmonth[flag][i]) break;
        time.month++;
        days -= dayOFmonth[flag][i];
    }
    time.day = days;
    return time;
}

// 这段优化基本没用
int days_Of_year[2222]={0};
inline void sieve(){
    for (int i = 1800 ; i <= 2200;++i )
        days_Of_year[i] = IsrunYear(i) ? 366 : 365;
}

//获取某年的天数
inline int getdayOFyear( int y ){
    return IsrunYear( y ) ? 366 : 365;
}

//计算两个日期相差的天数
int  Cnt_day_between_Times(Time t1,Time t2){
    //2010/3/20-2010/1/10=69
    //printf("%d/%d/%d-
%d/%d/%d=",t1.year,t1.month,t1.day,t2.year,t2.month,t2.day);
    int d1 = 0,d2 = 0;
```

```
    bool flag1 = IsrunYear( t1.year ) ,
         flag2 = IsrunYear( t2.year ) ;
    if ( t1.year == t2.year )
    {
        if (t1.month == t2.month )
            return t1.day-t2.day;
        else{
            d1 = t1.day + dayOFmonth[flag1][t2.month] - t2.day;
            for (int i = t2.month+1 ; i <= t1.month-1;++i)
                d1 += dayOFmonth[flag1][i];
            return d1;
        }
    }else{
        d1 = t1.day;
        for(int i = 1;i <= t1.month-1;i++)
            d1 += dayOFmonth[flag1][i];
        d2 = dayOFmonth[flag2][t2.month] - t2.day;
        for (int i = 12;i >= t2.month+1;--i)
            d2 += dayOFmonth[flag2][i];
        for (int y = t2.year+1;y <= t1.year-1;++y)
            d2 += getdayOFyear( y );
        return d1+d2;
    }
}

// 一个日期n天后的日期
Time next_Days_after_Time( Time time,int days ){
    Time ans(0,0,0);
    int CntDay =  Cnt_day_of_year( time );
    int leave = getdayOFyear(time.year) - CntDay ;
    if( days <= leave ){
        ans = Time_of_nTH_day( time.year, CntDay + days );
    }else {
        days -= leave , time.year++;
        while( days > days_Of_year[time.year] )
            days -= days_Of_year[time.year] ,  time.year++;
        ans = Time_of_nTH_day(time.year,days);
    }
    return ans;
}
```

一年有365天，闰年有366天，所谓闰年，即:能被4整除且不能被100整除的年份，或能被400整除的年份

5. 图论

5.1. 拓扑排序

对一个**有向无环图**(Directed Acyclic Graph简称DAG)G进行拓扑排序，是将G中所有顶点排成一个**线性序列**，使得图中任意一对顶点u和v，**若边**$(u,v) \in E(G)$**，则**u**在线性序列中出现在**v**之前。**通常，这样的线性序列称为满足拓扑次序(Topological Order)的序列，简称拓扑序列。

`toposort()` 可以用来判断有向图中是否有环

```
    static int maxn = 105;
    static int t, n, m;
    static int G[][] = new int[maxn][maxn]; // 存储有向图
    static int topo[] = new int[maxn]; // 拓扑序列
    static int vis[] = new int[maxn]; // 标记该节点状态
    // v[i]=0,该节点尚未被访问;
    // v[i]=1该节点已经被访问，并且还完成了该节点的dfs(i)
    // v[i]=-1表示该节点正在被访问。 这些标志可以用来判断图中是否有环

    static boolean dfs(int u) {
        vis[u] = -1; // 标记当前结点正在被访问
        for (int v = 1; v <= n; v++) {
            if (G[u][v] == 1) {
                if (vis[v] < 0) // 从结点u出发存在有向环
                    return false;
                else if (vis[v] == 0 && !dfs(v)) // 从结点v出发有无相环
                    return false;
            }
        }

        vis[u] = 1; // 标记该节点已经完成dfs(u);
        topo[t--] = u; // 回溯法，因此是首部添加！！！
        return true;
    }

    static boolean toposort() {
        // 构造拓扑序列
        // 判断有向图中是否有环
        t = n;
        Arrays.fill(vis, 0);
        for (int u = 1; u <= n; u++) // 假设图的结点编号从1开始
        {
            if (vis[u] == 0) {
                if (!dfs(u))
                    return false;
            }
        }
```

