CoinIn 的板子

基础算法

二分

mid=l+r>>1

```
while(l<r)
{
    int mid=l+r>>1;
    if(check(mid)) r=mid;
    else l=mid+1;
}
```

mid=l+r+1>>1

```
while(l<r)
{
    int mid=l+r+1>>1;
    if(check(mid)) l=mid;
    else r=mid-1;
}
```

三分

```
while(r-l>eps)
{
    double lmid=(l+l+r)/3.0;
    double rmid=(l+r+r)/3.0;
    if(f(lmid)<f(rmid)) l=lmid;
    else r=rmid;
}
```

二维前缀和

```
// 法一：容斥
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
       a[i][j]=a[i-1][j]+a[i][j-1]-a[i-1][j-1];

// 法二：模拟
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            a[i][j] += a[i - 1][j];
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            a[i][j] += a[i][j - 1];
```

三维前缀和

```
// 法一：容斥
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            for (int k = 1; k <= n; k++) {
                a[i][j][k] += a[i][j][k - 1] + a[i][j - 1][k] + a[i - 1][j][k];
                a[i][j][k] -= a[i][j - 1][k - 1] + a[i - 1][j - 1][k] + a[i - 1][j][k - 1];
                a[i][j][k] += a[i - 1][j - 1][k - 1];
            }
        }
    }



// 法二：模拟
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
```

```
            for (int k = 1; k <= n; k++) {
                a[i][j][k] += a[i][j][k - 1];
            }
        }
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            for (int k = 1; k <= n; k++) {
                a[i][j][k] += a[i][j - 1][k];
            }
        }
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            for (int k = 1; k <= n; k++) {
                a[i][j][k] += a[i - 1][j][k];
            }
        }
    }
```

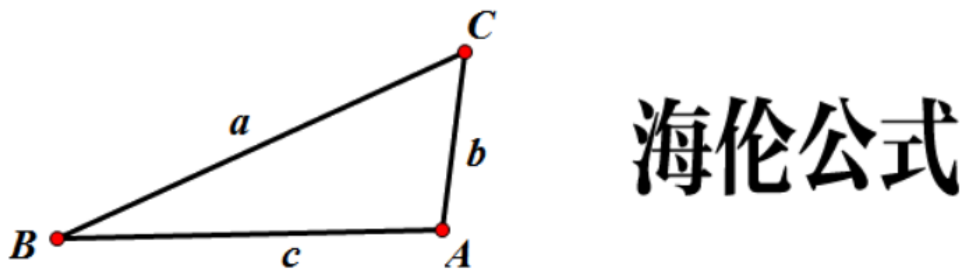
```
ll fa[N];
ll findx(ll x)
{
    if(fa[x]==x) return x;
    return fa[x]=findx(fa[x]);
}
bool merge(ll x,ll y)
{
    ll xx=findx(x),yy=findx(y);
    if(xx==yy) return false;
    fa[xx]=yy;
    return true;
}
```

常用公式

平方和公式：

$$1^2+2^2+3^2+\cdots+n^2=\frac{n(n+1)(2n+1)}{6}$$

海伦公式：



$$S=\sqrt{p(p-a)(p-b)(p-c)}\quad 将\quad p=\frac{1}{2}(a+b+c)\quad 代入$$

$$S=\frac{1}{4}\sqrt{(a+b+c)(a+b-c)(a+c-b)(b+c-a)}$$

数据结构

单调队列：

```
int main()
{
    int n,m,h,t;
    cin>>n>>m;
    for(int i=1;i<=n;i++) scanf("%d",&(a[i]));
    h=1,t=0;
    for(int i=1;i<=n;i++)//求 m 区间内最小值 以 i 为末尾的 m 区间   维护增队列
    {
```

```
        while(h<=t&&a[q[t]]>=a[i]) t--;//删去队尾的无用元素
        q[++t]=i;
        while(i-q[h]+1>=m+1) h++;//队头删去在所需区间外的元素
        if(i>=m)
            ans1[i-m+1]=a[q[h]];
    }
    h=1,t=0;
    for(int i=1;i<=n;i++)//求 m 区间内最大值    以 i 为末尾的 m 区间
    {
        while(h<=t&&a[q[t]]<=a[i]) t--;//删去队尾的无用元素
        q[++t]=i;
        while(i-q[h]+1>=m+1) h++;//队头删去在所需区间外的元素
        if(i>=m)
            ans2[i-m+1]=a[q[h]];
    }
    for(int i=1;i+m-1<=n;i++) printf("%d ",ans1[i]);
    cout<<endl;
    for(int i=1;i+m-1<=n;i++) printf("%d ",ans2[i]);
    cout<<endl;
    return 0;
}
```

主席树

(区间第 k 小)

```
struct segment{
    ll lc,rc,sum;
    #define lc(x) tree[x].lc
    #define rc(x) tree[x].rc
    #define sum(x) tree[x].sum
}tree[N];ll idx,root[N];
// build 的作用为初始化，可不写
//ll build(ll l,ll r)
//{
//  ll p=++idx;
```

```
//  if(l==r)
//  {
//      sum(p)=0;
//      return p;
//  }
//  ll mid=l+r>>1;
//  lc(p)=build(l,mid);rc(p)=build(mid+1,r);
//  sum(p)=sum(lc(p))+sum(rc(p));
//  return p;
//}
ll insert(ll now,ll l,ll r,ll x)
{
    ll p=++idx;
    tree[p]=tree[now];
    if(l==r)
    {
        sum(p)++;
        return p;
    }
    ll mid=l+r>>1;
    if(x<=mid) lc(p)=insert(lc(now),l,mid,x);
    else rc(p)=insert(rc(now),mid+1,r,x);
    sum(p)=sum(lc(p))+sum(rc(p));
    return p;
}
//rq,lq,l,r,k
ll ask(ll p,ll q,ll l,ll r,ll k)
{
    if(l==r) return l;
    ll mid=l+r>>1;
    ll lcnt=sum(lc(p))-sum(lc(q));
    if(lcnt>=k) return ask(lc(p),lc(q),l,mid,k);
    else return ask(rc(p),rc(q),mid+1,r,k-lcnt);
}
int main()
{
    IOS;
     idx=0;
    ll n,q;
```

```
    cin>>n>>q;
    vector<ll> a(n+1),v;
    v.push_back(0);
    for(int i=1;i<=n;i++) cin>>a[i],v.push_back(a[i]);
    sort(v.begin()+1,v.end());
    v.erase(unique(v.begin()+1,v.end()),v.end());
    //root[0]=build(1,n);
    for(int i=1;i<=n;i++)
    {
        ll x=lower_bound(v.begin()+1,v.end(),a[i])-v.begin();
        root[i]=insert(root[i-1],1,n,x);
    }
    for(int i=1;i<=q;i++)
    {
        ll l,r,k;
        cin>>l>>r>>k;
        cout<<v[ask(root[r],root[l-1],1,n,k)]<<"\n";
    }
}
```

珂朵莉树

(模板题 CF 896C)

```
#include<bits/stdc++.h>
#define debug cout<<"YES__!\n"
#define IOS ios::sync_with_stdio(false)
#define eps 1e-6

using namespace std;
typedef long long ll;
const ll N = 10010,M=200010;
const int inf=0x3f3f3f3f;
const ll mod=1e9+7;

struct Block
{
```

```
  int l;          //  区间左端点（包括）
  int r;          //  区间右端点（不包括）
  mutable int v;  //  区间元素

  //  自定义比较函数，关键字为区间左端点。
  bool operator<(const Block &rhs) const
  {
    return l < rhs.l;
  }
};

set<Block> tree;
typedef set<Block>::iterator iter;


void init(int l, int r, int v)
{
  tree.insert({l, r, v});
}

//  分裂区块，返回以 x 为左端点的区块的迭代器
//  分裂区块，返回以 x 为左端点的区块的迭代器
iter split(int x)
{
  //  寻找左端点第一个大于等于 x 区块
  iter it = tree.lower_bound({x,0,0});
  //  如果存在区块正好是左端点，那么就不用分裂，直接返回迭代器即可
  if (it != tree.end() && it->l == x)
  {
    return it;
  }

  //  否则就要退回一个区块
  it--;
  Block o = *it;
  //  删除原来的区块，插入新的区块即可
  tree.erase(it);
```

```cpp
    tree.insert({o.l, x - 1, o.v});
    return tree.insert({x, o.r, o.v}).first;
}

// 将区间[l,r]内的元素全部赋值为v
void assign(int l, int r, int v)
{
    // 切分区间
    iter itr = split(r + 1);
    iter itl = split(l);

    // 删除区间内的所有小区块
    tree.erase(itl, itr);

    // 插入目标区块
    tree.insert({l, r, v});
}

// 将区间[l,r]操作
//模板
//void proc(ll l, ll r)
//{
//    // 切分区间
//    iter itr = split(r + 1);
//    iter itl = split(l);
//
//    for (iter i = itl; i != itr; i++)
//    {
//        // TODO 遍历区块
//    }
//
//    // 删除区间内的所有小区块
//    tree.erase(itl, itr);
//
//    // 插入目标区块
//    tree.insert(Block(l, r, v));
//}
```

```cpp
//例子
//void add(ll l, ll r, int v)
//{
//   // 切分区间
//   iter itr = split(r + 1);
//   iter itl = split(l);
//   // 挨个加
//   for (iter i = itl; i != itr; i++)
//   {
//     i->v += v;
//   }
//}

//区间块的第 k 大
//int kth(ll l, ll r, int k)
//{
//   // 切分区间
//   iter itr = split(r + 1);
//   iter itl = split(l);
//   // 挨个加
//   priority_queue<int> pq;
//   for (iter i = itl; i != itr; i++)
//   {
//     pq.push(i->v);
//   }
//
//   for (int i = 0; i < k - 1; i++)
//   {
//     pq.pop();
//   }
//   return pq.top();
//}


signed main()
{
    IOS;
```

```
    return 0;
}
```

启发式合并

序列启发式合并

思想：把小序列合并到大序列

模板题：acwing2154：求序列有多少个颜色段，可修改

```
#include<bits/stdc++.h>
#define debug cout<<"YES__!\n"
#define IOS ios::sync_with_stdio(false)
#define eps 1e-6


using namespace std;
typedef long long ll;
const ll N = 1000010,M=400010;
const ll inf=0x3f3f3f3f3f3f3f3f;
const ll mod=1e9+7;

int ver[N],head[N],Nxt[N],idx;
int Sz[N],a[N];
int ans;
int n,m;
void add(int x,int y)
{
    ver[++idx]=y;Nxt[idx]=head[x];head[x]=idx;
    Sz[x]++;
}
void merge(int &x,int &y)
{
    if(x==y) return;
    if(Sz[x]>Sz[y]) swap(x,y);
    for(int i=head[x];i;i=Nxt[i])
    {
        int j=ver[i];
```

```
        ans-=(a[j-1]==y)+(a[j+1]==y);
    }
    for(int i=head[x];i;i=Nxt[i])
    {
        int j=ver[i];
        a[j]=y;
        if(!Nxt[i])
        {
            Nxt[i]=head[y],
            head[y]=head[x];
            break;
        }
    }
    head[x]=0;
    Sz[y]+=Sz[x],Sz[x]=0;
}
int now[N];
signed main()
{
    scanf("%d %d",&n,&m);
    ans=1;
    for(int i=1;i<=n;i++)
    {
        scanf("%d",&a[i]);
        add(a[i],i);
        if(i>1&&a[i]!=a[i-1]) ans++;
    }
    for(int i=1;i<N;i++) now[i]=i;
    int op,x,y;
    while(m--)
    {
        scanf("%d",&op);
        if(op==2) printf("%d\n",ans);
        else
        {
            scanf("%d %d",&x,&y);
            merge(now[x],now[y]);
        }
    }
```

```
}
```

树上启发式合并（DSU on tree）

思想：只保留重树，清空轻树。

模板题：acwing3189：求子树颜色最多的颜色， 颜色数值和。

```
#include <bits/stdc++.h>
#define IOS ios::sync_with_stdio(false)

using namespace std;
typedef long long ll;
const ll N = 200010,M=400010;
const ll inf=0x3f3f3f3f3f3f3f3f;
const ll mod=1e9+7;

int son[N],sz[N];
int head[N],Nxt[N],ver[N],tot;
ll color[N],n,ans[N],cnt[N];
ll sum,mx;

void add(int x,int y)
{
    ver[++tot]=y;Nxt[tot]=head[x];head[x]=tot;
}
//找重儿子
void dfs_son(int x,int fa)
{
    sz[x]=1;
    for(int i=head[x];i;i=Nxt[i])
    {
        int y=ver[i];
        if(y==fa) continue;
        dfs_son(y,x);
        sz[x]+=sz[y];
        if(sz[y]>sz[son[x]]) son[x]=y;
```

```
    }
}

//更新数据
void update(int x,int fa,int val,int pson)
{
    int c=color[x];
    cnt[c]+=val;
    if(cnt[c]>mx)
    {
        sum=c;
        mx=cnt[c];
    }
    else if(cnt[c]==mx)
    {
        sum+=c;
    }
    for(int i=head[x];i;i=Nxt[i])
    {
        int y=ver[i];
        if(y==fa||y==pson) continue;
        update(y,x,val,pson);//pson 不变，因为只用保留最初的子树的重树
    }
}

void dfs(int x,int fa,int keep)
{
    //先遍历轻树
    for(int i=head[x];i;i=Nxt[i])
    {
        int y=ver[i];
        if(y==fa||y==son[x]) continue;
        dfs(y,x,0);
    }
    if(son[x]) dfs(son[x],x,1);//有重树即遍历
    update(x,fa,1,son[x]);
    ans[x]=sum;
    if(!keep) update(x,fa,-1,0),sum=mx=0;//如果该树是轻树，则清空
```

```
}

signed main()
{
    scanf("%d",&n);
    for(int i=1;i<=n;i++) scanf("%d",&color[i]);
    for(int i=1;i<n;i++)
    {
        int x,y;scanf("%d %d",&x,&y);
        add(x,y);add(y,x);
    }
    dfs_son(1,-1);
    dfs(1,-1,1);
    for(int i=1;i<=n;i++) printf("%lld ",ans[i]);
    puts("");

}
```

