查分+前缀和优化

我们称序列b 1 ,b 2 ,...,b n (n > 1) 的美丽值为，min ∣b i −b j ∣ （1≤i<j≤n），给你一个序列a1, a2 ,...an和一个数字k，请您计算出长度为k的序列a的所有子序列的美丽值之和

分析： 1.若要求美丽值刚好为v的序列不好算，若是要求美丽值大于等于v的序列较为好算，因此这题考虑差分，设F(i)表示美丽值大于等于i的序列个数，则ans = F(1) + F(2) +... F(mx)；而可以根据抽屉原理，得出mx = bmax / (k - 1)
2. 所以可以从1到mx依次计算个数，对于某一个美丽值v，设F(i, j)表示从前i个数中选择j个数的序列(第i个数要取到)，由于数列的顺序不影响答案，所以可以从小到大排序，可以得到F(i， j) = ∑F(k, j - 1) (其中k < i，ak + v ≤ ai),若是按此转移方程复杂度为三次方，考虑到对于满足F(i, j)的k,一定满足F(i+1, j)，因此可以处理好前缀和，记g(i,j)为前i个数中选择j个数的序列(第j个数未必要选择到), 记录p为ap + v <= ai的最大整数，则F(i, j) = g(p, j-1), 而F(i+1,..)的p一定是大于F(i)的p的，所以若是按照前缀和 + 指针的方式，就可以降低复杂度到二次方。 3. 所以有三重循环，第一重v从1到bn / (k-1), 第二重i从2到n(i = 1时美丽值无定义), 第三重j从2到min(i, k), 转移方程为f(i, j) = g(p, j-1), g(i, j) = g(i -1, j) + f(i, j)；对于边界，当p为0时，g(p, j - 1)自然为0，当p为1时，g(1, 1)自然为1，由递推式g(i， 1)为i，或者考虑到g(i， 1)只可能由f(i, 2)才能得到，由于有p个数满足不等式，则g(p,1) = p

```
#include <bits/stdc++.h>

using namespace std;
const int maxn = 1005;
const int mod = 998244353;

int f[maxn][maxn], pre[maxn][maxn], n, k, a[maxn], ans;

int main()
{
    scanf("%d %d", &n, &k);
    for(int i = 1; i <= n; i++) scanf("%d", &a[i]);
    sort(a + 1, a + 1 + n);                                 //先排序


    for(int i = 1; i <= n; i++) pre[i][1] = i;             //确定边界
    for(int v = 1; v * (k - 1) <= a[n]; v++)               //第一重循环美丽值v
    {
        int p = 1;
        for(int i = 2; i <= n; i++)                         //第二重循环选取范围i
        {
            while(a[p] + v <= a[i]) p++;                    //指针移动
            for(int j = 2; j <= min(i, k); j++)             //第二重循环长度j
            {
                f[i][j] = pre[p-1][j-1];
                pre[i][j] = (f[i][j] + pre[i-1][j]) % mod;
            }
        }
        ans = (ans + pre[n][k]) % mod;                     //根据差分思想
        if(ans == 0) break;
    }

    printf("%d\n", ans);
}
```

等价类

喜串的定义:字符串 a 与字符串 b 互为喜串需满足以下两个条件之一:

- a 和 b 相同。
- 将 a 分成 a1 与 a2 两个等长串，b 分成 b1 与 b2 两个等长串，其子串需满足以下两个条件之一:
  - a1 与 b1 互为喜串且 a2 与 b2 互为喜串。
  - a1 与 b2 互为喜串且 a2 与 b1 互为喜串。

喜串是等价，a1 = b1, a2 = b2, a1 = b2, a2 = b1 不可能出现3T1F的情况，不需要四次判断

$T(n) = 3T(n/2) + O(n)$

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;

char a[(1<<18)+1], b[(1<<18)+1];

bool isEqual(int len)
{
    for(int i = 0; i < len; i++)
        if(a[i] != b[i])
            return false;
    return true;
}

void Sort(int L, int R, int len, char* a)
{
    if(!(len & 1))
    {
        Sort(L, L + len / 2, len / 2, a);
        Sort(R, R + len / 2, len / 2, a);
    }
    int p = 0;
    while(p < len && a[L+p] == a[R+p]) p++;
    if(p < len && a[L+p] > a[R+p])
    {
        while(p < len)
        {
            swap(a[L+p], a[R+p]);
            p++;
        }
    }
}

int main()
{
    scanf("%s %s", a, b);
    int len = strlen(a);
    if(!(len & 1))
    {
        Sort(0, len / 2, len / 2, a);
        Sort(0, len / 2, len / 2, b);
    }
    if(isEqual(len)) printf("Yes\n");
    else printf("No\n");
}
```

分类+状压DP

题意：一个序列d由n(n <= 300)位二进制数(0或1)表示，要让这个序列的a[i+m] = a[i], (1 <= m <= n)可以进行两种操作：1.翻转前k*m个数，k为任意整数；2.翻转任意一位的数，问最小的操作数

分析：可以分块分类讨论； 1. 当 m <= sqrt(n) <= 17时，由于m的位数很小，可以枚举最后每一块m的状态，令f(i, j, k)表示从后往前考虑，考虑到第i块状态是j时，k表示是否翻转，需要的最小操作数，然后枚举f(0, j, k)的最小值即为所求。对于最后一块，由于其个数<= m,所以需要单独考虑， j从低位到高位对应于每一块从小到大顺序

2. else 这时 n / m <= sqrt(n) ; 这时每一块比较大，而块数比较小，所以可以考虑枚举每一块是否翻转，j从低到高表示从第1块到最后一块，统计此时的操作数，然后统计第一类操作数(取相应位上0,1个数最少)，取最小答案

```
#include <bits/stdc++.h>

using namespace std;

int d[400], n, m, cnt, tmp1, yu, k0, f[40][1<<17][2], num1[400], num0[400];
int main()
{
    int ans = 999999999;
    char ch = '1';
    while((ch = getchar()) != '\n')   //读取n位数，存在数组d中(从下标0开始)
    {
        if(ch == '1') d[n++] = 1;
        else d[n++] = 0;
    }

    scanf("%d", &m);
    k0 = (n - 1) / m + 1; yu = n % m == 0 ? m : n % m;       //总共有k0块，最后一块有yu
个数
    if(m <= sqrt(n))                                         //每一块个数比较小
    {
        int tmp = (k0 - 1) * m;                             //最后一块在d的起始下标
        for(int j = 0; j < (1 << yu); j++)                  //先枚举yu位数的状态
        {
            for(int w = 0; w < yu; w++)
                if(((j>>w) & 1) ^ d[tmp+w]) f[k0-1][j][0]++;
            f[k0-1][j][1] = yu - f[k0-1][j][0] + 1;
        }
        for(int j = (1 << yu); j < (1 << m); j++)          //再枚举(yu+1)到m位状态的数
        {
            f[k0-1][j][0] = f[k0-1][j%(1<<yu)][0];
            f[k0-1][j][1] = f[k0-1][j%(1<<yu)][1];
        }

        for(int i = k0 - 2; i >= 0; i--)                    //从倒数第二块开始枚举
        {
            tmp -= m;
            for(int j = 0; j < (1 << m); j++)
            {
                for(int w = 0; w < m; w++)
                    if(((j>>w) & 1) ^ d[tmp+w]) f[i][j][0]++;
                f[i][j][1] = m - f[i][j][0];                //相邻两块k不同次数加1
                f[i][j][0] += min(f[i+1][j][1] + 1, f[i+1][j][0]);
```