```
        return true;
    }
```

5.2. Kruskal最小生成树

带权无向图的最小生成树

```
static int maxn = 10000;
    static int n, m;                                    // n条边，m个顶点
    static int u[] = new int[maxn], v[] = new int[maxn], w[] = new
int[maxn];// 无向图的边，以及其权值
    static int p[] = new int[maxn];                     // 用于并查集
    static Integer r[] = new Integer[maxn];     // 根据权值间接排序，第i小的边的
序号保存再r[i]中
                                                        //注意一定要定义为Integer类，因
为Java只能对引用类型
                                                        //使用自定义比较器排序

    //间接比较的比较器
    static Comparator<Integer> comparator = new Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            return Integer.compare(w[o1], w[o2]);
        }
    };
    static int find(int x) {
        if (p[x] == x)
            return x;
        else {
            return p[x] = find(p[x]);
        }
    }

    static int Kruskal() {
         int ans = 0;
        for (int i=0;i<n;i++)
            p[i] = i;           //初始化并查集
        for (int i=0;i<m;i++)
            r[i] = i;           //初始化边的权值排序状态
        Arrays.sort(r, 0, m, comparator);
        for(int i=0;i<m;i++) {
            int e=r[i];
            int x=find(u[e]);
            int y=find(v[e]);
            if(x!=y) {
```

```
                ans+=w[e];
                p[x]=y;
            }
        }
        return ans;
    }
```

5.3. Dijkstra 最短路O(VLogV)

适用于边权为正，单源最短路。单源最短路：从单个源点出发，到所有结点的最短路，该算法**同时适用于有向图与无向图。**

> Dijkstr算法每个结点可以进队多次，但是只能出队一次，出队时**必然完成更新源点到当前点的最短路。**
>
> **反证法可证明：**假设源点是i，当前结点j出队。若$d[j]$不是从i到j的最短路，则必然存在结点k尚未出队，且使得$d[j]=d[k]+w[k][j]$。
>
> 显然$d[k]$肯定小于$d[j]$——没有负权边，根据**Dijkstra的贪心性质——总是将距离当前结点最近的结点入队**，则结点k必然会先于结点j入队，与假设矛盾，得证。

下面算法读入有向边，邻接表存图。

```
public class Dijkstra {
    static class Edge{          //存储边的信息
        int from,to,dist;
        public Edge(int x,int y,int z) {
            from=x;
            to=y;
            dist=z;
        }
    }

    static class Node implements Comparable<Node>{      //结点信息
        int dis,id;         //dis是源点到顶点id的距离
        public Node(int x,int t) {
            dis=x;
            id=t;
        }
        @Override
            public int compareTo(Node o) {
                return Integer.compare(dis, o.dis);
            }
    }
    static int maxn=10000;
```

```
23      int n,m;                                                    //n是顶点数目，m是边数
        目
24      ArrayList<Edge> edges=new ArrayList<>();                    //输入的边的信息列表
25      ArrayList<Integer>[] gra=new ArrayList[maxn];               //邻接表
26      int[] dis=new int[maxn];                                        //记录源点到每个点
        的距离
27      int[] p=new int[maxn];                                      //记录最短路中的上一条
        弧，用于寻找路径。
28
29      void init() {                                               //初始化
30          edges=new ArrayList<>();
31           for(int i=1;i<=n;i++) {                                //顶点编号从1到n
32              gra[i]=new ArrayList<>();
33              dis[i]=Integer.MAX_VALUE;                          //将d[]全部初始化为最
                大值
34          }
35      }
36
37      void addEdge(int form,int to,int w) {
38          edges.add(new Edge(form, to, w));
39          gra[form].add(edges.size()-1);
40      }
41
42      void dijkstra(int s) {
43          dis[s]=0;
44          PriorityQueue<Node> queue=new PriorityQueue<>();
45          queue.add(new Node(0,s));                   //源点入队
46          while(!queue.isEmpty()) {
47              Node cur=queue.poll();
48              int id=cur.id;
49              if(cur.dis!=dis[id])                        //该节点已经完成更新，
50                                                          //某个结点只用被取出一次，
                就能更新彻底
51                  continue;
52
53              //松弛操作,更新结点cur的邻接点距离 源点的距离
54              for(int i=0;i<gra[id].size();i++) {
55                  Edge e=edges.get(gra[cur.id].get(i));
56                  if(dis[e.to]>dis[id]+e.dist) {
57                      dis[e.to]=dis[id]+e.dist;
58                      p[e.to]=gra[id].get(i);                         //记录父节点
59                      queue.add(new Node(dis[e.to], e.to));       //邻接点入队
60                  }
61              }
62          }
63      }
```