可持久化并查集

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

using namespace std;
typedef long long ll;
const int N=200010;

int n,idx;
struct Segment{
    int root[N],st[N];//根和初始值
    int lc[N*40],rc[N*40],val[N*40];
    int idx;
    int build(int l,int r)
    {
        int p=++idx;
        if(l==r)
        {
            val[p]=st[l];
            return p;
```

```
        }
        int mid=l+r>>1;
        lc[p]=build(l,mid);
        rc[p]=build(mid+1,r);
        return p;
    }
    int change(int now,int l,int r,int pos,int x)
    {
        int p=++idx;
        lc[p]=lc[now];rc[p]=rc[now];
        if(l==r)
        {
            val[p]=x;
            return p;
        }
        int mid=l+r>>1;
        if(pos<=mid) lc[p]=change(lc[now],l,mid,pos,x);
        else rc[p]=change(rc[now],mid+1,r,pos,x);
        return p;
    }
    int ask(int now,int l,int r,int pos)
    {
        if(l==r)
        {
            return val[now];
        }
        int mid=l+r>>1;
        if(pos<=mid) return ask(lc[now],l,mid,pos);
        else return ask(rc[now],mid+1,r,pos);
    }
}fa,rk;

int findx(int ti,int x)
{
    int y=fa.ask(fa.root[ti],1,n,x);
    if(y==x) return x;
    return findx(ti,y);
}
```

```
void merge(int ti,int x,int y)
{
    x=findx(ti,x);
    y=findx(ti,y);
    if(x==y) return;
    int rx=rk.ask(rk.root[ti],1,n,x);
    int ry=rk.ask(rk.root[ti],1,n,y);
    if(rx<=ry)
    {
        fa.root[ti]=fa.change(fa.root[ti],1,n,x,y);
        //以下二选一
        //1. rk 存的 size
        //rk.root[ti]=rk.change(rk.root[ti],1,n,y,ry+rx);

        //2. rk 存的 height
        //if(rx==ry) rk.root[ti]=rk.change(rk.root[ti],1,n,y,ry+1);
    }
    else
    {
        fa.root[ti]=fa.change(fa.root[ti],1,n,y,x);
        //1. rk 存的 size
        //rk.root[ti]=rk.change(rk.root[ti],1,n,x,ry+rx);
    }
}


signed main()
{
    int m;
    scanf("%d %d",&n,&m);
    for(int i=1;i<=n;i++) fa.st[i]=i;
    fa.root[0]=fa.build(1,n);
    for(int i=1;i<=n;i++) rk.st[i]=1;
    rk.root[0]=rk.build(1,n);
    int op,x,y;
    for(int i=1;i<=m;i++)
    {
        scanf("%d",&op);
```

```
        fa.root[i]=fa.root[i-1];
        rk.root[i]=rk.root[i-1];
        if(op==1)
        {
            scanf("%d %d",&x,&y);
            merge(i,x,y);
        }
        else if(op==2)
        {
            scanf("%d",&x);
            fa.root[i]=fa.root[x];
            rk.root[i]=rk.root[x];
        }
        else
        {
            scanf("%d %d",&x,&y);
            if(findx(i,x)==findx(i,y)) puts("1");
            else puts("0");
        }
    }
}
```

莫队

```
struct node
{
    int id, l, r;
}q[M];
int a[N], n, m, len, res;
int ans[M], cnt[maxn];
int get(int x)
{
    return (x - 1) / len;
}
//玄学优化 奇偶分开搞
bool cmp(node x, node y)
{
```

```cpp
    int l = get(x.l), r = get(y.l);
    if (l != r) return l < r;
    if (l & 1) return x.r < y.r;
    else return x.r > y.r;
}
void add(int x)
{
    if (!cnt[x]) res ++;
    cnt[x] ++;
}
void del(int x)
{
    cnt[x] --;
    if (!cnt[x]) res --;
}
void work()
{
    cin >> n;
    len = sqrt(n);
    for (int i = 1; i <= n; i ++ ) cin >> a[i];
    cin >> m;
    for (int i = 0; i < m; i ++ )
    {
        int l, r; cin >> l >> r;
        q[i] = {i, l, r};
    }
    sort(q, q + m, cmp);
    for (int k = 0, i = 0, j = 1; k < m; k ++ )
    {
        int id = q[k].id, l = q[k].l, r = q[k].r;
        while (i < r) add(a[ ++ i]);
        while (i > r) del(a[i -- ]);
        while (j < l) del(a[j ++ ]);
        while (j > l) add(a[ -- j]);
        ans[id] = res;
    }
    for (int i = 0; i < m; i ++ ) cout << ans[i] << "\n";
}
```

数学知识

复数的相关性质

设 a,b 为实数，i^2=-1$i^2=-1$，形如 a+bi 的数叫负数，其中 i 被称为虚数单位，复数域是目前已知最大的域

在复平面中，xx 代表实数，yy 轴（除原点外的点）代表虚数，从原点(0,0)到(a,b)的向量表示复数 a+bi

模长：从原点(0,0)到点(a,b)的距离，即 sqrt(a^2+b^2)

幅角：假设以逆时针为正方向，从 x 轴正半轴到已知向量的转角的有向角叫做幅角

加法：

因为在复平面中，复数可以被表示为向量，因此复数的加法与向量的加法相同，都满足平行四边形定则（就是上面那个）

乘法：

几何定义：复数相乘，模长相乘，幅角相加

代数定义：

(a+bi)*(c+di)=(ac-bd)+(bc+ad)i

FFT

```
const int N = 300010;
const double PI = acos(-1);

int n, m;
struct Complex
{
    double x, y;
    Complex operator+ (const Complex& t) const
```

```
    {
        return {x + t.x, y + t.y};
    }
    Complex operator- (const Complex& t) const
    {
        return {x - t.x, y - t.y};
    }
    Complex operator* (const Complex& t) const
    {
        return {x * t.x - y * t.y, x * t.y + y * t.x};
    }
}a[N], b[N];
int rev[N], bit, tot;

void fft(Complex a[], int inv)
{
    for (int i = 0; i < tot; i ++ )
        if (i < rev[i])
            swap(a[i], a[rev[i]]);
    for (int mid = 1; mid < tot; mid <<= 1)
    {
        auto w1 = Complex({cos(PI / mid), inv * sin(PI / mid)});
        for (int i = 0; i < tot; i += mid * 2)
        {
            auto wk = Complex({1, 0});
            for (int j = 0; j < mid; j ++, wk = wk * w1)
            {
                auto x = a[i + j], y = wk * a[i + j + mid];
                a[i + j] = x + y, a[i + j + mid] = x - y;
            }
        }
    }
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i <= n; i ++ ) scanf("%lf", &a[i].x);
    for (int i = 0; i <= m; i ++ ) scanf("%lf", &b[i].x);
```

```
    while ((1 << bit) < n + m + 1) bit ++;
    tot = 1 << bit;
    for (int i = 0; i < tot; i ++ )
        rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (bit - 1));
    fft(a, 1), fft(b, 1);
    for (int i = 0; i < tot; i ++ ) a[i] = a[i] * b[i];
    fft(a, -1);
    for (int i = 0; i <= n + m; i ++ )
        printf("%d ", (int)(a[i].x / tot + 0.5));

    return 0;
}
```

欧拉函数:

O（logn）（单一欧拉函数）：

```
int phi(int n)
{
    int ans=n;
    for(int i=2;i<=sqrt(n);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;
}
```

O（nlogn）：

```
int phi[N];
void euler(int n)
{
    for(int i=2;i<=n;i++) phi[i]=i;
    for(int i=2;i<=n;i++)
    {
        if(phi[i]==i)
            for(int j=i;j<=n;j+=i)
                phi[j]=phi[j]/i*(i-1);
    }
}
```

O（n)（线筛的思想）：

```
int v[N],prime[N],phi[N];
int m;
void euler(int n)
{
    memset(v,0,sizeof v);   //最小质因子
    m=0;   //质数数量
    for(int i=2;i<=n;i++)
    {
        if(v[i]==0)       //i 为质数
        {
            v[i]=i,prime[++m]=i;
            phi[i]=i-1;
        }
        //给 i 乘上一个质因子
        for(int j=1;j<=m;j++)
        {
            //i 有比 prime[j]更小的质因子，或者超出 n 的范围，停止循环
            if(prime[j]>v[i]||prime[j]>n/i) break;
            //prime[j]是合数 i*prime[j]的最小质因子
            v[i*prime[j]]=prime[j];
```

```
                phi[i*prime[j]]=phi[i]*(i%prime[j]?prime[j]-1:prime[j]);
            }
        }
}
```

逆元：

inv[x]=phi(mod)-1

欧拉函数 phi(x) 为 x 的 欧拉函数

矩阵：

一些递推式可以用矩阵 ksm 去处理

例：**斐波那契数列**

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix}$$

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} f(n-2) \\ f(n-3) \end{bmatrix} = \cdots = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} f(2) \\ f(1) \end{bmatrix}$$

```
//矩阵
struct stu {
    int s[110][110];
} ma;

int n,m,p;
//n*m   m*p
stu mul(stu a,stu b) {
    stu c;
    for(int i=1; i<=n; i++) {
        for(int j=1; j<=p; j++) {
            c.s[i][j]=0;
```

```
            for(int k=1; k<=m; k++) {
                c.s[i][j]=c.s[i][j]+a.s[i][k]*b.s[k][j]%mod;
                c.s[i][j]%=mod;
            }
        }
    }
    return c;
}

//n*n
stu ksm(stu a,int k) {
    stu res;
     memset(res.s,0,sizeof res.s);
    for(int i=1;i<=n;i++) res.s[i][i]=1;
    while(k) {
        if(k&1) res=mul(res,a);
        a=mul(a,a);
        k>>=1;
    }
    return res;
}
```

大指数欧拉降幂：

费马小定理:

当 $a,p\in\mathbb{Z}$ 且 p 为质数，且 $a\not\equiv 0 \pmod p$ 时有:
$a^{p-1}\equiv 1 \pmod p$。

所以 $a^b\equiv a^{b \bmod (p-1)} \pmod p$。

欧拉定理:

当 $a,m\in\mathbb{Z}$，且 $\gcd(a,m)=1$ 时有:
$a^{\varphi(m)}\equiv 1 \pmod m$。

这里 $\varphi(x)$ 是数论中的欧拉函数。

所以 $a^b\equiv a^{b \bmod \varphi(m)} \pmod m$。

扩展欧拉定理:

当 $a,m\in\mathbb{Z}$ 时有:
$a^b\equiv\begin{cases}a^b & ,b<\varphi(m)\\ a^{b \bmod \varphi(m)+\varphi(m)} & ,b\geq\varphi(m)\end{cases} \pmod m$。

```
ll phi(ll n)
{
    ll ans=n;
    for(ll i=2;i<=sqrt(n);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;
}
ll ksm(ll n,ll x,ll mod)
{
    ll res=1;
    while(x)
```

```
    {
        if(x&1) res=res*n%mod;
        n=n*n%mod;
        x>>=1;
    }
    return res%mod;
}

signed main()
{
    IOS;
    ll a,mod;
    string s;
    cin>>a>>mod>>s;
    ll elmod=phi(mod);
    ll sum=0;
    bool flag=false;
    for(ll i=0;i<s.size();i++)
    {
        ll x=s[i]-'0';
        sum=sum*10+x;
        if(sum>=elmod)
        {
            flag=true;
            sum%=elmod;
        }
    }
    if(flag) sum+=elmod;
    cout<<ksm(a,sum,mod)<<"\n";
}
```

组合数：