```
                f[i][j][1] += min(f[i+1][j][0] + 1, f[i+1][j][1]);
            }
        }

        for(int j = 0; j < (1 <<m); j++) ans = min(f[0][j][1], min(ans, f[0][j]
[0]));
        cout<<ans<<endl;
    }
    else                        //块数比较小
    {
        for(int i = 0; i < (1 << k0); i++)           //枚举块数的状态
        {
            int tmp = 0;
            for(int j = 1; j < k0; j++)
                if(((i>>j) & 1) ^ (i>>(j-1) & 1))  tmp++;
            tmp += ((i >>(k0 - 1)) == 1);           //先统计第一类操作数

            memset(num1, 0, sizeof(num1));
            memset(num0, 0, sizeof(num0));
            for(int j = 0; j < n; j++)             //统计第二类操作数
            {
                if((i>>(j/m) & 1) ^ d[j]) num1[j%m]++;    //和0异或是本身，和1异或是翻转
                else num0[j%m]++;
            }
            for(int j = 0; j < m; j++)
                tmp += min(num1[j], num0[j]);
            ans = min(ans, tmp);
        }
        cout<<ans<<endl;
    }
}
```

积性函数:

对于一个定义域为N+的函数f，对于任意两个互质的正整数a,b均满足f(ab)=f(a)f(b) 完全积性函数：对于任意正整数a, b均满足上式

$$f(1)=1,\text{且对于}N=\prod p_i^{a_i},p_i\text{为互不相同的素数}:f(N)=f(\prod p_i^{a_i})=\prod f(p_i^{a_i})$$

两个积性函数的狄利克雷卷积仍是积性函数，任何积性函数都可以线性筛

欧拉函数φ:

小于n的正整数中与n互质的个数

$$1.\text{对于正整数}n,\ \sum\nolimits_{d|n}\varphi(d)=n$$

$$\text{证明：对于}n=1,\varphi(1)=1;\text{一般情况，}n=p_1^{a_1}p_2^{a_2}\dots p_k^{a_k},$$

$$\text{当}n\text{只含一个质数，即}n=p^a,\text{有}\sum_{i=0}^{a}\varphi(p^i)=1+\sum_{i=1}^{a}p^i-p^{i-1}=p^a=n$$

$$\text{当}n=p_1^{a_1}p_2^{a2}\dots p_k^{n_k},\text{由积性函数的性质：}\sum\nolimits_{d|n}\varphi(d)=\sum_{i=0}^{a_1}\varphi(p_1^i)\sum_{i=0}^{a_2}\varphi(p_2^i)\dots\sum_{i=0}^{a_k}\varphi(p_k^i)=p_1^{a_1}p_2^{a2}\dots p_k^{n_k}=n$$

$$2.\text{当}n > 1\text{时，小于}n\text{且与}n\text{互质的数之和}\quad sum = \frac{n * \varphi(n)}{2}$$

```
int pri[maxn], vis[maxn + 10], tot, F[maxn], mx;
void Prim()
{
    F[1] = 1;
    for(int i = 2;i <= mx; i++)
    {
        if(!vis[i])
        {
            pri[tot++] = i;
            F[i] = i - 1;                          //如果是素数
        }
        for(int j = 0; j<tot && i*pri[j]<=mx; j++)
        {
            vis[i*pri[j]] = 1;
            if(i % pri[j] == 0)              //若a为质数, b % a = 0,  F[a*b] =  F[b]
* a
            {
                F[i*pri[j]] = F[i] * pri[j];
                 break;
            }
            else                             //积性函数的性质
                F[i*pri[j]] = F[i] * (pri[j] - 1);
        }
    }
}
```

莫比乌斯函数μ:

$$\mu \;=\; \begin{cases} 1 \quad (n=1) \\ (-1)^k \quad (n = p_1p_2...p_k\ \text{这些因子互不相同}) \\ 0 \quad (else) \end{cases}$$

$$1.\sum_{d|n}\mu(d) \;=\; [n == 1]$$

$$\text{证明：当}n = 1\text{时，等式成立；}$$

$$\text{当}n > 1\text{时，令}n = p_1^{a_1}p_2^{a_2}...p_k^{a_k}\text{，则}\sum_{d|n}\mu(d) = C_k^0 - C_k^1 + C_k^2...+(-1)^kC_k^k = 0$$

$$2.\ \sum_{d|n}\frac{\mu(d)}{d} = \frac{\varphi(n)}{n}$$

$$\text{证明：令}f(n) = n = \sum_{d|n}\varphi(d),\ \text{由莫比乌斯反演，}\varphi(n) = \sum_{d|n} n * \frac{\mu(d)}{d} = n * \sum_{d|n}\frac{\mu(d)}{d}$$

```
int pri[maxn], vis[maxn + 10], tot, u[maxn], mx;
void Prim()
{
    u[1] = 1;
    for(int i = 2;i <= mx; i++)
    {
        if(!vis[i])
        {
            pri[tot++] = i;
            u[i] = -1;                         //质数为-1
```

```
        }
        for(int j = 0; j<tot && i*pri[j]<=mx; j++)
        {
            vis[i*pri[j]] = 1;
            if(i % pri[j] == 0)    //有平方因子
            {
                u[i*pri[j]] = 0;
                 break;
            }
            else                        //多了一个不同的质数换号
                u[i*pri[j]] = -u[i];
        }
    }
}
```

约数个数函数d:

$$若\,n = p_1^{a1} p_2^{a_2} \dots p_k^{a_k},\ \ d(n) = \prod_{i=1}^{k} (a_i + 1)$$

```
int pri[maxn], vis[maxn + 10], tot, d[maxn], cnt[maxn], mx;
//cnt[i]记录每一个数i的最小质因子的指数
void Prim()
{
    d[1] = 1;
    for(int i = 2;i <= mx; i++)
    {
        if(!vis[i])
        {
            pri[tot++] = i;
            d[i] = 2;  cnt[i] = 1;              //质数约数为2
        }
        for(int j = 0; j<tot && i*pri[j]<=mx; j++)
        {
            vis[i*pri[j]] = 1;
            if(i % pri[j] == 0)    //若j为i的最小质数，根据公式
            {
                 d[i*pri[j]] = d[i] / (cnt[i] + 1) * (cnt[i] + 2);
                 cnt[i*pri[j]] = cnt[i] + 1;
                 break;
            }
            d[i*pri[j]] = d[i] << 1;
            cnt[i*pri[j]] = 1;
        }
    }
}
```

约数和函数σ:

$$若\,n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k},\ \ \sigma(n) = \sum_{i=0}^{a_1} p_1^i \sum_{i=0}^{a_2} p_2^i \dots \sum_{i=0}^{a_k} p_k^i$$

```
int pri[maxn], vis[maxn + 10], tot, sigma[maxn], sum[maxn], Mx[maxn], mx;
//sum[i]表示i最小质因数p1的贡献和, Mx[i]表示p1^a1,a1为p1的最高次数
```

```cpp
void Prim()
{
    sigma[1] = 1;
    for(int i = 2;i <= mx; i++)
    {
        if(!vis[i])
        {
            sum[i] = sigma[i] = 1 + i;
            Mx[i] = i;
        }
        for(int j = 0; j<tot && i*pri[j]<=mx; j++)
        {
            vis[i*pri[j]] = 1;
            if(i % pri[j] == 0)    //
            {
                 sigma[i*pri[j]] = sigma[i] / sum[i] * (sum[i] + Mx[i] * pri[j]);
                 sum[i*pri[j]] = sum[i] + Mx[i] * pri[j];
                 Mx[i*pri[j]] = Mx[i] * pri[j];
                 break;
            }
            sigma[i*pri[j]] = sigma[i] * (1 + pri[j]);
            sum[i*pri[j]] = 1 + pri[j];
            Mx[i*pri[j]] = pri[j];
        }
    }
}
```

矩阵快速幂

```cpp
struct mt                                     //N表示方阵的维数
{
    ll a[N][N];
};
mt mult(mt A, mt B, ll mod)                   //计算方阵A * B
{
    mt res;
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            res.a[i][j]=0;
            for(int k = 0; k < N; k++)
            {
                res.a[i][j] += A.a[i][k] * B.a[k][j] % mod;
                res.a[i][j] %= mod;
            }
        }
    }
    return res;
}

mt power(mt a,ll b, ll mod)                  //计算方阵A^b
{
    mt res;
    for(int i = 0; i < N; i++)
```

```
        for(int j = 0; j < N; j++)
            res.a[i][j] = 0;
    for(int i = 0; i < N; i++) res.a[i][i] = 1;    //res为单位矩阵

    while(b)
    {
        if(b & 1) res = mult(res, a, mod);
        b >>= 1;
        a = mult(a, a, mod);
    }
    return res;
}
```

斐波拉契数列

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix} = \begin{bmatrix} f_{n+1} \\ f_{n+2} \end{bmatrix}$$
$$\text{则} \begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}$$