```
}
```

5.4. Bellman-Ford最短路径算法O(VE)

与Dijkstra算法不同的是，**该算法适用于含负边（不能含负环）的带权图。**

```
public class BellmanFord {
    static class Edge{          //存储边的信息
        int from,to,dist;
        public Edge(int x,int y,int z) {
            from=x;
            to=y;
            dist=z;
        }
    }

     static int maxn=10000;
     int n,m;                                          //n是结点数目，m是边数目
     ArrayList<Edge> edges=new ArrayList<>();          //输入的边的信息列表
     ArrayList<Integer>[] gra=new ArrayList[maxn];     //邻接表
     int[] dis=new int[maxn];                          //记录源点到每个点的距离
     int[] p=new int[maxn];                            //记录最短路中的上一条弧，用于寻找路径。
     boolean inq[]=new boolean[maxn];                  //记录某结点是否在队列中
     int cnt[]=new int[maxn];                          //记录一个结点在出现在队列中的次数

     void init() {                                     //初始化
        edges=new ArrayList<>();
         for(int i=1;i<=n;i++) {                       //顶点编号从1到n
            gra[i]=new ArrayList<>();
            dis[i]=Integer.MAX_VALUE;                  //将d[]全部初始化为最大值
        }
    }

    void addEdge(int form,int to,int w) {
        edges.add(new Edge(form, to, w));
        gra[form].add(edges.size()-1);
    }

    boolean bellmanFord(int s) {
```

```
        Queue<Integer>queue=new ArrayDeque<>();          //保存进行松弛操作的结点
        dis[s]=0;
        inq[s]=true;
        queue.add(s);
        while(!queue.isEmpty()) {
            int cur=queue.poll();
            inq[cur]=false;

            //松弛操作
            for(int i=0;i<gra[cur].size();i++) {
                Edge e=edges.get(gra[cur].get(i));
                if(dis[cur]<Integer.MAX_VALUE&&dis[e.to]>dis[cur]+e.dist) {
                    dis[e.to]=dis[cur]+e.dist;      //更新cur的邻接点到源点的距离
                    p[e.to]=gra[cur].get(i);
                    if(!inq[e.to]) {
                        queue.add(e.to);            //只将不在队列中的结点入队
                        inq[e.to]=true;             //标记当前节点已经在队列中
                        if(++cnt[e.to]>n)
                            return false;           //一个结点入队多于n次必然有负环
                    }
                }
            }
        }
        return true;
    }
}
```

Floyd_warshell$O(N^3)$

> 是解决**任意两点间的最短路径**的一种，可以**正确处理有向图、无向图或负权（但不可存在负权回路）**的最短路径问题，同时也被用于计算有向图的传递闭包

证明k在外层循环的正确性只需证明下面结论

假设i和j之间的最短路径上的结点集里(不包含i,j),编号最大的一个是x.那么在外循环$k=x$时,$d[i][j]$肯定得到了最小值.

```
public class FloydWarshell {
    static int maxn;
    int dis[][] = new int[maxn][maxn]; // 权值矩阵,初始化为INF,令dis[i][i]=0
    int path[][] = new int[maxn][maxn];//path[i][j]记录i到j最路径上j的前驱点
    int n; // n个结点

    void init() {
        // 权值矩阵预处理，假设结点从0到n-1编号
        Arrays.fill(dis, Integer.MAX_VALUE);
        Arrays.fill(path, 0);
        for (int i = 0; i < n; i++)
            dis[i][i] = 0;
    }

    void floyd() {
        for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (dis[i][k] < Integer.MAX_VALUE && dis[k][j] <
Integer.MAX_VALUE) // 这样避免了溢出问题
            {
                if (dis[i][k] + dis[k][j] < dis[i][j]) {
                    dis[i][j] = dis[i][k] + dis[k][j];
                    path[i][j] = k;
                }
            }
    }
    void output(int i,int j) {
        if(path[i][j]==0) return;

        output(i, path[i][j]);
        System.out.println(path[i][j]);
        output(path[i][j], j);
    }
```

```
34
35 }
```

Floyed求解传递闭包

> 在有向图中，有时不必关心路径的长度，而只关心每两点间是否有通路。则可以用1和0分别表示“连通”和“不连通”。在使用Warshell算法，就可以求得原图的传递闭包

```
1 for(int i=0;i<n;i++) d[i][i]=1;
2 for(int k=0;k<n;k++)
3       for(int i=0;i<n;i++)
4         for(int j=0;j<n;j++)
5             d[i][j]=d[i][j]|(d[i][k]&&d[k][j]);
```