```
//组合数
ll cal(ll a,ll b)
{
```

```
    ll res1=1,res2=1;
    for(ll i=a,j=1;j<=b;j++,i--)
    {
        res1=res1*i%mod;
        res2=res2*j%mod;
    }
    res1=res1*ksm(res2,mod-2)%mod;
    return res1%mod;
}
```

这题需要求组合数，我总结了一下我知道的组合数取模的求法（P1313模板）：

1. 使用杨辉三角$C_n^0 = C_n^n = 1$，$C_n^i = C_{n-1}^i + C_{n-1}^{i-1}$ 代码见这里
2. **当p是质数时**可以得出a的逆元为a^{p-2}，$C_n^0 = 1$，$C_n^i = C_n^{i-1}\frac{k-i+1}{i}$ 代码见这里
3. **当p不是质数时**只能用上述方法，筛出质数并约分代码见这里

卡特兰数：

前几项：**1, 1, 2, 5, 14, 42, 132, 429, 1430**

应用

- 出栈次序（依次入栈，有多少个出栈队列）
- n 对括号正确匹配数目
- 给定节点组成二叉搜索树
- 在圆上选择 2n 个点，将这些点成对连接起来使得所得到的 n 条线段不相交的方法数
- 求一个凸多边形区域划分成三角形区域的方法数

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

$$
=\frac{C_{2n}^{n}}{n+1}=C_{2n}^{n}-C_{2n}^{n-1}
$$

递推写法：

f(n)=f(n−1)∗ (4∗ n−2)/(n+1)

线筛

```
int v[N],prime[N],m;
void iit(int n)
{
    m=0;
    v[1]=1;
    for(int i=2;i<=n;i++)
    {
        if(v[i]==0)
        {
            v[i]=i;
            prime[++m]=i;
        }
        for(int j=1;j<=m;j++)
        {
            if(prime[j]>v[i]||prime[j]>n/i) break;
            v[i*prime[j]]=prime[j];
        }
    }
//  for(int i=1;i<=100;i++) cout<<v[i]<<" ";
}
```

字符串板子

马拉车算法

```
string tmp;//转换后的字符串
int Len[N<<1];
//转换原始串
int INIT(string st)
{
    tmp.clear();
     int i,len=st.size();
     tmp.push_back('@');//字符串开头增加一个特殊字符，防止越界
     for(i=1;i<=2*len;i+=2)
     {
         tmp.push_back('#');
         tmp.push_back(st[i/2]);
     }
     tmp.push_back('#');
     tmp.push_back('$');//字符串结尾加一个字符，防止越界
     tmp.push_back(char(0));
     return 2*len+1;//返回转换字符串的长度
}

//Manacher 算法计算过程
int MANACHER(int len)
{
      int mx=0,ans=0,po=0;//mx 即为当前计算回文串最右边字符的最大值
      for(int i=1;i<=len;i++)
      {
          if(mx>i)
          Len[i]=min(mx-i,Len[2*po-i]);//在 Len[j]和 mx-i 中取个小
          else
          Len[i]=1;//如果 i>=mx，要从头开始匹配
          while(tmp[i-Len[i]]==tmp[i+Len[i]])
          Len[i]++;
```

```
        if(Len[i]+i>mx)//若新计算的回文串右端点位置大于 mx，要更新 po 和 mx 的值
        {
            mx=Len[i]+i;
            po=i; //回文串的中心
        }
        ans=max(ans,Len[i]);
    }
    //r=po+ls>>1; 回文边界，ls 为回文长度
    // l=r-ls+1;
    return ans-1;//返回 Len[i]中的最大值-1 即为原串的最长回文子串额长度
}
```

中线扩展的简化版

```
for (int i = 0; i < 2 * n - 1; ++i)
     int l = i / 2, r = l + i % 2; // 中心扩展法
```

字典树板子

```
namespace Trie{
    int trie[N][26],idx;
    //N 个节点 ,每个节点后可有 26 个孩子,trie 记录的是 idx;
    void iit()
    {
                idx++;     //取值的时候用的++idx，清空要多清空一个
        memset(trie,0,26*idx*sizeof(int));
        idx=0;
    }
    void insert(string s)
    {
        int len=s.size(),now=0;
        for(int i=0;i<len;i++)
        {
            int k=s[i]-'a';
            if(trie[now][k]==0)
            {
```

```
                trie[now][k]=++idx;
            }
            now=trie[now][k];
        }
        //cntword[now]++ //单词统计
    }
    bool search(string s)
    {
        int len=s.size(),now=0;
        for(int i=0;i<len;i++)
        {
            now=trie[now][s[i]-'a'];
            if(now==0) return false;
        }
        return true;
    }
}
//Trie::insert(a[i]);
```

KMP 算法

```
int Nxt[N];
ll ans;
void Get_Nxt(string s)
{
    Nxt[0]=-1;
    for(int i=0,j=-1;i<s.size();)//无 i++!!!
    {
        if(j==-1||s[i]==s[j])Nxt[++i]=++j;
        else j=Nxt[j];
    }
}
//字符串过多时，慎重传参，时间较高
void KMP(string s1,string s2) //s1 是长串 s2 是短串
{
    Get_Nxt(s2);//s2 为模板串
```

```
    ans=0;
    for(int i=0,j=0;i<s1.size();)//无 i++!!!
    {
        if(j==-1||s1[i]==s2[j])i++,j++;
        else
        {
            j=Nxt[j];
        }
        if(j==s2.size())//完全匹配
        {
//          j=0;//不重叠匹配计数（二选一）
//          j=Nxt[j];//重叠匹配计数（二选一）
            ans++;
        }
    }
}
```

KMP 循环节

完全循环

```
//完全循环  abcabc
void work(string &s)//求前缀中最长的循环节
{
    Get_Nxt(s);
    for(int i=1;i<s.size();i++)
    {
        int len=i+1;
        int mid=len%(len-Nxt[len]);
        if(mid==0&&Nxt[len]!=0)
        {
            int res=len/(len-Nxt[len]);
            cout<<len<<" "<<res<<"\n";
        }
```

```
    }
}
```

不完全循环

```
//不完全循环  abca->abcabc 需要补充 bc 才能构成完全循环
void work(string &s)      //需要补充多少个才能变成完全循环
{
    Get_Nxt(s);
    int n=s.size();
    int len=n;
    int mid=n-Nxt[len];
    if(mid!=n&&n%mid==0)
    {
        cout<<0<<"\n";
    }
    else
    {
        cout<<mid-Nxt[len]<<"\n";
    }
}
```

KMP 自动机

```
int Nxt[N];
int to[N][26];
ll ans;
void Get_Nxt(string s)
{
    Nxt[0]=-1;
    to[0][s[0]-'A']=1;
    Nxt[1]=0;
    for(int i=1;i<s.size();i++)
```

```
    {
        for(int j=0;j<26;j++)
        {
            if(j==s[i]-'A')
            {
                    //类似 AC 自动机，对照 AC 自动机理解
                to[i][j]=i+1;
                Nxt[i+1]=to[Nxt[i]][j];
            }
            else
            {
                to[i][j]=to[Nxt[i]][j];
            }
        }
    }
}
//KMP 操作相同
```

AC 自动机

```
int trie[N][26];
int cntword[N];
int fail[N];
int cnt=0;

void insertword(string s)      //字典树插入
{
    int len=s.size(),now=0;
    for(int i=0;i<len;i++)
    {
        int k=s[i]-'a';
        if(trie[now][k]==0)
        {
            trie[now][k]=++cnt;
        }
        now=trie[now][k];
    }
```

```
    cntword[now]++;
}

void Get_Fail()      //构建失配指针，类似 KMP
{                   //Nxt[i]=j 意义：word[j]为 word[i]的最长后缀
    queue<int> q;
    for(int i=0;i<26;i++)
    {
        if(trie[0][i])
        {
            fail[trie[0][i]]=0;
            q.push(trie[0][i]);
        }
    }

    while(q.size())
    {
        int now=q.front();
        q.pop();

        for(int i=0;i<26;i++)
        {
            if(trie[now][i])
            {
                //有该子节点
                //该子节点的失配节点 设为 自己 失配节点 的 'a'+i 子节点
                fail[trie[now][i]]=trie[fail[now]][i];
                q.push(trie[now][i]);
            }
            else
            {
                //没有该子节点
                //将该子节点 设为 自己 失配的节点 的 'a'+i 的子节点
                trie[now][i]=trie[fail[now]][i];
            }
        }
    }
}
```

```
int query(string s) //模板串
{
    int now=0,ans=0;
    for(int i=0;i<s.size();i++)
    {
        now=trie[now][s[i]-'a'];
        for(int j=now;j&&cntword[j]!=-1;j=fail[j])
        {
            ans+=cntword[j];
            cntword[j]=-1;
        }
    }
    return ans;
}
```

Hash(单模)

```
const ull P=131;
ull p[N],h[N];
ull get_hash(ll l,ll r)
{
    return h[r]-h[l-1]*p[r-l+1];
}
void iit(string s)
{
    p[0]=1;
    for(ll i=0;i<s.size();i++)
    {
        p[j+1]=p[i]*P;
        h[j+1]=h[j]*P+s[i]-'a' + 1;
    }
}
```

Hash(双模)

```
map<array<ll, 2>,ll> mp;
struct Hash{
    int size, base[2] = {  20023,20011 };
    vector<array<ll, 2>> hash, pow_base;
    Hash(){}
    Hash(const string& s){
        size = s.size();
        hash.resize(size);
        pow_base.resize(size);
        pow_base[0][0] = pow_base[0][1] = 1;
        hash[0][0] = hash[0][1] = s[0];
        for(int i = 1; i < size; i++){
            hash[i][0] = (hash[i - 1][0] * base[0] + s[i]) % mod[0];
            hash[i][1] = (hash[i - 1][1] * base[1] + s[i]) % mod[1];
            pow_base[i][0] = pow_base[i - 1][0] * base[0] % mod[0];
            pow_base[i][1] = pow_base[i - 1][1] * base[1] % mod[1];
        }
    }
    array<ll, 2> operator[](const array<int, 2>& range)const{
        if(range[0] == 0)return hash[range[1]];
        return { (hash[range[1]][0] - hash[range[0] - 1][0] *
pow_base[range[1] - range[0] + 1][0] % mod[0] + mod[0]) % mod[0],
(hash[range[1]][1] - hash[range[0] - 1][1] * pow_base[range[1] -
range[0] + 1][1] % mod[1] + mod[1]) % mod[1] };
    }
};


        string s;cin>>s;
        Hash h(s);
        array<ll, 2> u=h[{l-1, r-1}];//[l,r]的 hash 值
```

图论

加边操作

```
using namespace std;
typedef long long ll;
const ll N = 200010,M=400010;
const ll inf=0x3f3f3f3f3f3f3f3f;
const ll mod=1e9+7;
int ver[M],Next[M],edge[M],head[N];
int n,m,tot;

//1e6 以上的数据 慎用 vector
void add(int x,int y,int z)
{
    ver[++tot]=y;edge[tot]=z;Next[tot]=head[x];head[x]=tot;
}
```

LCA(vector)

```
int d[N];
int f[N][20];
vector<int> son[N];

void bfs()
{
    queue<ll> q;
    q.push(1);d[1]=1;
    while(q.size())
    {
        int x=q.front();q.pop();
        for(auto y:son[x])
        {
            if(d[y]) continue;
```

```
                d[y]=d[x]+1;
                f[y][0]=x;
                for(int i=1;i<20;i++)
                {
                    f[y][i]=f[f[y][i-1]][i-1];
                }
                q.push(y);
            }
        }
}
int lca(int x,int y)
{
    if(d[x]>d[y]) swap(x,y);
    for(int i=19;i>=0;i--)
    {
        if(d[f[y][i]]>=d[x]) y=f[y][i];
    }
    if(x==y) return x;
    for(int i=19;i>=0;i--)
    {
        if(f[x][i]!=f[y][i]) x=f[x][i],y=f[y][i];
    }
    return f[x][0];
}
```

LCA(链式)

```
void bfs()
{
    q.push(1);d[1]=1;
    while(q.size())
    {
        int x=q.front();q.pop();
        for(int i=head[x];i;i=Next[i])
        {
            int y=ver[i];
            if(d[y]) continue;
```

```
            d[y]=d[x]+1;
            dist[y]=dist[x]+edge[i];
            f[y][0]=x;
            for(int j=1;j<20;j++)
            {
                f[y][j]=f[f[y][j-1]][j-1];
            }
            q.push(y);
        }
    }
}

int lca(int x,int y)
{
    if(d[x]>d[y]) swap(x,y);
    for(int i=t;i>=0;i--)
    {
        if(d[f[y][i]]>=d[x]) y=f[y][i];
    }
    if(x==y) return x;
    for(int i=19;i>=0;i--)
    {
        if(f[x][i]!=f[y][i]) x=f[x][i],y=f[y][i];
    }
    return f[x][0];
}
```

Tarjan

割点：删去该点，图分裂成两个子图

割边（桥）：删去该边，图分裂成两个子图

时间戳（dfn）：在图被 dfs 的过程中，点第一次被访问的顺序

搜索树：在无向连通图中任选一个节点出发进行深度优先遍历，每个点只访问一次，所有发生递归的边（x,y)构成一棵树，我们把他称之为“无向连通图的搜索树”

追溯值（low）：low[x]为以 x 为 根 节点的 子树 和任意一个到达该子树路径为 1 的点的 dfn 的最小值。

割边判定：

判定法则：当且仅当搜索树上存在 x 的一个子节点 y，满足 dfn[x]<low[y]