```
ll feb(ll n, ll mod)                //n = 2
{
    mt tmp;
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            tmp.a[i][j] = 0;
    tmp.a[0][1] = tmp.a[1][0] = tmp.a[1][1] = 1;
    tmp = power(tmp, n, mod);
    return (tmp.a[0][0] + tmp.a[0][1] + mod) % mod;
}
```

斐波那契数列数列求和

$$S_n = S_{n-1} + S_{n-2} + 1 \quad S_0 = 1,\ S_1 = 2;$$
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_n \\ S_{n+1} \\ 1 \end{bmatrix} = \begin{bmatrix} S_{n+1} \\ S_{n+2} \\ 1 \end{bmatrix}$$
$$\text{则} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}^n \begin{bmatrix} S_0 \\ S_1 \\ 1 \end{bmatrix} = \begin{bmatrix} S_n \\ S_{n+1} \\ 1 \end{bmatrix}$$

```
ll feb2(ll n, ll mod)
{
    mt tmp;
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            tmp.a[i][j] = 0;
    tmp.a[0][1] = tmp.a[1][0] = tmp.a[1][1] = tmp.a[1][2] = tmp.a[2][2] = 1;
    tmp = power(tmp, n, mod);
    return (tmp.a[0][0] + 2 * tmp.a[0][1] + tmp.a[0][2] + mod) % mod;
}
```

幂函数数求和

$$S_n=\sum_{i=1}^{n} i^k,\text{ 则 } S_n=S_{n-1}+i^k$$

$$S_n=S_{n-1}+(i+1-1)^k=S_{n-1}+C_k^0(n-1)^k+C_k^1(n-1)^{k-1}+\ldots+C_k^k$$

$$\begin{bmatrix} 1 & C_k^0 & C_k^1 & \ldots & C_k^k \\ 0 & C_k^0 & C_k^1 & \ldots & C_k^k \\ 0 & 0 & C_{k-1}^0 & \ldots & C_{k-1}^{k-1} \\ & & \ldots & & \\ & & \ldots & & C_0^0 \end{bmatrix} \begin{bmatrix} S_n \\ n^k \\ \ldots \\ n^0 \end{bmatrix} = \begin{bmatrix} S_{n+1} \\ (n+1)^k \\ \ldots \\ (n+1)^0 \end{bmatrix}$$

$$\text{则 } \begin{bmatrix} S_n \\ n^k \\ \ldots \\ n^0 \end{bmatrix} = \begin{bmatrix} 1 & C_k^0 & C_k^1 & \ldots & C_k^k \\ 0 & C_k^0 & C_k^1 & \ldots & C_k^k \\ 0 & 0 & C_{k-1}^0 & \ldots & C_{k-1}^{k-1} \\ & & \ldots & & \\ & & \ldots & & C_0^0 \end{bmatrix} \begin{bmatrix} S_1 \\ 1^k \\ \ldots \\ 1^0 \end{bmatrix}$$

$$S_n=\sum_{i=1}^{n}(a*i+b)^k,\text{ 则 } S_n=S_{n-1}+(a*n+b)^k$$

$$\begin{bmatrix} 1 & C_k^0 a^k k(a+b)^0 & C_k^1 a^{k-1}(a+b)^1 & \ldots & C_k^k a_0 k(a+b)^k \\ 0 & C_k^0 & C_k^1 & \ldots & C_k^k \\ 0 & 0 & C_{k-1}^0 & \ldots & C_{k-1}^{k-1} \\ & & \ldots & & \\ & & \ldots & & C_0^0 \end{bmatrix} \begin{bmatrix} S_n \\ n^k \\ \ldots \\ n^0 \end{bmatrix} = \begin{bmatrix} S_{n+1} \\ (n+1)^k \\ \ldots \\ (n+1)^0 \end{bmatrix}$$

$$\text{则 } \begin{bmatrix} S_n \\ n^k \\ \ldots \\ n^0 \end{bmatrix} = \begin{bmatrix} 1 & C_k^0 a^k k(a+b)^0 & C_k^1 a^{k-1}(a+b)^1 & \ldots & C_k^k a_0 k(a+b)^k \\ 0 & C_k^0 & C_k^1 & \ldots & C_k^k \\ 0 & 0 & C_{k-1}^0 & \ldots & C_{k-1}^{k-1} \\ & & \ldots & & \\ & & \ldots & & C_0^0 \end{bmatrix} \begin{bmatrix} S_1 \\ 1^k \\ \ldots \\ 1^0 \end{bmatrix}$$

指数函数求和

$$G_{n+1}=a*G_n+b^n$$

$$\begin{bmatrix} a & 1 \\ 0 & b \end{bmatrix} \begin{bmatrix} G_n \\ b^n \end{bmatrix} = \begin{bmatrix} G_{n+1} \\ b^{n+1} \end{bmatrix}$$

指数幂函数复合

$$S_{n+1}=S_n+n^k p^n$$

$$\begin{bmatrix} 1 & C_k^0 & C_k^1 & \ldots & C_k^k \\ 0 & pC_k^0 & pC_k^1 & \ldots & pC_k^k \\ 0 & 0 & pC_{k-1}^0 & \ldots & pC_{k-1}^{k-1} \\ & & \ldots & & \\ & & \ldots & & pC_0^0 \end{bmatrix} \begin{bmatrix} S_n \\ n^k p^n \\ \ldots \\ n^0 p^n \end{bmatrix} = \begin{bmatrix} S_{n+1} \\ (n+1)^k p^{n+1} \\ \ldots \\ (n+1)^0 p^{n+1} \end{bmatrix}$$

判断线段相交与向量方向

向量叉乘： a = (x1, y1), b = (x2, y2) a × b = x1*y2 - y1*x2 a×b,若结果小于0，表示向量b在向量a的顺时针方向；若结果大于0，表示向量b在向量a的逆时针方向；若等于0，表示向量a与向量b平行。(顺逆时针是指两向量平移至起点相连，从某个方向旋转到另一个向量小于180度)

两线段AB, CD相交的充要条件： 1.线段AB与CD所在的直线相交，即点A和点B分别在直线CD的两边； 2.线段CD与AB所在的直线相交，即点C和点D分别在直线AB的两边；

一般情况：如果线段CD的两个端点C和D，与另一条线段的一个端点（A或B，只能是其中一个）连成的向量，与向量AB做叉乘，若结果异号，表示C和D分别在直线AB的两边，若结果同号，则表示CD两点都在AB的一边，则肯定不相交。 特殊情况：1.若有一点相交，则可能有一个值为0； 2.若两个线段在同一直线而相交，则一定有一个值为0； 但是当两条线段所在直线重合而没有交点的情况也是有一个值为0 改进：判断时加入0，同时剔除等于0的不合法情况，即两条线段所在直线重合而没有交点的情况(用矩形来判断)

```
struct Point
{
    int x, y;
};

struct Segment
{
    Point p, q;
};

int det(Point k1, Point k2, Point k3)        //向量k1k2叉乘k1k3
{
    return (k2.x - k1.x) * (k3.y - k1.y) - (k3.x - k1.x) * (k2.y - k1.y);
}

bool isIntersect(Segment k1, Segment k2)
{
    if(max(k1.p.x, k1.q.x) < min(k2.p.x, k2.q.x) || max(k2.p.x, k2.q.x) <
min(k1.p.x, k1.q.x) ||
       max(k1.p.y, k1.q.y) < min(k2.p.y, k2.q.y) || max(k2.p.y, k2.q.y) <
min(k1.p.y, k1.q.y))                                    //首先判断矩形
        return false;
    if(det(k1.p, k2.p, k1.q) * det(k1.p, k1.q, k2.q) >= 0 && det(k2.p, k1.p, k2.q)
* det(k2.p, k2.q, k1.q) >= 0)                           //判断叉乘结果
        return true;
    return false;
}
```

搜索的技巧

可以在r×c的白格子里盖上十字架的章，黑色部分可以重复覆盖，问最少要覆盖几次(不存在则impossible)

5 7
.#.....
####...
.#####.
...####
.....#.

答案为3

分析：每个盖章的中心格子满足自己及其上下左右全为黑，先让所有中心格子盖上，如果这时仍有黑格子没有被覆盖，则答案为impossible, 如果存在解，最外围一圈肯定不存在解，倒数第二圈的中心格子必定要涂上才能保证答案，而对于里面的(5 × 5)个格子，则进行dfs搜索，中间可进行优化。