```
int dfn[N],low[N],num;
bool bridge[M];
void tarjan(int x,int in_edge)
{
    dfn[x]=low[x]=++num;
    for(int i=head[x];i;i=Next[i])
    {
        int y=ver[i];
        if(!dfn[y])
        {
            tarjan(y,i);
            low[x]=min(low[x],low[y]);
            if(low[y]>dfn[x])
                bridge[i]=bridge[i^1]=true;
        }
        else if(i!=(in_edge^1))
            low[x]=min(low[x],dfn[y]);
    }
}

signed main()
{
    IOS;
    tot=1;
    cin>>n>>m;
    for(int i=1;i<=m;i++)
    {
        int x,y;
        cin>>x>>y;
        add(x,y,1);add(y,x,1);
    }
    for(int i=1;i<=n;i++)
    {
        if(!dfn[i]) tarjan(i,0);
```

```
    }
    for(int i=2;i<tot;i+=2)
    {
        if(bridge[i])
        {
            cout<<ver[i^1]<<" "<<ver[i]<<"\n";
        }
    }
}
```

割点判定：

判定法则：

1.若 x 不是搜索树的根节点，则 x 是割点当且仅当搜索树上存在 x 的一个子节点 y，满足

dfn[x]<=low[y]

2.若 x 是搜索树的根节点，则 x 是割点当且仅当 x 搜索树上存在至少两个子节点 y1，y2 满足

dfn[x]<=low[y1] && dfn[x]<=low[y2]

```
int dfn[N],low[N],num,root;
bool cut[N];
void tarjan(int x)  //割点不用考虑重边
{
    dfn[x]=low[x]=++num;
    int flag=0;
    for(int i=head[x];i;i=Next[i])
    {
        int y=ver[i];
        if(!dfn[y])
        {
            tarjan(y);
            low[x]=min(low[x],low[y]);
            if(low[y]>=dfn[x])
            {
                flag++;
                if(x!=root||flag>1)
                    cut[x]=true;
            }
```

```
        }
        else low[x]=min(low[x],dfn[y]);
    }
}

signed main()
{
    IOS;
    tot=1;
    cin>>n>>m;
    for(int i=1;i<=m;i++)
    {
        int x,y;
        cin>>x>>y;
        if(x==y) continue;
        add(x,y,1);add(y,x,1);
    }
    for(int i=1;i<=n;i++)
    {
        if(!dfn[i]) root=i,tarjan(i);
    }
    for(int i=1;i<=n;i++)
    {
        if(cut[i])
        {
            cout<<i<<" ";
        }
    }
    cout<<"\n";
}
```

无向图的双连通分量

点双连通图：一张无向连通图不存在 割点

边双连通图：一张无向连通图不存在 桥

点双连通子图(v-DCC)：无向图的 极大 点双连通子图

边双连通子图(e-DCC)：无向图的 极大 边双连通子图

双连通分量(DCC)：点双连通子图 以及边双连通子图 二者统称为 双连通分量。

简单环：指的是不自交的环，也就是通常所画的环。

定理：

一张无向连通图是 “点双连通图” ，当且仅当满足下列两个条件之一：

1.图的顶点数量不超过 2

2.图中的任意两点都同时包含在至少一个简单环中。

一张无向连通图是 ” 边双连通图 “，当且仅当任意一条边都包含在至少一个简单的环中。

e-DCC 的求法

删去所有桥，剩余的每个连通块即为一个 e-DCC

```
int c[N],dcc;
//c[i] 表示 i 所属的边双连通分量的编号
void dfs(int x)
{
    c[x]=dcc;
    for(int i=head[x];i;i=Next[i])
    {
        int y=ver[i];
        if(c[y]||bridge[i]) continue;
        dfs(y);
    }
}

signed main()
{
/*
    Tarjan 割边判定
*/
    for(int i=1;i<=n;i++)
    {
        if(!c[i])
        {
            ++dcc;
```

```
            dfs(i);
        }
    }
    printf("There are %d e-DCCs.\n",dcc);
    for(int i=1;i<=n;i++)
    {
        printf("%d belongs to DCC %d.\n",i,c[i]);
    }
}
```

e-DCC 的缩点

```
int hc[N],vc[M],nc[M],tc;
void add_c(int x,int y)
{
    vc[++tc]=y,nc[tc]=hc[x],hc[x]=tc;
}
signed main()
{
/*
    Tarjan 割边判定
*/

/*

    e-DCC 的求法
*/
    tc=1;
    for(int i=2;i<=tot;i++)
    {
        int x=ver[i^1],y=ver[i];
        if(c[x]==c[y]) continue;
        add_c(c[x],c[y]);
    }
    printf("缩点之后的森林，点数%d，边数%d(可能有重边)\n",dcc,tc/2);
    for(int i=2;i<tc;i+=2)
    {
```

```
        printf("%d %d\n",vc[i^1],vc[i]);
    }
}
```

v-DCC 的求法

整个图被割点划分成若干个连通块，每个连通块就是一个 v-DCC

割点可能属于若干个 v-DCC

```
int dfn[N],low[N],num,root;
bool cut[N];

int stack[N],top,cnt;
vector<int>dcc[N];

void tarjan(int x)   //割点不用考虑重边
{
    dfn[x]=low[x]=++num;
    stack[++top]=x;
    if(x==root&&head[x]==0)
    {
        dcc[++cnt].push_back(x);
        return;
    }
    int flag=0;
    for(int i=head[x];i;i=Next[i])
    {
        int y=ver[i];
        if(!dfn[y])
        {
            tarjan(y);
            low[x]=min(low[x],low[y]);
            if(low[y]>=dfn[x])
            {
                flag++;
                if(x!=root||flag>1)
                    cut[x]=true;
```

```
                cnt++;
                int z;
                do{
                    z=stack[top--];
                    dcc[cnt].push_back(z);
                }while(z!=y);
                dcc[cnt].push_back(x);
            }
        }
        else low[x]=min(low[x],dfn[y]);
    }
}

signed main()
{
    IOS;
    tot=1;
    cin>>n>>m;
    for(int i=1;i<=m;i++)
    {
        int x,y;
        cin>>x>>y;
        if(x==y) continue;
        add(x,y,1);add(y,x,1);
    }
    for(int i=1;i<=n;i++)
    {
        if(!dfn[i]) root=i,tarjan(i);
    }
    printf("割点有:");
    for(int i=1;i<=n;i++)
    {
        if(cut[i])
        {
            cout<<i<<" ";
        }
    }
    cout<<"\n";
    for(int i=1;i<=cnt;i++)
```

```
    {
        printf("v-DCC #%d:",i)
        for(int j=0;j<dcc[i].size();j++)
        {
            printf(" %d",dcc[i][j]);
        }
        printf("\n");
    }
}
```

v-DCC 的缩点

割点单独列出来，dcc 专门缩成一个点。

```
int new_id[N],c[N];
int hc[N],vc[M],nc[M],tc;
void add_c(int x,int y)
{
    vc[++tc]=y,nc[tc]=hc[x],hc[x]=tc;
}

signed main()
{
    IOS;
/*
    v-DCC 的求法
*/
    // 给每个割点一个新的编号 编号从 cnt+1 开始
    num=cnt;
    for(int i=1;i<=n;i++)
        if(cut[i]) new_id[i]=++num;
    //建新图，从每个 v-DCC 到它包含的所有割点连边
    tc=1;
    for(int i=1;i<=cnt;i++)
    {
        for(int j=0;j<dcc[i].size();j++)
        {
```

```
            int x=dcc[i][j];
            if(cut[x])
            {
                add_c(i,new_id[x]);
                add_c(new_id[x],i);
            }
            else c[x]=i;//除割点外，其他店仅属于一个 dcc
        }
    }
    printf("缩点之后的森林，点数%d，边数%d\n",num,tc/2);
    printf("编号 1~%d 的为原图的 v-DCC，编号 大于%d 的为原图割点\n",cnt,cnt);
    for(int i=2;i<tc;i+=2)
    {
        printf("%d %d\n",vc[i^1],vc[i]);
    }
}
```

有向图连通性

流图：存在 r，满足从 r 出发能够到达 V 中所有的点，则称 G 是一个 “流图”

流图中存在四类边：

(x,y)=x ---->y

1. 树枝边：指搜索树中的边，即 x 是 y 的父节点
2. 前向边：指搜索树中 x 是 y 的祖先节点
3. 后向边：指搜索树中 y 是 x 的祖先节点
4. 横叉边：指除了以上三种情况之外的边，它一定满足 dfn[y]<dfn[x]

维护某个栈：当访问到节点 x 时，栈中保留以下两类节点：

1.搜索树上 x 的祖先节点，记为集合 anc(x)

2.已经访问过，并且存在一条路径到达 anc(x)的节点

追溯值：x 的追溯值 low[x]定义为满足以下条件的节点的最小时间戳(dfn):

1.该点在栈中

2.存在一条从 subtree(x)出发的有向边，以该点为终点

追溯值的求法：

1.当节点 x 第一次被访问时，把 x 入栈，初始化 low[x]=dfn[x]

2.扫描从 x 出发的每条边(x,y)：

a. 若 y 没被访问过，则说明(x,y)是树枝边，递归访问 y，从 y **回溯**之后，令 low[x]=min(low[x],low[y])。

b. 若 y 被访问过**并且** y 在栈中，则令 low[x]=min(low[x],dfn[y])

3.从 x 回溯之前，判断是否有 low[x]=dfn[x]。若成立，则不断从栈中弹出节点，直至 x 出栈

强连通分量判定法则

强连通图：给定一张有向图。若对于图中任意两个节点 x,y，既存在从 x 到 y 的路径，也存在 y 到 x 的路径，则称该有向图是 “强连通图”

强连通分量：有向图的极大强连通子图被称为 “强连通分量” ，简记为 SCC

```
#include <bits/stdc++.h>
#define IOS ios::sync_with_stdio(false)
#define debug cout<<"YES_____!"

using namespace std;
typedef long long ll;
const ll inf=0x3f3f3f3f;
const ll mod=1000000007;
const ll N=1000010,M=1000010;

int ver[M],Next[M],head[N],dfn[N],low[N];
int sta[N],ins[N],c[N];
vector<int> scc[N];
int n,m,tot,num,top,cnt;

void add(int x,int y)
{
    ver[++tot]=y,Next[tot]=head[x],head[x]=tot;
}
```

```
void tarjan(int x)
{
    dfn[x]=low[x]=++num;
    sta[++top]=x,ins[x]=1;
    for(int i=head[x];i;i=Next[i])
    {
        int y=ver[i];
        if(!dfn[y])
        {
            tarjan(y);
            low[x]=min(low[x],low[y]);
        }
        else if(ins[y])
        {
            low[x]=min(low[x],dfn[y]);
        }
    }
    if(dfn[x]==low[x])
    {
        cnt++;int z;
        do{
            z=sta[top--],ins[z]=0;
            c[z]=cnt,scc[cnt].push_back(z);
        }while(x!=z);
    }
}

signed main()
{
    scanf("%d %d",&n,&m);
    for(int i=1;i<=m;i++)
    {
        int x,y;
        scanf("%d %d",&x,&y);
        add(x,y);
    }
    for(int i=1;i<=n;i++)
    {
        if(!dfn[i]) tarjan(i);
```

```
    }
}
```

缩点

```
int hc[N],vc[M],nc[M],tc;
void add_c(int x,int y)
{
    vc[++tc]=y,nc[tc]=hc[x],hc[x]=tc;
}
signed main()
{
/*
    强连通分量的判断
*/
    for(int x=1;x<=n;x++)
    {
        for(int i=head[x];i;i=Next[i])
        {
            int y=ver[i];
            if(c[x]==c[y]) continue;
            add_c(c[x],c[y]);
        }
     }
}
```

网络流

EK 算法 （nm2)

tips：tot=1

```
#include <bits/stdc++.h>
#define IOS ios::sync_with_stdio(false)
```

```cpp
using namespace std;
typedef long long ll;
const ll N = 200010,M=400010;
const ll inf=0x3f3f3f3f;
const ll mod=1e9+7;
int ver[M],Next[M],edge[M],head[N],f[M];
int n,m,tot;

//1e6 以上的数据 慎用 vector
void add(int x,int y,int z)
{
    //tot 从 2 开始
    ver[++tot]=y;edge[tot]=z;Next[tot]=head[x];head[x]=tot;
    ver[++tot]=x;edge[tot]=0;Next[tot]=head[y];head[y]=tot;
}



int v[N],incf[N],pre[N];
bool bfs(int s,int t)
{
    memset(v,0,sizeof v);
    queue<int> q;
    q.push(s);
    v[s]=1;
    incf[s]=inf;
    while(q.size())
    {
        auto x=q.front();q.pop();
        for(int i=head[x];i;i=Next[i])
        {
            if(edge[i]>f[i])
            {
                int y=ver[i];
                if(v[y]) continue;
                incf[y]=min(incf[x],edge[i]-f[i]);
                pre[y]=i;
                q.push(y);
```

```
                v[y]=1;
                if(y==t) return true;
            }
        }
    }
    return false;
}

int EK(int s,int t)
{
    int maxflow=0;
    while(bfs(s,t))
    {
        int x=t;
        while(x!=s)
        {
            int i=pre[x];
            f[i]+=incf[t];
            edge[i^1]-=incf[t];
            x=ver[i^1];
        }
        maxflow+=incf[t];
    }
    return maxflow;
}

signed main()
{
    IOS;
    tot=1;
    int S,T;
    cin>>n>>m>>S>>T;
    for(int i=1;i<=m;i++)
    {
        int u,v,c;
        cin>>u>>v>>c;
        add(u,v,c);
    }
    cout<<EK(S,T)<<"\n";
```

```
}
```

Dinic 算法 （n2m)

```
#include <bits/stdc++.h>
#define IOS ios::sync_with_stdio(false)

using namespace std;
typedef long long ll;
const ll N = 200010,M=400010;
const ll inf=0x3f3f3f3f;
const ll mod=1e9+7;
int ver[M],Next[M],edge[M],head[N];
int n,m,tot;

//1e6 以上的数据 慎用 vector
void add(int x,int y,int z)
{
    ver[++tot]=y;edge[tot]=z;Next[tot]=head[x];head[x]=tot;
    ver[++tot]=x;edge[tot]=0;Next[tot]=head[y];head[y]=tot;
}



int d[N],now[M];
int s,t;

bool bfs()
{
    memset(d,0,sizeof d);
    queue<int> q;
    q.push(s);
    d[s]=1;
    now[s]=head[s];
    while(q.size())
    {
```

```
        auto x=q.front();q.pop();
        for(int i=head[x];i;i=Next[i])
        {
            int y=ver[i];
            if(edge[i]&&!d[y])
            {
                q.push(y);
                now[y]=head[y];
                d[y]=d[x]+1;
                if(y==t) return true;
            }
        }
    }
    return false;
}

int dfs(int x,int flow)
{
    if(x==t) return flow;
    int rest=flow,k,i;
    for(i=now[x];i&&rest;i=Next[i])
    {
        int y=ver[i];
        if(edge[i]&&d[y]==d[x]+1)
        {
            k=dfs(y,min(rest,edge[i]));
            if(!k) d[y]=0;            //剪枝
            edge[i]-=k;
            edge[i^1]+=k;
            rest-=k;
        }
        now[x]=i;//当前弧优化（避免重复遍历从 x 出发不可扩展的边）
    }
    return flow-rest;
}

int dinic()
{
    int flow=0;
```

```
    int maxflow=0;
    while(bfs())
    {
        while(flow=dfs(s,inf)) maxflow+=flow;
    }
    return maxflow;
}

signed main()
{
    IOS;
    tot=1;
    cin>>n>>m>>s>>t;
    for(int i=1;i<=m;i++)
    {
        int u,v,c;
        cin>>u>>v>>c;
        add(u,v,c);
     }
    cout<<dinic()<<"\n";
}
```

上下界可行流：

做法皆为，构建新图，维护流量守恒，将所有问题转化为无源汇可行流做法。

无源汇可行流：

```
signed main()
{
    IOS;
    tot=1;
    cin>>n>>m;
    s=n+1,t=n+2;
    vector<int> a(n+1),low(m+1);
```

```
    for(int i=1;i<=m;i++)
    {
        int u,v,c,cc;
        cin>>u>>v>>c>>cc;
        add(u,v,cc-c);
        low[i]=c;
        a[u]-=c;
        a[v]+=c;
     }
     int sum=0;
     for(int i=1;i<=n;i++)
     {
        //维持新图流量守恒
        if(a[i]>0) add(s,i,a[i]),sum+=a[i];//统计源点 s 的出量
        else if(a[i]<0) add(i,t,-a[i]);
    }
    if(dinic()<sum)  //是否满流
    {
        cout<<"NO\n";
    }
    else
    {
        cout<<"YES\n";
        for(int i=1;i<=m;i++)
        {
            //edge[i<<1|1]表示该边的流量
            cout<<low[i]+edge[i<<1|1]<<"\n";
        }
    }
}
```

有源汇最大流：

```
signed main()
```

```
{
    IOS;
    tot=1;
    int S,T;
    cin>>n>>m>>S>>T;
    s=n+1,t=n+2;
    vector<int> a(n+1),low(m+1);
    for(int i=1;i<=m;i++)
    {
        int u,v,c,cc;
        cin>>u>>v>>c>>cc;
        add(u,v,cc-c);
        low[i]=c;
        a[u]-=c;
        a[v]+=c;
     }
     int sum=0;
     for(int i=1;i<=n;i++)
     {
        //维持新图流量守恒
        if(a[i]>0) add(s,i,a[i]),sum+=a[i];//统计源点 s 的出量
        else if(a[i]<0) add(i,t,-a[i]);
    }
    add(T,S,inf);//维护流量守恒
    if(dinic()<sum)  //是否满流
    {
        cout<<"No Solution\n";
    }
    else
    {
        int res=edge[tot];//求新图中可行流中 s->t 的流量
        edge[tot-1]=edge[tot]=0;//删边 t->s, inf 的边
        s=S,t=T;
        cout<<res+dinic()<<"\n";//榨干 s->t 的流量
    }
}
```

有源汇最小流：

```
signed main()
{
    IOS;
    tot=1;
    int S,T;
    cin>>n>>m>>S>>T;
    s=n+1,t=n+2;
    vector<int> a(n+1),low(m+1);
    for(int i=1;i<=m;i++)
    {
        int u,v,c,cc;
        cin>>u>>v>>c>>cc;
        add(u,v,cc-c);
        low[i]=c;
        a[u]-=c;
        a[v]+=c;
     }
     int sum=0;
     for(int i=1;i<=n;i++)
     {
        //维持新图流量守恒
        if(a[i]>0) add(s,i,a[i]),sum+=a[i];//统计源点 s 的出量
        else if(a[i]<0) add(i,t,-a[i]);
    }
    add(T,S,inf);//维护流量守恒
    if(dinic()<sum)  //是否满流
    {
        cout<<"No Solution\n";
    }
    else
    {
```

```
        int res=edge[tot];//求新图中可行流 s->t 的流量
        edge[tot-1]=edge[tot]=0;//删 t->s,inf 的边
        s=T,t=S;//最大化 t->s 的流量，即最小化 s->t 的流量
        cout<<res-dinic()<<"\n";//榨干 t->s 的流量
    }
}
```

多源汇

建立超级源汇点，超级源点连接源点，汇点连接超级汇点

关键边

搜每一条正边，如果为 0，增加权值，跑增广路，如果 true 即为关键边

最大流的判定用法

```
//1e6 以上的数据 慎用 vector
void add(int x,int y,int z)
{
    //此题为双向边
    ver[++tot]=y;edge[tot]=z;Next[tot]=head[x];head[x]=tot;
    ver[++tot]=x;edge[tot]=z;Next[tot]=head[y];head[y]=tot;
    /*
    ver[++tot]=y;edge[tot]=z;Next[tot]=head[x];head[x]=tot;
    ver[++tot]=x;edge[tot]=0;Next[tot]=head[y];head[y]=tot;
    这种写法找一个临界点
    add(a,b,c)
    add(b,n+i,c)
    add(n+i,a,c)
    */
}

int K,f[M];
bool check(int mid)
{
```

```
    for(int i=1;i<=m;i++)
    {
        if(edge[i<<1]>mid) edge[i<<1]=0;
        else edge[i<<1]=1;
        if(edge[i<<1|1]>mid) edge[i<<1|1]=0;
        else edge[i<<1|1]=1;
    }
    return dinic()>=K;
}

signed main()
{
    IOS;
    tot=1;
    cin>>n>>m>>K;
    s=1;t=n;
    for(int i=1;i<=m;i++)
    {
        int x,y,z;cin>>x>>y>>z;
        add(x,y,z);
    }
    int l=1,r=1e6;
    memcpy(f,edge,M);//复制
    while(l<r)
    {
        int mid=l+r>>1;
        if(check(mid)) r=mid;
        else l=mid+1;
        memcpy(edge,f,M);
    }
    cout<<l<<"\n";
}
```

最大流最小割

最大流最小割：任何网络中最大流的流量都等于最小割的容量。

割：指网络中节点的划分，它把网络中的所有节点都划分成 S 和 T 的集合，源点 s∈S，汇点

t∈T，记为 CUT(S,T)，就像一条切割线把图中的节点切割成 S 和 T 两部分。

割的净容量 f(S,T):指切割线切中的边中，从 S 到 T 的边的 **流量 减** 从 T 到 S 的边的 **流量**。

割的容量 c(S,T)：指切割线切中的边中，从 S 到 T 的边的 **容量** 之和，最小割指**容量**最小的割。

最小割：指 容量 最小的割。

tips:割的容量不计算 T 到 S 的边。

引理 1：若 f 是网络 G 的一个流，CUT(S,T)是 G 的任意一个割，则 f 的流值等于割的净流量 f(S,T)。

f (S,T)= | f |

推论 1：若 f 是网络 G 的一个流，CUT(S,T)是 G 的任意一个割，则 f 的流值不超过割的容量 c(S,T)

| f | <= c(S,T)

最大流最小割定理：若 f 是网络中的最大流，CUT(S,T)是 G 的最小割，则最大流 f 的值等于 最小割的容量 c(S,T)。

最大权闭合图：

建图 tips:源点连正向收益点，负向收益点连汇点，正向收益及负向收益按关系连边

费用流

最大费用最大流算法

a. 求最大费用路

b. 沿最大费用路增流

```
int ver[M],Next[M],edge[M],head[N];
int n,m,tot;

int cost[M];
//1e6 以上的数据 慎用 vector
void add(int x,int y,int z,int c)
{
    ver[++tot]=y;edge[tot]=z;cost[tot]=c;Next[tot]=head[x];head[x]=tot;
    ver[++tot]=x;edge[tot]=0;cost[tot]=-c;Next[tot]=head[y];head[y]=tot;
}
```

```
int v[N],incf[N],pre[N],d[N];
int s,t,maxflow,ans,k;

bool spfa()
{
	queue<int> q;
	memset(d,0xcf,sizeof d);  //-inf
	memset(v,0,sizeof v);
	q.push(s);d[s]=0;v[s]=1;
	incf[s]=1<<30;
	while(q.size())
	{
		int x=q.front();v[x]=0;q.pop();
		for(int i=head[x];i;i=Next[i])
		{
			if(!edge[i]) continue;
			int y=ver[i];
			if(d[y]<d[x]+cost[i])
			{
				d[y]=d[x]+cost[i];
				incf[y]=min(incf[x],edge[i]);
				pre[y]=i;
				if(!v[y])v[y]=1,q.push(y);
			}
		}
	}
	if(d[t]==0xcfcfcfcf) return false;
	return true;
}
void update()
{
	int x=t;
```

```
	while(x!=s)
	{
		int i=pre[x];
		edge[i]-=incf[t];
		edge[i^1]+=incf[t];
		x=ver[i^1];
	}
	maxflow+=incf[t];
	ans+=d[t]*incf[t];
}

int f(int i,int j,int k)
{
	return (i-1)*n+j+k*n*n;
}

signed main()
{
	IOS;
	tot=1;
	cin>>n>>k;
	s=1,t=2*n*n;
	for(int i=1;i<=n;i++)
	{
		for(int j=1;j<=n;j++)
		{
			int c;cin>>c;
			add(f(i,j,0),f(i,j,1),1,c);
			add(f(i,j,0),f(i,j,1),k-1,0);
			if(j<n) add(f(i,j,1),f(i,j+1,0),k,0);
			if(i<n) add(f(i,j,1),f(i+1,j,0),k,0);
		}
```

```
	}
	while(spfa()) update(); //计算最大费用最大流
	cout<<ans<<"\n";
}
```

匈牙利算法（n2m)

```
using namespace std;
typedef long long ll;
const ll N = 1010,M=5010;
const ll inf=0x3f3f3f3f3f3f3f3f;
const ll mod=1e9+7;
ll ver[M],Next[M],head[N];
ll n,m,tot;

//1e6 以上的数据 慎用 vector
void add(ll x,ll y)
{
	ver[++tot]=y;Next[tot]=head[x];head[x]=tot;
}

int match[N],v[N];

bool maxmatch(int x)
{
	for(int i=head[x];i;i=Next[i])
	{
		int y=ver[i];
		if(!v[y])
		{
```

```
			v[y]=1;
			if(!match[y]||maxmatch(match[y]))
			{
				match[y]=x;
				return true;
			}
		}
	}
	return false;
}


signed main()
{
	IOS;
	while(cin>>n>>m)
	{
		tot=0;
		memset(match,0,sizeof match);
		memset(head,0,sizeof head);
		for(int i=1;i<=n;i++)
		{
			int cnt;
			cin>>cnt;
			for(int j=1;j<=cnt;j++)
			{
				int x;cin>>x;
				add(i,x+n);
				add(x+n,i);
			}
		}
		int ans=0;
```

```
        for(int i=1;i<=n;i++)
        {
            memset(v,0,sizeof v);
            if(maxmatch(i))ans++;
        }
        cout<<ans<<"\n";
    }
}
```

计算几何

```
#include <bits/stdc++.h>
#define debug cout<<"YES___!"
#define IOS ios::sync_with_stdio(false)

constexpr double eps=1e-8;
constexpr double PI=3.14159265358979323841;
using namespace std;