```
#include <bits/stdc++.h>
```

```
using namespace std;
typedef long long ll;
const int maxn = 30;

int r, c, t, rest, vis[10][10], cross[30][2], cross2[30][2], tol, tol2, ans;
char mp[10][10];

void update(int x, int y, int k)
{
    rest -= k;
    vis[x][y]++;  vis[x+1][y]++;  vis[x-1][y]++;  vis[x][y+1]++;  vis[x][y-1]++;
}

void unupdate(int x, int y, int k)
{
    vis[x][y]--;  vis[x+1][y]--;  vis[x-1][y]--;  vis[x][y+1]--;  vis[x][y-1]--;
    rest += k;
}

void dfs(int id, int cur)                          //现在搜到第id个中心格子，共涂了cur
个格子
{
    if(rest == 0)                                  //若是涂满了
    {
        ans = min(ans, cur);
        return;
    }
    if(id == tol) return;                          //已经搜完
    if((rest - 1) / 4 + 1 + cur > ans) return;                  //可行性剪枝

    int x = cross[id][0], y = cross[id][1], k = (!vis[x][y]) + (!vis[x+1][y]) +
(!vis[x-1][y])
          + (!vis[x][y+1]) + (!vis[x][y-1]);
    if(k == 0) dfs(id + 1, cur);                    //如果这一个格子涂上没有效果，则不
涂
    else if(k <= 2)                                //如果这个格子涂上效果少，先不涂，再
涂
    {
        dfs(id + 1, cur);
        update(x, y, k);
        dfs(id + 1, cur + 1);
        unupdate(x, y, k);
    }
    else                                          //如果这个格子涂上效果大，先涂，再不涂
    {
        update(x, y, k);
        dfs(id + 1, cur + 1);
        unupdate(x, y, k);
        dfs(id + 1, cur);
    }

}
int main()
{
    scanf("%d", &t);
    for(int cas = 1; cas <= t; cas++)
    {
        scanf("%d %d", &r, &c);                            //读入地图
        for(int i = 1; i <= r; i++)  scanf("%s", mp[i] + 1);
```

```
        printf("Image #%d: ", cas);

        rest = 0, tol = 0, tol2 = 0, ans = 0;
        memset(vis, 0, sizeof(vis));
        for(int i = 2; i <= r - 1; i++)                     //先把所有可以涂的中心格子涂满
        {
            for(int j = 2; j <= c - 1; j++)
            {
                if(mp[i][j] == '#' && mp[i-1][j] == '#' && mp[i+1][j] == '#'
                   && mp[i][j+1] == '#' && mp[i][j-1] == '#')
                {
                    if(i != 2 && i != r - 1 && j != 2 && j != c - 1)      //如果不是从外数倒数第二层
                    {
                        cross[tol][0] = i; cross[tol++][1] = j;
                    }
                    else
                    {
                        cross2[tol2][0] = i; cross2[tol2++][1] = j;
                    }
                    vis[i][j] = vis[i-1][j] = vis[i+1][j] = vis[i][j-1] = vis[i][j+1] = 1;
                }
            }
        }

        int flag = 0;                           //统计#个数以及判断是否存在解
        for(int i = 1; i <= r; i++)
        {
            for(int j = 1; j <= c; j++)
            {
                if(mp[i][j] == '#')
                {
                    rest++;
                    if(!vis[i][j])
                    {
                        flag = -1; break;
                    }
                }
            }
            if(flag == -1) break;
        }
        if(flag == -1)                          //如果不存在解
        {
            printf("impossible\n");
            if(cas < t) putchar('\n');
            continue;
        }
        memset(vis, 0, sizeof(vis));            //倒数第二圈必然要全部涂满
        for(int i = 0; i < tol2; i++)
        {
            int k = (!vis[cross2[i][0]][cross2[i][1]]) + (!vis[cross2[i][0]+1][cross2[i][1]]) +
            (!vis[cross2[i][0]-1][cross2[i][1]]) + (!vis[cross2[i][0]][cross2[i][1]+1]) + (!vis[cross2[i][0]][cross2[i][1]-1]);
            update(cross2[i][0], cross2[i][1], k);
        }
        ans = rest;                             //答案的上界
        dfs(0, 0);
```

```
        printf("%d\n", ans + tol2);
        if(cas < t) putchar('\n');
    }
}
```

中国剩余定理

题面： 求解n个同余方程，无解输出-1， 有解输出最小正整数解

思路：扩展欧几里得 + 中国剩余定理

中国剩余定理适用于余数两两互质，改进用归纳法，用满足前i-1个方程的解去求出满足前i个方程的解

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

中国剩余定理适用于余数两两互质，改进用归纳法，用满足前i-1个方程的解去求出满足前i个方程的解

```
ll m[maxn], rest[maxn], n; //M存储商，rest存储余数
ll China()
{
    ll M = m[1], X = rest[1], x, y; //假设此时X已经满足前i-1个等式, 即通解为 X + M * k
    for(int i = 2; i <= n; i++) //求解一个t1满足: X + t1*M ≡ rest[i] (mod m[i])
    {
        ll d = exgcd(M, m[i], x, y); //即 t1*M + t2*m[i] = rest[i] - X，首先判断有无解
        if((rest[i] - X) % d) return -1;
        x = (rest[i] - X) / d * x % m[i]; //原先的x 为 t1*M + t2*m[i] = gcd(M， m[i])
= d的解
        X += M * x; //更新特解X
        M *= m[i] / d; //更新最小公倍数M
        X %= M;
    }
    return (X + M) % M; //求得最小正整数解
}
```

欧拉筛+容斥原理

题面：输出第x小的最小质因数是y的数，如果大于1e9，输出0

思路：1. 31622是平方和小于1e9的最大的数，首先先对[1, 31622]进行欧拉筛，找出素数，如果x = 1,则输出y本身，如果y > 31622, 所求数肯定大于1e9, 输出0。 2. 判断一个区间[1, n]有几个满足条件的数，可以采用容斥原理进行dfs， 具体见代码。 3. 二分答案，L表示[1, L]范围里满足题意的数< x, R表示[1, R]范围里满足题意的数 >= x。

欧拉筛： 对于每一个数（无论质数合数）x，筛掉所有小于x最小质因子的质数乘以x的数。比如对于77,它分解质因数是7*11，那么筛掉所有小于7的质数77，筛掉2*77、3*77、5*77。o(n)

```
//vis用1/0表示是否为质数，pri记录质数
const int maxn = 31622;
const int lim = 1e9;
int pri[maxn], vis[maxn + 10], tot;

void Prim()
{
    for(int i = 2;i <= maxn; i++)
    {
        if(!vis[i])
        pri[tot++] = i;
        for(int j = 0;j < tot; j++)
        {
            if(i * pri[j] > maxn) break; //防止越界
            vis[i*pri[j]] = 1; //筛出小于最小质因子乘x的数
            if(i % pri[j] == 0) break; //若到了最小公因子
        }
    }
}

int dfs(int n, int pos) //计算[1, n]里，不能被pri[0, pos]范围质数整除的数的个数
{
    if(pos == -1) return n;
    if(pri[pos] >= n) return 1; //只有1不行
    return dfs(n, pos - 1) - dfs(n / pri[pos], pos - 1); //容斥原理，考虑一个数就改变符
号一次
}