//点类|线段类
template <typename T> struct Point
{
    T x,y;
    bool operator ==(const Point &a) const{ return (abs(x-a.x)<=eps &&
abs(y-a.y)<=eps);}
    bool operator <(const Point &a) const {if (abs(x-a.x)<=eps) return
y<a.y-eps; return x<a.x-eps;}
    Point operator +(const Point &a) const{return {x+a.x,y+a.y};}
    Point operator -(const Point &a) const {return {x-a.x,y-a.y};}
    Point operator -() const{return {-x,-y};}
    Point operator *(const T k)const {return {k*x,k*y};}
    Point operator / (const T k) const {return{x/k,y/k};}
    T operator*(const Point &a)const {return x*a.x+y*a.y;}    //点积
    T operator^(const Point &a)const {return x*a.y-y*a.x;}
    int toleft(const Point &a) const{const auto t=(*this)^a;return
(t>eps)-(t<-eps);} //toleft 检验
```

```
    T len2() const {return (*this)*(*this);}    //模长的平方
    T dis2(const Point &a) const{return (a-(*this)).len2();} //距离的平
方
    double len() const {return sqrt(len2());}       //模长
    double dis(const Point &a) const{return sqrt(dis2(a));}  //距离
    double ang(const Point &a) const {return acos(max(-
1.0,min(1.0,((*this)*a)/len()*a.len())));}  //两线段的夹角
    Point rot(const double rad) const {return {x*cos(rad)-
y*sin(rad),x*sin(rad)+y*cos(rad)};}      //逆时针选转 rad 度的向量
};

//直线类
template <typename T> struct Line
{
    Point<T> p,v;    //p+tv
    bool operator ==(const Line &a) const{return
v.toleft(a.v)==0&&v.toleft(p-a.p)==0;}
    int toleft (const Point <T> &a) const{return v.toleft(a-p);}
    Point<T> inter(const Line &a) const{return p+v*((a.v^(p-
a.p))/(v^a.v));} //两直线交点
    double dis(const Point<T> &a) const{return abs(v^(a-p))/v.len();}
        //点到线的距离
    Point<T> proj(const Point<T> &a)const{return p+v*((v*(a-
p))/(v*v));}      //点到线的垂足
};

//多边形类
template<typename T> struct Polygon
{
    vector< Point<T> >p;
    inline size_t nxt(const size_t i) const{return i==p.size()-
1?0:i+1;}
    inline size_t pre(const size_t i) const{return i==0?p.size()-1:i-
1;}

    pair<bool,int> winding(const Point<T> &a) const       //回转数 站在 a
点
```

```
    {                                        //True|False 表示是否在边上
        int cnt=0;                           //cnt 表示回转数 ,回转数为 0  点在外部
        for(size_t i=0;i<p.size();i++)
        {
            Point<T> u=p[i],v=p[nxt(i)];
            if(abs((a-u)^(a-v))<=eps&&(a-u)*(a-v)<=eps) return {true,0};//判断是否在边上
            if(abs(u.y-v.y)<=eps) continue;
            Line<T> uv={u,v-u};
            if(u.y<v.y-eps &&uv.toleft(a)<=0) continue;
            if(u.y>v.y+eps &&uv.toleft(a)>=0) continue;
            if(u.y<a.y-eps &&v.y>=a.y-eps) cnt++;       //逆时针
            if(u.y>=a.y-eps &&v.y<a.y-eps) cnt--;       //顺时针
        }
        return {false,cnt};
    }
    double circ()const      //周长
    {
        double sum=0;
        for(size_t i=0;i<p.size();i++)
        {
            sum+=p[i].dis(p[nxt(i)]);
        }
        return sum;
    }
    T area2() const      //面积
    {
        T sum=0;
        for(size_t i=0;i<p.size();i++)
        {
            sum+=p[i]^p[nxt(i)];
        }
        return abs(sum);
    }
};
```

```cpp
using point=Point<double>;
using line=Line<double>;
using polygon=Polygon<double>;

//极角序
//(1) 常熟略大
struct argcmp
{
    bool operator()(const point&a,const point&b)const
    {
        //下半区<原点<x 正半轴<上半区<x 负半轴
        //逆时针排序
        const auto quad=[](const point &a)
        {
            if(a.y<-eps) return 1;
            if(a.y>eps) return 4;
            if(a.x<-eps) return 5;
            if(a.x>eps) return 3;
            return 2;
        };
        const int qa=quad(a),qb=quad(b);
        if(qa!=qb) return qa<qb;
        const auto t=a^b;
        if(abs(t)<=eps) return a*a<b*b-eps;
        return t>eps;
    }
};
//(2)用 atan2(y,x)函数 (精度误差)
//用法如下:
//bool operator<(const Point &x)const
//{
//  return rad<x.rad;//rad=atan2(y,x);
// }

//判断直线 l 是否穿过 a,b 之间.
inline bool is_inter(line l,point a,point b)
{
    return l.toleft(a)*l.toleft(b)<=0;
```

```
}


int main()
{
    return 0;
}
```

计算几何（nowcoder）

```
#include <bits/stdc++.h>
using namespace std;

using point_t=long double;  //全局数据类型，可修改为 long long 等

constexpr point_t eps=1e-8;
constexpr long double PI=3.1415926535897932384l;

// 点与向量
template<typename T> struct point
{
    T x,y;

    bool operator==(const point &a) const {return (abs(x-a.x)<=eps &&
abs(y-a.y)<=eps);}
    bool operator<(const point &a) const {if (abs(x-a.x)<=eps) return
y<a.y-eps; return x<a.x-eps;}
    bool operator>(const point &a) const {return !(*this<a ||
*this==a);}
    point operator+(const point &a) const {return {x+a.x,y+a.y};}
    point operator-(const point &a) const {return {x-a.x,y-a.y};}
    point operator-() const {return {-x,-y};}
    point operator*(const T k) const {return {k*x,k*y};}
    point operator/(const T k) const {return {x/k,y/k};}
    T operator*(const point &a) const {return x*a.x+y*a.y;}  // 点积
    T operator^(const point &a) const {return x*a.y-y*a.x;}  // 叉积，
注意优先级
```

```
    int toleft(const point &a) const {const auto t=(*this)^a; return (t>eps)-(t<-eps);}  // to-left 测试
    T len2() const {return (*this)*(*this);}  // 向量长度的平方
    T dis2(const point &a) const {return (a-(*this)).len2();}  // 两点距离的平方

    // 涉及浮点数
    long double len() const {return sqrtl(len2());}  // 向量长度
    long double dis(const point &a) const {return sqrtl(dis2(a));}  // 两点距离
    long double ang(const point &a) const {return acosl(max(-1.0l,min(1.0l,((*this)*a)/(len()*a.len()))));}  // 向量夹角
    point rot(const long double rad) const {return {x*cos(rad)-y*sin(rad),x*sin(rad)+y*cos(rad)};}  // 逆时针旋转（给定角度）
    point rot(const long double cosr,const long double sinr) const {return {x*cosr-y*sinr,x*sinr+y*cosr};}  // 逆时针旋转（给定角度的正弦与余弦）
};

using Point=point<point_t>;

// 极角排序
struct argcmp
{
    bool operator()(const Point &a,const Point &b) const
    {
        const auto quad=[](const Point &a)
        {
            if (a.y<-eps) return 1;
            if (a.y>eps) return 4;
            if (a.x<-eps) return 5;
            if (a.x>eps) return 3;
            return 2;
        };
        const int qa=quad(a),qb=quad(b);
        if (qa!=qb) return qa<qb;
        const auto t=a^b;
```

```
        // if (abs(t)<=eps) return a*a<b*b-eps;  // 不同长度的向量需要分
开
        return t>eps;
    }
};

// 直线
template<typename T> struct line
{
    point<T> p,v;  // p 为直线上一点，v 为方向向量

    bool operator==(const line &a) const {return v.toleft(a.v)==0 &&
v.toleft(p-a.p)==0;}
    int toleft(const point<T> &a) const {return v.toleft(a-p);}  //
to-left 测试
    bool operator<(const line &a) const  // 半平面交算法定义的排序
    {
        if (abs(v^a.v)<=eps && v*a.v>=-eps) return toleft(a.p)==-1;
        return argcmp()(v,a.v);
    }

    // 涉及浮点数
    point<T> inter(const line &a) const {return p+v*((a.v^(p-
a.p))/(v^a.v));}  // 直线交点
    long double dis(const point<T> &a) const {return abs(v^(a-
p))/v.len();}  // 点到直线距离
    point<T> proj(const point<T> &a) const {return p+v*((v*(a-
p))/(v*v));}  // 点在直线上的投影
};

using Line=line<point_t>;

//线段
template<typename T> struct segment
{
    point<T> a,b;
```

```
    bool operator<(const segment &s) const {return
make_pair(a,b)<make_pair(s.a,s.b);}

    // 判定性函数建议在整数域使用

    // 判断点是否在线段上
    // -1 点在线段端点 | 0 点不在线段上 | 1 点严格在线段上
    int is_on(const point<T> &p) const
    {
        if (p==a || p==b) return -1;
        return (p-a).toleft(p-b)==0 && (p-a)*(p-b)<-eps;
    }

    // 判断线段直线是否相交
    // -1 直线经过线段端点 | 0 线段和直线不相交 | 1 线段和直线严格相交
    int is_inter(const line<T> &l) const
    {
        if (l.toleft(a)==0 || l.toleft(b)==0) return -1;
        return l.toleft(a)!=l.toleft(b);
    }

    // 判断两线段是否相交
    // -1 在某一线段端点处相交 | 0 两线段不相交 | 1 两线段严格相交
    int is_inter(const segment<T> &s) const
    {
        if (is_on(s.a) || is_on(s.b) || s.is_on(a) || s.is_on(b))
return -1;
        const line<T> l{a,b-a},ls{s.a,s.b-s.a};
        return l.toleft(s.a)*l.toleft(s.b)==-1 &&
ls.toleft(a)*ls.toleft(b)==-1;
    }

    // 点到线段距离
    long double dis(const point<T> &p) const
    {
        if ((p-a)*(b-a)<-eps || (p-b)*(a-b)<-eps) return
min(p.dis(a),p.dis(b));
        const line<T> l{a,b-a};
```

```
        return l.dis(p);
    }

    // 两线段间距离
    long double dis(const segment<T> &s) const
    {
        if (is_inter(s)) return 0;
        return min({dis(s.a),dis(s.b),s.dis(a),s.dis(b)});
    }
};

using Segment=segment<point_t>;

// 多边形
template<typename T> struct polygon
{
    vector<point<T>> p;  // 以逆时针顺序存储

    size_t nxt(const size_t i) const {return i==p.size()-1?0:i+1;}
    size_t pre(const size_t i) const {return i==0?p.size()-1:i-1;}

    // 回转数
    // 返回值第一项表示点是否在多边形边上
    // 对于狭义多边形，回转数为 0 表示点在多边形外，否则点在多边形内
    pair<bool,int> winding(const point<T> &a) const
    {
        int cnt=0;
        for (size_t i=0;i<p.size();i++)
        {
            const point<T> u=p[i],v=p[nxt(i)];
            if (abs((a-u)^(a-v))<=eps && (a-u)*(a-v)<=eps) return {true,0};
            if (abs(u.y-v.y)<=eps) continue;
            const Line uv={u,v-u};
            if (u.y<v.y-eps && uv.toleft(a)<=0) continue;
            if (u.y>v.y+eps && uv.toleft(a)>=0) continue;
            if (u.y<a.y-eps && v.y>=a.y-eps) cnt++;
            if (u.y>=a.y-eps && v.y<a.y-eps) cnt--;
```

```
        }
        return {false,cnt};
    }

    // 多边形面积的两倍
    // 可用于判断点的存储顺序是顺时针或逆时针
    T area() const
    {
        T sum=0;
        for (size_t i=0;i<p.size();i++) sum+=p[i]^p[nxt(i)];
        return sum;
    }

    // 多边形的周长
    long double circ() const
    {
        long double sum=0;
        for (size_t i=0;i<p.size();i++) sum+=p[i].dis(p[nxt(i)]);
        return sum;
    }
};

using Polygon=polygon<point_t>;

//凸多边形
template<typename T> struct convex: polygon<T>
{
    // 闵可夫斯基和
    convex operator+(const convex &c) const
    {
        const auto &p=this->p;
        vector&lt;Segment&gt;
e1(p.size()),e2(c.p.size()),edge(p.size()+c.p.size());
        vector<point<T>> res; res.reserve(p.size()+c.p.size());
        const auto cmp=[](const Segment &u,const Segment &v) {return
argcmp()(u.b-u.a,v.b-v.a);};
        for (size_t i=0;i<p.size();i++) e1[i]={p[i],p[this->nxt(i)]};
        for (size_t i=0;i<c.p.size();i++)
```

```
e2[i]={c.p[i],c.p[c.nxt(i)]};

rotate(e1.begin(),min_element(e1.begin(),e1.end(),cmp),e1.end());

rotate(e2.begin(),min_element(e2.begin(),e2.end(),cmp),e2.end());

merge(e1.begin(),e1.end(),e2.begin(),e2.end(),edge.begin(),cmp);
        const auto check=[](const vector<point<T>> &res,const point<T>
&u)
        {
            const auto back1=res.back(),back2=*prev(res.end(),2);
            return (back1-back2).toleft(u-back1)==0 && (back1-
back2)*(u-back1)>=-eps;
        };
        auto u=e1[0].a+e2[0].a;
        for (const auto &v:edge)
        {
            while (res.size()>1 && check(res,u)) res.pop_back();
            res.push_back(u);
            u=u+v.b-v.a;
        }
        if (res.size()>1 && check(res,res[0])) res.pop_back();
        return {res};
    }