int main()
{
    int x, y;
    scanf("%d %d", &x, &y);
    if(x == 1) printf("%d\n", y);
    else if(y > maxn) printf("%d\n", 0);
    else
    {
        Prim();
        int cnt = lower_bound(pri, pri + tot, y) - pri; //找到y在pri的下标
        int l = y, r = lim + 1; //在[1,l]范围里小于x， 在[1, r]范围里大于等于x
        while(r - l > 1)
        {
            int mid = (l + r) / 2;
            if(dfs(mid / y, cnt - 1) >= x) r = mid;
            else l = mid;
        }
        if(r > lim) printf("%d\n", 0);
        else printf("%d\n", r);
    }
}
```

给定一个完全二分图，图的左右两边的顶点数目相同，都是 n。我们要给图中的每条边染成红色、蓝色、或者绿色，并使得任意两条红边不共享端点、同时任意两条蓝边也不共享端点。

计算所有满足条件的染色的方案数，并对 10^9+7 取模，其中 $n \le 1e7$。

分析

完全二分图是指左边的每一个点与右边的每一个点都有连线。其实这题涂成绿色可以看成不涂，我们只需要考虑涂成红色和蓝色的情况。

经过简单思考，我们发现只涂一种颜色是比较简单的，若是一边顶点数是 n，则只涂一种颜色的方案数为 $\sum_{i=0}^{n} C_n^i A_n^i$ (左边每个顶点最多只有一条出边染色，若有染色边的顶点数为 i, 有 C_n^i 种选法，由于要**有顺序地映射到右边顶点**，则有 A_n^i 种映射)，我们可以记这个值为 F_n。

而两种颜色就比较复杂了，若是两种颜色选取的方式是独立的，那么答案就是 F_n^2，但实际上左边的一个点与右边的一个点仅有一条连线，不能既涂上红色又涂成蓝色，所以 F_n^2 是算多了，所以我们这时候考虑**虑容斥原理来去重。**

若记有红蓝色涂边重复为事件 P，具体一些 P_{ij}表示左边第 i 个点 和右边第 j 个点选边重复，我们所需要的答案 $ans = F_n^2 - |P|$, 而根据容斥原理:

$P = |P_{11} \cup P_{12} \ldots \cup P_{1n} \ldots \cup P_{nn}| = \sum |P_{ij}| - \sum |P_{i_1j_1} \cap P_{i_2j_2}| \ldots + (-1)^{k+1} \sum |P_{i_1j_1} \cap P_{i_2j_2} \ldots \cap P_{i_kj_k}| + \ldots$

说的通俗一些，红蓝色涂边重复的方案 = 至少有一条重复的方案 = 组合枚举某一条边重复的方案 - 组合枚举某两条边重复的方案 ....=

$C_n^1 A_n^1 F_{n-1}^2 - C_n^2 A_n^2 F_{n-2}^2 \ldots + (-1)^{n+1} C_n^2 A_n^2 F_0^2 = \sum_{i=1}^{n} (-1)^{i+1} C_n^i A_n^i F_{n-i}^2$, 于是可以得到:

$$ans = F_n^2 - |P| = F_n^2 - \sum_{i=1}^{n} (-1)^{i+1} C_n^i A_n^i F_{n-i}^2 = C_n^0 A_n^0 F_n^2 + \sum_{i=1}^{n} (-1)^i C_n^i A_n^i F_{n-i}^2 = \sum_{i=0}^{n} (-1)^i C_n^i A_n^i F_{n-i}^2$$

所以通过容斥原理我们可以得到上面这个比较漂亮的式子，**将两种颜色的组合问题划归到一种颜色的组合问题**，由于组合数可以 $O(n)$ 时间预处理出来，接下来我们好好考虑一种颜色 F_n 该如何处理。

边界易得到 $F_0 = 1, F_1 = 2$，而上面的分析我们也可以得到 $F_n = \sum_{i=0}^{n} C_n^i A_n^i$, 虽然组合数可以预处理出来，但是对于每一个 n 我们都要处理一遍，所以总复杂度还是 $O(n^2)$ 的，所以我们**不能用通项公式，需要再看看递推公式。**从 F_{n-1} 变成 F_n，左右各增加了一个顶点，总共会有五种情况:

①新增的两个点不参与涂色，这样方案数是 F_{n-1}；

②新增的两个点之间涂色，这样方案数是 F_{n-1};

③新增的左边点与右边原来的 $n-1$ 个点涂色，这样方案数为 $(n-1)F_{n-1}$；

④新增的右边点与左边原来的 $n-1$ 个点涂色，这样方案数为 $(n-1)F_{n-1}$；

⑤新增的左边点与右边原来的 $n-1$ 个点涂色 且 新增的右边点与左边原来的 $n-1$ 个点涂色， 即③④的重复计数，这样方案数为 $(n-1)^2 F_{n-2}$；

对于这五种情况的讨论，其实③④⑤也用到了容斥原理，我们可以得到递推式:

$F_n = 2nF_{n-1} - (n-1)^2 F_{n-2}$

这样，F_n 也可以通过 $O(n)$ 时间计算出来，整个算法的复杂度就是 $O(n)$，捋一捋，即分为两步，**先计算一种颜色的情况，再通过容斥原理计算两种颜色的情况**，所以其实三种颜色在这个基础上也是可以继续算的，只不过更加复杂啦。

代码

```
#include <bits/stdc++.h>

using namespace std;
```

```cpp
typedef long long ll;
typedef pair<int, int> P;
const int maxn = 1e7 + 10;
const int INF = 0x3f3f3f3f;
const ll mod = 1e9 + 7;

int inv[maxn], k[maxn], f[maxn];
int n;

int main()
{
    scanf("%d", &n);

    inv[0] = inv[1] = 1;
    for(int i = 2; i <= n; i++)                  //预处理逆元
        inv[i] = 1LL * (mod - mod / i) * inv[mod % i] % mod;
    for(int i = 2; i <= n; i++)                  //inv[i] 现在表示 i! 的逆元
        inv[i] = 1LL * inv[i-1] * inv[i] % mod;
    int fac = 1;
    for(int i = 2; i <= n; i++)                   //fac = n!
        fac = 1LL * fac * i % mod;

    k[0] = 1;                                      //计算组合数 A_n^i C_n^i
    for(int i = 1; i <= n; i++)
    {
        int tmp = 1LL * fac * inv[n-i] % mod;
        k[i] = 1LL * tmp * tmp % mod * inv[i] % mod;
    }

    f[0] = 1, f[1] = 2;                          //计算一种颜色的情况
    for(int i = 2; i <= n; i++)
        f[i] = (2LL * i * f[i-1] - 1LL * (i - 1) * (i - 1) % mod * f[i-2]) % mod;

    ll ans = 0;                                //用容斥原理计算两种颜色的情况
    for(int i = 0; i <= n; i++)
    {
        if(i & 1)                                 //奇数为符号
            ans = (ans - 1LL * k[i] * f[n-i] % mod * f[n-i] % mod) % mod;
        else
            ans = (ans + 1LL * k[i] * f[n-i] % mod * f[n-i] % mod) % mod;
    }
    if(ans < 0)
        ans = ans + mod;
    printf("%lld\n", ans);
}
```

逆元的计算

有一颗树，树有n个结点。有k种不同颜色的染料给树染色。一个染色方案是合法的，当且仅当对于所有相同颜色的点对(x,y)，x到y的路径上的所有点的颜色都要与x和y相同。请统计方案数。

其中，$n,k \leq 300$

分析

若是从某个根节点在dfs 或者 bfs 过程中统计，是非常麻烦的事。子结点和父亲一个颜色，这种情况还比较简单，若是不同颜色，则其兄弟结点也不能和该子结点一个颜色(因为这两者通过父结点连接)，这样整个过程就不好计算了。

或许可以换一种思路，若是我们已经涂了一个部分，并且这些部分是联通的，即涂了这棵树一棵结点数为 i 的子树，这棵树共有 j 种不同颜色，那么可以记涂法为 $dp[i][j]$, 若是已经涂过颜色的 v_p 与没有涂过颜色的 v_q 相连，那么可以扩展，若 $Color(v_p) = Color(v_q)$，那么 v_q 有一种涂法；若是 $Color(v_p) \neq Color(v_q)$，那么 v_q 有 $k-j$ 种涂法(已经涂过的 j 种颜色不能再用了，因为 v_q 与已经涂过颜色的结点之间的路径必须经过 v_p)。于是可以得到如下转移式子：

$$dp[i][j] = dp[i-1][j] + dp[i-1][j-1] * (k-j+1)$$

而我们所需要的答案就是 $\sum_{i=1}^{n} dp[n][i]$，复杂度为 $O(nk)$ 所以通过动态规划得出答案就可以。

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
typedef pair<int, long long> P;
const int maxn = 1e5 + 10;
const int INF = 0x3f3f3f3f;
const ll mod = 1e9 + 7;

ll dp[302][302], ans;
int n, k;

int main()
{
    scanf("%d %d", &n, &k);
    dp[1][1] = k;                                //一个结点涂一种颜色有k种答案
    for(int i = 2; i <= n; i++)
        for(int j = 1; j <= k; j++)               //进行dp
        {
            dp[i][j] = dp[i-1][j] + dp[i-1][j-1] * (k - j + 1);
            dp[i][j] %= mod;
        }
    for(int i = 1; i <= k; i++)                   //累加答案
        ans = (ans + dp[n][i]) % mod;
    printf("%lld\n", ans);
}
```

或许还可以这样想，相同颜色的结点即构成一个连通块，而对于树来说，每切去一条边就多一个连通块，最多可以切去 $n-1$ 条边，形成 n 个连通块。所以一棵树我们有 $C_{n-1}^{i-1} (1 \leq i \leq n)$ 种切法，每次对于得到的 i 个连通块，用 k 种颜色去涂，总共有 A_k^i 种涂法，所以最终答案为：

$\sum_n^i C_{n-1}^{i-1} A_k^i$

复习一下组合数学的知识， $C_m^n = \frac{m!}{(m-n)!n!}$, $A_m^n = \frac{m!}{(m-n)!}$， 其中阶乘我们可以先预处理出来，但这里有除法的取模，我们需要用到**费马小定理: 对于质数** p, **若** a **不是** p **的倍数，则有** $a^{p-1} \equiv 1 (mod\ p)$

那么若记 $a^{-1} \equiv b (mod\ p)$, 那么 $a^{-1} * a^{p-1} \equiv b * a^{p-1} (mod\ p)$, 又由于费马小定理可得 $b * a^{p-1} = b (mod\ p)$， 于是 $a^{p-2} \equiv b (mod\ p)$， 这样就**将一个求负幂次的形式变成了求正幂次的形式**，这样我们就可以用快速幂 $O(logn)$ 的时间进行计算了，这样总复杂度就是 $O(nlogn)$

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
typedef pair<int, long long> P;
const int maxn = 1e5 + 10;
```

```
const int INF = 0x3f3f3f3f;
const ll mod = 1e9 + 7;

ll fac[302], ans;
int n, k;

ll qpow(ll x, ll n)                 //快速幂
{
    ll t = 1;
    for (; n; n >>= 1, x = x * x % mod)
        if (n & 1)
            t = t * x % mod;
    return t;
}

ll C(int n, int k)
{
    return fac[n] * qpow(fac[k] * fac[n-k] % mod, mod - 2) % mod;
}

ll A(int n, int k)
{
    return fac[n] * qpow(fac[n-k] % mod, mod - 2) % mod;
}

int main()
{
    scanf("%d %d", &n, &k);

    fac[0] = fac[1] = 1;
    for(int i = 2; i <= n; i++)           //预处理阶乘
        fac[i] = fac[i-1] * i % mod;

    for(int i = 1; i <= min(n, k); i++)
        ans = (ans + C(n - 1, i - 1) * A(k, i) % mod) % mod;
    printf("%lld\n", ans);
}
```

上面用快速幂求逆元用了对数时间，而其实可以先用线性时间先预处理出逆元，这样总复杂度就可以降到 $O(n)$。i^{-1} 在模 p 运算下如何得出呢?

不妨记 $p = k * i + r$, 其中 $r < i$，$1 < i < p$, 那么 $k = \lfloor \frac{p}{i} \rfloor$，$r = p\%i$, 可以进行如下推导:

$k * i + r \equiv 0 (mod\, p)$

两边同乘 $r^{-1} i^{-1}$:

$k * r^{-1} + i^{-1} \equiv 0 (mod\, p)$

$i^{-1} \equiv -k * r^{-1} (mod\, p)$

$i^{-1} \equiv -\lfloor \frac{p}{i} \rfloor * (p\%i)^{-1} (mod\, p)$

若记 $inv[i]$ 表示 i 的逆元，那么 $inv[i] = -p/i * inv[p\%i]$, 调整一下符号，有 $inv[i] = (p - p/i) * inv[p\%i]$, 然后算组合数的时候由于求的是阶乘，所以还要前缀和处理一下。

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
typedef pair<int, long long> P;
const int maxn = 1e5 + 10;
const int INF = 0x3f3f3f3f;
```

```
const ll mod = 1e9 + 7;

ll inv[302], fac[302], ans;
int n, k;

ll A(int n, int k)
{
    return fac[n] * inv[n-k] % mod;
}

ll C(int n, int k)
{
    return fac[n] * inv[k] % mod * inv[n-k] % mod;
}

int main()
{
    scanf("%d %d", &n, &k);

    inv[0] = inv[1] = 1;
    fac[0] = fac[1] = 1;
    for(int i = 2; i <= n; i++)           //求逆元和阶乘
    {
        inv[i] = (mod - mod / i) * inv[mod % i] % mod;
        fac[i] = fac[i-1] * i % mod;
    }
    for(int i = 2; i <= n; i++)               //对阶乘进行前缀和处理
        inv[i] = inv[i] * inv[i-1] % mod;

    for(int i = 1; i <= min(n, k); i++)
        ans = (ans + C(n - 1, i - 1) * A(k, i) % mod) % mod;
    printf("%lld\n", ans);
}
```

kmp + Hash

题意是说，定义了两个字符串间的函数 $f(s,t)$ 表示字符串 s 的前缀和字符串 t 后缀能相等的最大长度，而总共有 n 个串，求 $\sum_{i=1}^{n}\sum_{j=1}^{n}f(s_i,s_j)$。其中 $1\le n\le 1e5, \sum|s_i|\le 1e6$。

比赛中这题过的人不是很多，感觉也没有很巧的解法。暴力的话，我们可以先将每个字符串的后缀处理出来，由于 $\sum|s_i|\le 1e6$，那么最多有 $1e6$ 个后缀，然后再遍历前缀来和存下来的后缀匹配。但是这样的话一定是会重复的，比如只有一个串 aba，我存的后缀有 a, ba, aba， 那么我遍历前缀的时候 a 和 aba 都可以匹配到，但是我们只需要长度最大的那个，这该如何去重呢?

我们可以再举一个例子来看一看，比如 $ababa$ 与自身匹配，进行遍历前缀时:

①a 匹配到，所以 $ans[1]++$(记录个数);

②ab匹配不到;

③aba 匹配得到，所以我们现在知道 a 会重复，所以 $ans[1]--, ans[3]++$;

④$abab$ 匹配不到;

⑤$ababa$ 匹配到，所以我们现在知道 aba 重复，所以$ans[3]--, ans[5]++$。

所以我们**每匹配到一个前缀，就要减去其能相等的最大长度的前后缀的计数(不包括自身)，而这正好就是 kmp 算法里的 next 数组(或者叫 fail 失败链接)。**

而存后缀的话，翻了翻 AC 代码，大部分人都是通过函数 $\sum_{i=0}^{len-1}s[i]*131^i$ 将字符串转为 unsigned long long, 然后用 map 进行映射，所以我也采取了这种哈希方式，这样的映射**稍微长一点的字符串肯定会自然溢出，但是相等的字符串一定能映射成相同的数值。**

需要注意的是，用普通的 map 花了 $2.4s$, 而用 unordered_map 花了 $0.9s$，所以如果卡 map 的常数，**一定要用 unordered_map**。

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
typedef unsigned long long ull;
const int maxn = 1e5 + 10;
const int maxm = 1e6 + 10;
const ll mod = 998244353;

int nxt[maxm], n;
ll ans, num[maxm];
string p[maxn];
unordered_map<ull, int> mp;

void calnext(int k)                    //nxt[i] 表示字符串下标[0, i-1](即长度为i)的最长前后缀相等长度(不包括自身)
{
    nxt[0] = -1;
    int i = 1, j;
    while(p[k][i-1])
    {
        j = nxt[i-1];
        while(j != -1 && p[k][j] != p[k][i-1])
            j = nxt[j];
        nxt[i] = j + 1;
        i++;
    }
}

int main()
{
    cin>>n;
    for(int i = 1; i <= n; i++)
        cin>>p[i];

    for(int i = 1; i <= n; i++)
    {
        ull cur = 0, base = 1;
        for(int j = p[i].size() - 1; j >= 0; j--)        //计算哈希值
        {
            cur = cur + base * p[i][j];
            base *= 131;
            mp[cur]++;                         //用map给后缀计数
        }
    }

    for(int i = 1; i <= n; i++)
    {
        calnext(i);
        ll cur = 0;
        for(int j = 0; p[i][j]; j++)
        {
            cur = cur * 131 + p[i][j];          //计算哈希值
            num[j+1] = mp[cur];
```