    // 旋转卡壳
    // func 为更新答案的函数，可以根据题目调整位置
    template<typename F> void rotcaliper(const F &func) const
    {
        const auto &p=this->p;
        const auto area=[](const point<T> &u,const point<T> &v,const
point<T> &w){return (w-u)^(w-v);};
        for (size_t i=0,j=1;i<p.size();i++)
        {
            const auto nxti=this->nxt(i);
            func(p[i],p[nxti],p[j]);
            while
(area(p[this->nxt(j)],p[i],p[nxti])>=area(p[j],p[i],p[nxti]))
            {
```

```
                j=this->nxt(j);
                func(p[i],p[nxti],p[j]);
            }
        }
    }

    // 凸多边形的直径的平方
    T diameter2() const
    {
        const auto &p=this->p;
        if (p.size()==1) return 0;
        if (p.size()==2) return p[0].dis2(p[1]);
        T ans=0;
        auto func=[&](const point<T> &u,const point<T> &v,const point<T> &w){ans=max({ans,w.dis2(u),w.dis2(v)});};
        rotcaliper(func);
        return ans;
    }

    // 判断点是否在凸多边形内
    // 复杂度 O(logn)
    // -1 点在多边形边上 | 0 点在多边形外 | 1 点在多边形内
    int is_in(const point<T> &a) const
    {
        const auto &p=this->p;
        if (p.size()==1) return a==p[0]?-1:0;
        if (p.size()==2) return segment<T>{p[0],p[1]}.is_on(a)?-1:0;
        if (a==p[0]) return -1;
        if ((p[1]-p[0]).toleft(a-p[0])==-1 || (p.back()-p[0]).toleft(a-p[0])==1) return 0;
        const auto cmp=[&](const Point &u,const Point &v){return (u-p[0]).toleft(v-p[0])==1;};
        const size_t i=lower_bound(p.begin()+1,p.end(),a,cmp)-p.begin();
        if (i==1) return segment<T>{p[0],p[i]}.is_on(a)?-1:0;
        if (i==p.size()-1 && segment<T>{p[0],p[i]}.is_on(a)) return -1;
        if (segment<T>{p[i-1],p[i]}.is_on(a)) return -1;
```

```
        return (p[i]-p[i-1]).toleft(a-p[i-1])>0;
    }

    // 凸多边形关于某一方向的极点
    // 复杂度 O(logn)
    // 参考资料: https://codeforces.com/blog/entry/48868
    template<typename F> size_t extreme(const F &dir) const
    {
        const auto &p=this->p;
        const auto check=[&](const size_t i){return
dir(p[i]).toleft(p[this->nxt(i)]-p[i])>=0;};
        const auto dir0=dir(p[0]); const auto check0=check(0);
        if (!check0 && check(p.size()-1)) return 0;
        const auto cmp=[&](const Point &v)
        {
            const size_t vi=&v-p.data();
            if (vi==0) return 1;
            const auto checkv=check(vi);
            const auto t=dir0.toleft(v-p[0]);
            if (vi==1 && checkv==check0 && t==0) return 1;
            return checkv^(checkv==check0 && t<=0);
        };
        return partition_point(p.begin(),p.end(),cmp)-p.begin();
    }

    // 过凸多边形外一点求凸多边形的切线，返回切点下标
    // 复杂度 O(logn)
    // 必须保证点在多边形外
    pair<size_t,size_t> tangent(const point<T> &a) const
    {
        const size_t i=extreme([&](const point<T> &u){return u-a;});
        const size_t j=extreme([&](const point<T> &u){return a-u;});
        return {i,j};
    }

    // 求平行于给定直线的凸多边形的切线，返回切点下标
    // 复杂度 O(logn)
    pair<size_t,size_t> tangent(const line<T> &a) const
```

```
    {
        const size_t i=extreme([&](...){return a.v;});
        const size_t j=extreme([&](...){return -a.v;});
        return {i,j};
    }
};

using Convex=convex<point_t>;

// 圆
struct Circle
{
    Point c;
    long double r;

    bool operator==(const Circle &a) const {return c==a.c && abs(r-
a.r)<=eps;}
    long double circ() const {return 2*PI*r;}  // 周长
    long double area() const {return PI*r*r;}  // 面积

    // 点与圆的关系
    // -1 圆上 | 0 圆外 | 1 圆内
    int is_in(const Point &p) const {const long double d=p.dis(c);
return abs(d-r)<=eps?-1:d<r-eps;}

    // 直线与圆关系
    // 0 相离 | 1 相切 | 2 相交
    int relation(const Line &l) const
    {
        const long double d=l.dis(c);
        if (d>r+eps) return 0;
        if (abs(d-r)<=eps) return 1;
        return 2;
    }

    // 圆与圆关系
    // -1 相同 | 0 相离 | 1 外切 | 2 相交 | 3 内切 | 4 内含
    int relation(const Circle &a) const
```

```
    {
        if (*this==a) return -1;
        const long double d=c.dis(a.c);
        if (d>r+a.r+eps) return 0;
        if (abs(d-r-a.r)<=eps) return 1;
        if (abs(d-abs(r-a.r))<=eps) return 3;
        if (d<abs(r-a.r)-eps) return 4;
        return 2;
    }

    // 直线与圆的交点
    vector<Point> inter(const Line &l) const
    {
        const long double d=l.dis(c);
        const Point p=l.proj(c);
        const int t=relation(l);
        if (t==0) return vector<Point>();
        if (t==1) return vector<Point>{p};
        const long double k=sqrt(r*r-d*d);
        return vector<Point>{p-(l.v/l.v.len())*k,p+(l.v/l.v.len())*k};
    }

    // 圆与圆交点
    vector<Point> inter(const Circle &a) const
    {
        const long double d=c.dis(a.c);
        const int t=relation(a);
        if (t==-1 || t==0 || t==4) return vector<Point>();
        Point e=a.c-c; e=e/e.len()*r;
        if (t==1 || t==3)
        {
            if (r*r+d*d-a.r*a.r>=-eps) return vector<Point>{c+e};
            return vector<Point>{c-e};
        }
        const long double costh=(r*r+d*d-
a.r*a.r)/(2*r*d),sinth=sqrt(1-costh*costh);
        return vector<Point>{c+e.rot(costh,-
sinth),c+e.rot(costh,sinth)};
    }
```

```
    // 圆与圆交面积
    long double inter_area(const Circle &a) const
    {
        const long double d=c.dis(a.c);
        const int t=relation(a);
        if (t==-1) return area();
        if (t<2) return 0;
        if (t>2) return min(area(),a.area());
        const long double costh1=(r*r+d*d-a.r*a.r)/(2*r*d),costh2=(a.r*a.r+d*d-r*r)/(2*a.r*d);
        const long double sinth1=sqrt(1-costh1*costh1),sinth2=sqrt(1-costh2*costh2);
        const long double th1=acos(costh1),th2=acos(costh2);
        return r*r*(th1-costh1*sinth1)+a.r*a.r*(th2-costh2*sinth2);
    }

    // 过圆外一点圆的切线
    vector<Line> tangent(const Point &a) const
    {
        const int t=is_in(a);
        if (t==1) return vector<Line>();
        if (t==-1)
        {
            const Point v={-(a-c).y,(a-c).x};
            return vector<Line>{{a,v}};
        }
        Point e=a-c; e=e/e.len()*r;
        const long double costh=r/c.dis(a),sinth=sqrt(1-costh*costh);
        const Point t1=c+e.rot(costh,-sinth),t2=c+e.rot(costh,sinth);
        return vector<Line>{{a,t1-a},{a,t2-a}};
    }

    // 两圆的公切线
    vector<Line> tangent(const Circle &a) const
    {
        const int t=relation(a);
        vector<Line> lines;
        if (t==-1 || t==4) return lines;
```

```
        if (t==1 || t==3)
        {
            const Point p=inter(a)[0],v={-(a.c-c).y,(a.c-c).x};
            lines.push_back({p,v});
        }
        const long double d=c.dis(a.c);
        const Point e=(a.c-c)/(a.c-c).len();
        if (t<=2)
        {
            const long double costh=(r-a.r)/d,sinth=sqrt(1-costh*costh);
            const Point d1=e.rot(costh,-sinth),d2=e.rot(costh,sinth);
            const Point u1=c+d1*r,u2=c+d2*r,v1=a.c+d1*a.r,v2=a.c+d2*a.r;
            lines.push_back({u1,v1-u1}); lines.push_back({u2,v2-u2});
        }
        if (t==0)
        {
            const long double costh=(r+a.r)/d,sinth=sqrt(1-costh*costh);
            const Point d1=e.rot(costh,-sinth),d2=e.rot(costh,sinth);
            const Point u1=c+d1*r,u2=c+d2*r,v1=a.c-d1*a.r,v2=a.c-d2*a.r;
            lines.push_back({u1,v1-u1}); lines.push_back({u2,v2-u2});
        }
        return lines;
    }

    // 圆的反演
    tuple<int,Circle,Line> inverse(const Line &l) const
    {
        const Circle null_c={{0.0,0.0},0.0};
        const Line null_l={{0.0,0.0},{0.0,0.0}};
        if (l.toleft(c)==0) return {2,null_c,l};
        const Point v=l.toleft(c)==1?Point{l.v.y,-l.v.x}:Point{-l.v.y,l.v.x};
        const long double d=r*r/l.dis(c);
        const Point p=c+v/v.len()*d;
        return {1,{(c+p)/2,d/2},null_l};
```

```
    }

    tuple<int,Circle,Line> inverse(const Circle &a) const
    {
        const Circle null_c={{0.0,0.0},0.0};
        const Line null_l={{0.0,0.0},{0.0,0.0}};
        const Point v=a.c-c;
        if (a.is_in(c)==-1)
        {
            const long double d=r*r/(a.r+a.r);
            const Point p=c+v/v.len()*d;
            return {2,null_c,{p,{-v.y,v.x}}};
        }
        if (c==a.c) return {1,{c,r*r/a.r},null_l};
        const long double d1=r*r/(c.dis(a.c)-
a.r),d2=r*r/(c.dis(a.c)+a.r);
        const Point p=c+v/v.len()*d1,q=c+v/v.len()*d2;
        return {1,{(p+q)/2,p.dis(q)/2},null_l};
    }
};

// 圆与多边形面积交
long double area_inter(const Circle &circ,const Polygon &poly)
{
    const auto cal=[](const Circle &circ,const Point &a,const Point
&b)
    {
        if ((a-circ.c).toleft(b-circ.c)==0) return 0.0l;
        const auto ina=circ.is_in(a),inb=circ.is_in(b);
        const Line ab={a,b-a};
        if (ina && inb) return ((a-circ.c)^(b-circ.c))/2;
        if (ina && !inb)
        {
            const auto t=circ.inter(ab);
            const Point p=t.size()==1?t[0]:t[1];
            const long double ans=((a-circ.c)^(p-circ.c))/2;
            const long double th=(p-circ.c).ang(b-circ.c);
            const long double d=circ.r*circ.r*th/2;
            if ((a-circ.c).toleft(b-circ.c)==1) return ans+d;
```

```
            return ans-d;
        }
        if (!ina && inb)
        {
            const Point p=circ.inter(ab)[0];
            const long double ans=((p-circ.c)^(b-circ.c))/2;
            const long double th=(a-circ.c).ang(p-circ.c);
            const long double d=circ.r*circ.r*th/2;
            if ((a-circ.c).toleft(b-circ.c)==1) return ans+d;
            return ans-d;
        }
        const auto p=circ.inter(ab);
        if (p.size()==2 && Segment{a,b}.dis(circ.c)<=circ.r+eps)
        {
            const long double ans=((p[0]-circ.c)^(p[1]-circ.c))/2;
            const long double th1=(a-circ.c).ang(p[0]-circ.c),th2=(b-
circ.c).ang(p[1]-circ.c);
            const long double
d1=circ.r*circ.r*th1/2,d2=circ.r*circ.r*th2/2;
            if ((a-circ.c).toleft(b-circ.c)==1) return ans+d1+d2;
            return ans-d1-d2;
        }
        const long double th=(a-circ.c).ang(b-circ.c);
        if ((a-circ.c).toleft(b-circ.c)==1) return circ.r*circ.r*th/2;
        return -circ.r*circ.r*th/2;
    };

    long double ans=0;
    for (size_t i=0;i<poly.p.size();i++)
    {
        const Point a=poly.p[i],b=poly.p[poly.nxt(i)];
        ans+=cal(circ,a,b);
    }
    return ans;
}

// 点集的凸包
// Andrew 算法，复杂度 O(nlogn)
Convex convexhull(vector<Point> p)
```

```
{
    vector<Point> st;
    if (p.empty()) return Convex{st};
    sort(p.begin(),p.end());
    const auto check=[](const vector<Point> &st,const Point &u)
    {
        const auto back1=st.back(),back2=*prev(st.end(),2);
        return (back1-back2).toleft(u-back1)<=0;
    };
    for (const Point &u:p)
    {
        while (st.size()>1 && check(st,u)) st.pop_back();
        st.push_back(u);
    }
    size_t k=st.size();
    p.pop_back(); reverse(p.begin(),p.end());
    for (const Point &u:p)
    {
        while (st.size()>k && check(st,u)) st.pop_back();
        st.push_back(u);
    }
    st.pop_back();
    return Convex{st};
}