```
            if(nxt[j+1]) num[nxt[j+1]] -= num[j+1];         //去重
        }
        for(int j = 1; p[i][j-1]; j++)                   //计算题目所需要的那个值
            ans += num[j] * j % mod * j % mod, ans %= mod;
    }

    cout<<ans<<endl;
}
   
```

单调队列

```
#include <bits/stdc++.h>
using namespace std;
int a[1000010];
int du[1000010];
int n, m;
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++)
    {
        scanf("%d", &a[i]);
    }
    int l = 0;
    int r = 1;              //区间从l开始，下一个要加入的位置是r
    du[0] = 1;
    if (m == 1) printf("%d ", a[1]);
    for (int i = 2; i <= n; i++)             //最大值
    {
        if (i - du[l] >= m && (l < r)) l++;

        while (r > l && a[du[r - 1]] >= a[i]) r--;
        du[r++] = i;
        if (i >= m) printf("%d ", a[du[l]]);
    }
    printf("\n");
    l = 0;
    r = 1;
    du[0] = 1;
    if (m == 1) printf("%d ", a[1]);
    for (int i = 2; i <= n; i++)                  //最小值
    {
        if (i - du[l] >= m && (l < r)) l++;

        while (r > l && a[du[r - 1]] <= a[i]) r--;
        du[r++] = i;
        if (i >= m) printf("%d ", a[du[l]]);
    }
    printf("\n");
    return 0;

}
```

bitset神奇用法

题目大意是给定长度为 n 的序列 A 和长度为 m 的序列 B，其中 $n \geq m$，A中可以截取长度为 m 的连续区间 S，问满足对于任意 $1 \leq i \leq m$，都有 $S_i \geq B_i$ 的区间个数。其中 $m \leq 4e4$，$n \leq 1.5e5$。

看了好久才看懂标程......首先为了之后的状态转移，对于每一个 A_i，都有一个 bitset $I[i]$，若 $I[i][j] = 1$, 表示 $A_i \geq B_j$, 反之为 0 表示 $A_i < B_j$。若是单纯的暴力匹配得 bitset 的值，我们的空间复杂度和时间复杂度都是 $O(\frac{nm}{64})$, 所以我们**可以考虑先排序，用双指针的方式进行遍历。**因为若 $A_{i_1} \leq A_{i_2}$，那么 $I[i_1]$ 为 1 的地方，$I[i_2]$ 也一定为1, 所以排序之后，后一个数的 bitset 可以在前一个数 bitset 的基础上进行修改。

而其实我们也不需要 n 个 bitset, 因为序列 B 长度为 m，所以最多有 $m+1$ 个不同的 bitset, 这样空间复杂度可以降到 $O(\frac{m^2}{64})$。

接下来比较重要的就是这个转移方法了，我们用一个长度为 $m+1$ 的 bitset，来记录状态，**若第 i 位为 1，表示当前可以匹配到 序列B 的前 i 位，否则表示没有匹配到。**这个还是需要例子来说明，若 $A = [1,2,2,3,5], B = [1,2,3]$, bitset 初始为 $[1,0,0,0]$(第 0 位为 1 表示可以匹配到的区间长度为0)

①当前匹配到区间长度为0，我们尝试去扩展区间，由于 $A[1] \geq B[1]$，bitset 变成 $[1,1,0,0]$;

②当前匹配到区间长度为0或1，我们尝试去扩展区间，由于 $A[2] \geq B[1]$，$A[2] \geq B[2]$，bitset 变成 $[1,1,1,0]$

③当前匹配到区间长度为0或1或2，我们尝试去扩展区间，由于 $A[3] \geq B[1]$，$A[3] \geq B[2]$，但 $A[3] < B[3]$，bitset 变成 $[1,1,1,0]$

④当前匹配到区间长度为0或1或2，我们尝试去扩展区间，由于 $A[4] \geq B[1]$，$A[4] \geq B[2]$，$A[4] \geq B[3]$,bitset 变成 $[1,1,1,1]$，答案+1；

④当前匹配到区间长度为0或1或2，我们尝试去扩展区间，由于 $A[5] \geq B[1]$，$A[5] \geq B[2]$，$A[5] \geq B[3]$,bitset 变成 $[1,1,1,1]$，答案+1；

扩展区间操作相当于将当前 bitset 向左移一位，然后与 $I[i]$ 进行与操作：$cur = (cur << 1) \& I[i]$

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
const int maxn = 1.5e5 + 10;
const int maxm = 4e4 + 10;

bitset<40010> I[40010];
int n, m, a[maxn], b[maxm], k1[maxn], k2[maxm], mark[maxn], tol, ans;

int main()
{
    scanf("%d %d", &n, &m);
    for(int i = 1; i <= n; i++)
    {
        scanf("%d", &a[i]);
        k1[i] = i;
    }
    for(int i = 1; i <= m; i++)
    {
        scanf("%d", &b[i]);
        k2[i] = i;
    }

    sort(k1 + 1, k1 + 1 + n, [](int x, int y){return a[x] < a[y];});     //k1[i] 表示a 数组第i小的数的下标
    sort(k2 + 1, k2 + 1 + m, [](int x, int y){return b[x] < b[y];});

    bitset<40010> tmp;
    int p = 1, flag;
```

```
    for(int i = 1; i <= n; i++)
    {
        flag = 0;
        while(p <= m && a[k1[i]] >= b[k2[p]])      //双指针标记bitset
        {
            tmp.set(k2[p]);
            p++;
            flag = 1;
        }
        if(flag) I[++tol] = tmp;
        mark[k1[i]] = tol;
    }

    bitset<40010> cur;
    for(int i = 1; i <= n; i++)
    {
        cur.set(0);                        //第0位始终为1，因为总可以从区间长度为0开始扩展
        cur = (cur<<1) & I[mark[i]];
        if(cur[m] == 1)
            ans++;
    }

    printf("%d\n", ans);
}
```

然后翻别的大佬的 AC 代码，看到一个只用两个 bitset 就过了的...太神仙了，理解了好久，不太能写出来......大致思路是设置一个长度为 n 的 bitset, 一开始全部初始化为 1，第 i 位为1表示从当前开始长度为 m 的区间满足要求，然后通过从大到小遍历来去掉不可能的位置。这内存压的太nb了。

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
typedef unsigned long long ull;
typedef pair<double, double> P;
const int maxn = 1.5e5 + 10;
const int maxm = 4e4 + 10;
const int INF = 0x3f3f3f3f;
const double eps = 1e-11;
const ll mod = 998244353;

bitset<maxn> tmp, cur;
int n, m, a[maxn], b[maxm], k1[maxn], k2[maxm];

int main()
{
    scanf("%d %d", &n, &m);
    for(int i = 1; i <= n; i++)
    {
        scanf("%d", &a[i]);
        k1[i] = i;
        cur.set(i);      //全部初始化成1
    }
    for(int i = 1; i <= m; i++)
    {
        scanf("%d", &b[i]);
        k2[i] = i;
    }
```

```
    sort(k1 + 1, k1 + 1 + n, [](int x, int y){return a[x] > a[y];});
    sort(k2 + 1, k2 + 1 + m, [](int x, int y){return b[x] > b[y];});

    int p = 1;
    for(int i = 1; i <= m; i++)          //从大到小遍历
    {
        while(p <= n && a[k1[p]] >= b[k2[i]])
            tmp.set(k1[p++]);
        cur &= tmp >> (k2[i] - 1);        //不满足条件的位置会被置为0
    }

    printf("%d\n", cur.count());
}
```

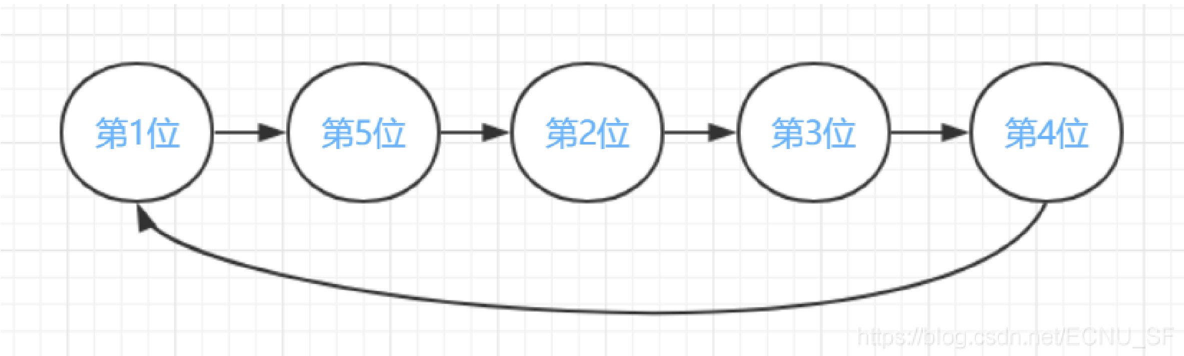
群论

题意是说一开始给你一个排列 $\{1,2,3...n\}$，经过 k 次置换，变成 $\{a_1,a_2....a_n\}$，问若是将原来的排列只置换一次，会变成什么？($1 \le n \le 1e5$, k 为质数)

首先什么是排列的置换呢？**根据抽象代数里的定义，一个集合的排列置换是自身对自身的一个双射。**这可以理解成将一个排列的元素打乱顺序，得到一个新的排列。比如排列 $\{1,2,3,4,5\}$ 可以经过 k 次置换变成了 $\{5,3,4,1,2\}$。

而排列是可以分解成若干 cycle 的。比如上面的排列，原来的第 1 位 变成了原来的第 5 位， 第 5 位变成了原来的第 2 位，第 2 位变成了原来的第 3 位，第 3 位变成了原来的第 4 位，第 4 位变成了原来的第 1 位，这样就可以写作一个 cycle ：$(1,5,2,3,4)$。再比如 k 次置换变成了 $\{5,3,4,2,1\}$, 可以写作 2 个 cycle：$(1,5)(2,3,4)$。

我们可以发现，若经过 k 次置换变成了 $\{5,3,4,1,2\}$，那么经过 $5k$ 次置换可以变回 $\{1,2,3,4,5\}$ 。其实 k 次置换可以看成一个双射函数，而 tk 次置换就是一个复合函数，接下来就推一推。



k次变换对应的 cycle 即为上图，以第 1 位为例，一开始是 1:

①经过 k 次变换后变成了原来的第 5 位，所以 k 次变换后为 5;

②经过 $2k$ 次变换后变成了 k次变换后的 第 5 位，即没有变换时的第 2 位，所以 $2k$ 次变换后为 2;

③经过 $3k$ 次变换后变成了 $2k$次变换后的 第 5 位，即k次变换后的 第 2 位，没有变换时的第 3 位，所以 $3k$ 次变换后为 3;

④经过 $4k$ 次变换后变成了 $3k$次变换后的 第 5 位，即$2k$次变换后的 第 2 位，k次变换后的 第 3 位，没有变换时的第 4 位，所以 $4k$ 次变换后为 4;

⑤经过 $5k$ 次变换后又回到了 1。

若记上述的 cycle 的变换关系为 $b[] = \{1,5,2,3,4\}$，那么 经过 tk 次变换后$a[1] = b[(t+1)\%\ 5]$，**推广一下可以得到 经过 tk 次变换后:** $a[i] = b[(t+i)\%\ len(cycle)]$。

我们在这里可以知道，**一个 cycle 的元素要返回原来的位置，要经过 $len(cycle) * k$ 次变换**；若得到所有 cycle 的 lcm, 那么整个排列要返回自身就是 $lcm * k$ 次变换。

当我们知道了一个 cycle tk 次变化后对应到什么，那么对于一个排列可以分解为若干 cycle, 分开处理即可。**我们最终要求的是置换 1 次的结果，即对于每一个 cycle，长度为 l_i, 都进行 $t_i k$ 次变换，其中 $t_i k \equiv 1\ (mod\ l_i)$, 即 k 对于 l_i 的逆元**，若是有一个同余方程无解，则说明解不存在，但是由于 k 为质数，所以 t_i 肯定有解，可以通过枚举或者扩展欧几里得的方法得到 t_i。

由于对于每一个 cycle 都可以线性时间得到逆元和进行置换，所以总复杂度为 $O(n)$

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
const int maxn = 1e5 + 10;
const int INF = 0x3f3f3f3f;

int ans[maxn], a[maxn], visit[maxn];
int n, k;
int main()
{
    scanf("%d %d", &n, &k);
    for(int i = 1; i <= n; i++)
        scanf("%d", &a[i]);

    for(int i = 1; i <= n; i++)
    {
        if(visit[i]) continue;          //通过 dfs 得到每个 cycle，记录变换关系

        vector<int> v;
        int j = i, inv;
        while(!visit[j])
        {
            v.push_back(j);
            visit[j] = 1;
            j = a[j];
        }
        for(inv = 1; inv < v.size(); inv++)                     //找到相应的逆元
            if(1LL * inv * k % v.size() == 1)  break;
        for(int h = 0; h < v.size(); h++)                       //根据推得的结果进行变换
            ans[v[h]] = v[(h+inv)%v.size()];
    }

    for(int i = 1; i < n; i++)
        printf("%d ", ans[i]);
    printf("%d\n", ans[n]);
}
```

STL中list的使用

题意是说对于一张图有 n 个点，m 条边，一开始每个点自为一个 $group$，我们现在有 q 次操作，每次操作选定 $group$ 的编号为 o_i，若是 o_i 组没有点，则不发生变化，否则与 o_i 组的点相连的点若是属于其他组 o_j，则 o_j 组的点并入 o_i， o_j 变空。全部操作之后输出每个点属于那一组。有多组输入，$n, m, q \leq 5e8$

。

经过分析我们发现，若是 o_i 组在一轮操作中被并掉，那么 o_i 组就一直为空；若是一个点在一次操作中和另一个点处于同一组，那么之后它们永远属于同一组。于是自然想到**用并查集记录每个 $group$ 的元素，并记录与每个 $group$ 相连的点。**

一开始想到用 $vector$ 记录，但是这样的话一个 $group$ 被并入另一个 $group$ 时，它所记录的相连的点也要相应复制到另一个 $group$，这样就会进行很多次复制。查看别人的题解发现有链式前向星的做法，只要修改一下链表，而不用复制。但是前者复制反而会过，后者只是修改链表却超时....分析原因可能是链式前向星要初始化太多东西，多组输入就容易超时。

为此，**可以采用 $list$ 来记录，$list$ 有一个 $splice$ 方法，可以完成两个链表的合并，这种方式比链式前向星写的手动链表要快。**

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
typedef unsigned long long ull;
typedef pair<double, double> P;
const int maxn = 8e5 + 10;
const int INF = 0x3f3f3f3f;
const double eps = 1e-11;
const ll mod = 1e9 + 7;

list<int> G[maxn];
int fa[maxn];

int Find(int x) { return fa[x] == x? x : fa[x] = Find(fa[x]); }

int main()
{
    int t, n, m, q, x;
    scanf("%d", &t);
    while(t--)
    {
        scanf("%d %d", &n, &m);
        for(int i = 0; i < n; i++)
        {
            fa[i] = i;               //初始化并查集和链表
            G[i].clear();
        }

        for(int i = 0; i < m; i++)
        {
            int x, y;
            scanf("%d %d", &x, &y);
            G[x].push_back(y);  G[y].push_back(x);
        }

        scanf("%d", &q);
        while(q--)
        {
            scanf("%d", &x);
            if(fa[x] != x)  continue;              //若是该组已经没有点
            int len = G[x].size(), y;
            for(int i = 0; i < len; i++)
            {
                y = G[x].front();  G[x].pop_front();
                y = Find(y);              //查找与这个组相邻的点
                if(y == x) continue;          //若是同属于一个组
```

```
                fa[y] = x;
                G[x].splice(G[x].end(), G[y]);           //修改链表
            }
        }

        for(int i = 0; i < n - 1; i++)
            printf("%d ", Find(i));
        printf("%d\n", Find(n - 1));
    }
}
```