// 半平面交
// 排序增量法，复杂度 O(nlogn)
// 输入与返回值都是用直线表示的半平面集合
vector<Line> halfinter(vector<Line> l, const point_t lim=1e9)
{
    const auto check=[](const Line &a,const Line &b,const Line
&c){return a.toleft(b.inter(c))<0;};
    // 无精度误差的方法，但注意取值范围会扩大到三次方
    /*const auto check=[](const Line &a,const Line &b,const Line &c)
    {
        const Point p=a.v*(b.v^c.v),q=b.p*(b.v^c.v)+b.v*(c.v^(b.p-
c.p))-a.p*(b.v^c.v);
        return p.toleft(q)<0;
```

```
    };*/
    l.push_back({{-lim,0},{0,-1}}); l.push_back({{0,-lim},{1,0}});
    l.push_back({{lim,0},{0,1}}); l.push_back({{0,lim},{-1,0}});
    sort(l.begin(),l.end());
    deque<Line> q;
    for (size_t i=0;i<l.size();i++)
    {
        if (i>0 && l[i-1].v.toleft(l[i].v)==0 && l[i-1].v*l[i].v>eps)
continue;
        while (q.size()>1 && check(l[i],q.back(),q[q.size()-2]))
q.pop_back();
        while (q.size()>1 && check(l[i],q[0],q[1])) q.pop_front();
        if (!q.empty() && q.back().v.toleft(l[i].v)<=0) return
vector<Line>();
        q.push_back(l[i]);
    }
    while (q.size()>1 && check(q[0],q.back(),q[q.size()-2]))
q.pop_back();
    while (q.size()>1 && check(q.back(),q[0],q[1])) q.pop_front();
    return vector<Line>(q.begin(),q.end());
}

// 点集形成的最小最大三角形
// 极角序扫描线，复杂度 O(n^2logn)
// 最大三角形问题可以使用凸包与旋转卡壳做到 O(n^2)
pair<point_t,point_t> minmax_triangle(const vector<Point> &vec)
{
    if (vec.size()<=2) return {0,0};
    vector<pair<int,int>> evt;
    evt.reserve(vec.size()*vec.size());
    point_t maxans=0,minans=numeric_limits<point_t>::max();
    for (size_t i=0;i<vec.size();i++)
    {
        for (size_t j=0;j<vec.size();j++)
        {
            if (i==j) continue;
            if (vec[i]==vec[j]) minans=0;
            else evt.push_back({i,j});
        }
```

```
    }
    sort(evt.begin(),evt.end(),[&](const pair<int,int> &u,const pair<int,int> &v)
    {
        const Point du=vec[u.second]-vec[u.first],dv=vec[v.second]-vec[v.first];
        return argcmp()({du.y,-du.x},{dv.y,-dv.x});
    });
    vector<size_t> vx(vec.size()),pos(vec.size());
    for (size_t i=0;i<vec.size();i++) vx[i]=i;
    sort(vx.begin(),vx.end(),[&](int x,int y){return vec[x]<vec[y];});
    for (size_t i=0;i<vx.size();i++) pos[vx[i]]=i;
    for (auto [u,v]:evt)
    {
        const size_t i=pos[u],j=pos[v];
        const size_t l=min(i,j),r=max(i,j);
        const Point vecu=vec[u],vecv=vec[v];
        if (l>0) minans=min(minans,abs((vec[vx[l-1]]-vecu)^(vec[vx[l-1]]-vecv)));
        if (r<vx.size()-1) minans=min(minans,abs((vec[vx[r+1]]-vecu)^(vec[vx[r+1]]-vecv)));
        maxans=max({maxans,abs((vec[vx[0]]-vecu)^(vec[vx[0]]-vecv)),abs((vec[vx.back()]-vecu)^(vec[vx.back()]-vecv))});
        if (i<j) swap(vx[i],vx[j]),pos[u]=j,pos[v]=i;
    }
    return {minans,maxans};
}

// 判断多条线段是否有交点
// 扫描线，复杂度 O(nlogn)
bool segs_inter(const vector&lt;Segment&gt; &segs)
{
    if (segs.empty()) return false;
    using seq_t=tuple<point_t,int,Segment>;
    const auto seqcmp=[](const seq_t &u, const seq_t &v)
    {
        const auto [u0,u1,u2]=u;
        const auto [v0,v1,v2]=v;
        if (abs(u0-v0)<=eps) return make_pair(u1,u2)<make_pair(v1,v2);
```

```
        return u0<v0-eps;
    };
    vector<seq_t> seq;
    for (auto seg:segs)
    {
        if (seg.a.x>seg.b.x+eps) swap(seg.a,seg.b);
        seq.push_back({seg.a.x,0,seg});
        seq.push_back({seg.b.x,1,seg});
    }
    sort(seq.begin(),seq.end(),seqcmp);
    point_t x_now;
    auto cmp=[&](const Segment &u, const Segment &v)
    {
        if (abs(u.a.x-u.b.x)<=eps || abs(v.a.x-v.b.x)<=eps) return u.a.y<v.a.y-eps;
        return ((x_now-u.a.x)*(u.b.y-u.a.y)+u.a.y*(u.b.x-u.a.x))*(v.b.x-v.a.x)<((x_now-v.a.x)*(v.b.y-v.a.y)+v.a.y*(v.b.x-v.a.x))*(u.b.x-u.a.x)-eps;
    };
    multiset&lt;Segment,decltype(cmp)&gt; s{cmp};
    for (const auto [x,o,seg]:seq)
    {
        x_now=x;
        const auto it=s.lower_bound(seg);
        if (o==0)
        {
            if (it!=s.end() && seg.is_inter(*it)) return true;
            if (it!=s.begin() && seg.is_inter(*prev(it))) return true;
            s.insert(seg);
        }
        else
        {
            if (next(it)!=s.end() && it!=s.begin() && (*prev(it)).is_inter(*next(it))) return true;
            s.erase(it);
        }
    }
    return false;
}
```

```
// 多边形面积并
// 轮廓积分，复杂度 O(n^2logn)，n 为边数
// ans[i] 表示被至少覆盖了 i+1 次的区域的面积
vector<long double> area_union(const vector<Polygon> &polys)
{
    const size_t siz=polys.size();
    vector<vector<pair<Point,Point>>> segs(siz);
    const auto check=[](const Point &u,const Segment
&e){return !((u<e.a && u<e.b) || (u>e.a && u>e.b));};

    auto cut_edge=[&](const Segment &e,const size_t i)
    {
        const Line le{e.a,e.b-e.a};
        vector<pair<Point,int>> evt;
        evt.push_back({e.a,0}); evt.push_back({e.b,0});
        for (size_t j=0;j<polys.size();j++)
        {
            if (i==j) continue;
            const auto &pj=polys[j];
            for (size_t k=0;k<pj.p.size();k++)
            {
                const Segment s={pj.p[k],pj.p[pj.nxt(k)]};
                if (le.toleft(s.a)==0 && le.toleft(s.b)==0)
                {
                    evt.push_back({s.a,0});
                    evt.push_back({s.b,0});
                }
                else if (s.is_inter(le))
                {
                    const Line ls{s.a,s.b-s.a};
                    const Point u=le.inter(ls);
                    if (le.toleft(s.a)<0 && le.toleft(s.b)>=0)
evt.push_back({u,-1});
                    else if (le.toleft(s.a)>=0 && le.toleft(s.b)<0)
evt.push_back({u,1});
                }
            }
        }
```

```
        sort(evt.begin(),evt.end());
        if (e.a>e.b) reverse(evt.begin(),evt.end());
        int sum=0;
        for (size_t i=0;i<evt.size();i++)
        {
            sum+=evt[i].second;
            const Point u=evt[i].first,v=evt[i+1].first;
            if (!(u==v) && check(u,e) && check(v,e))
segs[sum].push_back({u,v});
            if (v==e.b) break;
        }
    };

    for (size_t i=0;i<polys.size();i++)
    {
        const auto &pi=polys[i];
        for (size_t k=0;k<pi.p.size();k++)
        {
            const Segment ei={pi.p[k],pi.p[pi.nxt(k)]};
            cut_edge(ei,i);
        }
    }
    vector<long double> ans(siz);
    for (size_t i=0;i<siz;i++)
    {
        long double sum=0;
        sort(segs[i].begin(),segs[i].end());
        int cnt=0;
        for (size_t j=0;j<segs[i].size();j++)
        {
            if (j>0 && segs[i][j]==segs[i][j-1])
segs[i+(++cnt)].push_back(segs[i][j]);
            else cnt=0,sum+=segs[i][j].first^segs[i][j].second;
        }
        ans[i]=sum/2;
    }
    return ans;
}
```

```
// 圆面积并
// 轮廓积分，复杂度 O(n^2logn)
// ans[i] 表示被至少覆盖了 i+1 次的区域的面积
vector<long double> area_union(const vector<Circle> &circs)
{
    const size_t siz=circs.size();
    using arc_t=tuple<Point,long double,long double,long double>;
    vector<vector<arc_t>> arcs(siz);
    const auto eq=[](const arc_t &u,const arc_t &v)
    {
        const auto [u1,u2,u3,u4]=u;
        const auto [v1,v2,v3,v4]=v;
        return u1==v1 && abs(u2-v2)<=eps && abs(u3-v3)<=eps && abs(u4-
v4)<=eps;
    };

    auto cut_circ=[&](const Circle &ci,const size_t i)
    {
        vector<pair<long double,int>> evt;
        evt.push_back({-PI,0}); evt.push_back({PI,0});
        int init=0;
        for (size_t j=0;j<circs.size();j++)
        {
            if (i==j) continue;
            const Circle &cj=circs[j];
            if (ci.r<cj.r-eps && ci.relation(cj)>=3) init++;
            const auto inters=ci.inter(cj);
            if (inters.size()==1) evt.push_back({atan2l((inters[0]-
ci.c).y,(inters[0]-ci.c).x),0});
            if (inters.size()==2)
            {
                const Point dl=inters[0]-ci.c,dr=inters[1]-ci.c;
                long double
argl=atan2l(dl.y,dl.x),argr=atan2l(dr.y,dr.x);
                if (abs(argl+PI)<=eps) argl=PI;
                if (abs(argr+PI)<=eps) argr=PI;
                if (argl>argr+eps)
                {
                    evt.push_back({argl,1}); evt.push_back({PI,-1});
```

```
                    evt.push_back({-PI,1}); evt.push_back({argr,-1});
                }
                else
                {
                    evt.push_back({argl,1});
                    evt.push_back({argr,-1});
                }
            }
        }
        sort(evt.begin(),evt.end());
        int sum=init;
        for (size_t i=0;i<evt.size();i++)
        {
            sum+=evt[i].second;
            if (abs(evt[i].first-evt[i+1].first)>eps)
arcs[sum].push_back({ci.c,ci.r,evt[i].first,evt[i+1].first});
            if (abs(evt[i+1].first-PI)<=eps) break;
        }
    };

    const auto oint=[](const arc_t &arc)
    {
        const auto [cc,cr,l,r]=arc;
        if (abs(r-l-PI-PI)<=eps) return 2.0l*PI*cr*cr;
        return cr*cr*(r-l)+cc.x*cr*(sin(r)-sin(l))-cc.y*cr*(cos(r)-
cos(l));
    };

    for (size_t i=0;i<circs.size();i++)
    {
        const auto &ci=circs[i];
        cut_circ(ci,i);
    }
    vector<long double> ans(siz);
    for (size_t i=0;i<siz;i++)
    {
        long double sum=0;
        sort(arcs[i].begin(),arcs[i].end());
        int cnt=0;
```

```
        for (size_t j=0;j<arcs[i].size();j++)
        {
            if (j>0 && eq(arcs[i][j],arcs[i][j-1]))
arcs[i+(++cnt)].push_back(arcs[i][j]);
            else cnt=0,sum+=oint(arcs[i][j]);
        }
        ans[i]=sum/2;
    }
    return ans;
}
```

动态规划

数位 dp

工具需要检测的号码特征有两个：号码中要出现至少 3 个相邻的相同数字；号码中不能同时出现 8 和 4。号码必须同时包含两个特征才满足条件。满足条件的号码例如：13000988721、23333333333、14444101000。而不满足条件的号码例如：1015400080、10010012022。

手机号码一定是 11 位数，前不含前导的 0。工具接收两个数 L 和 R，自动统计出 $[L, R]$ 区间内所有满足条件的号码数量。L 和 R 也是 11 位的手机号码。

输入格式

输入文件内容只有一行，为空格分隔的 2 个正整数 L, R。

输出格式

输出文件内容只有一行，为 1 个整数，表示满足条件的手机号数量。

输入输出样例

输入 #1 复制

```
12121284000 12121285550
```

输出 #1 复制

```
5
```

代码：

```
#include<bits/stdc++.h>
#define IOS ios::sync_with_stdio(false)

using namespace std;
typedef long long ll;
const int N = 15,M=200010;
const ll mod=1000000007;
const ll inf=0x3f3f3f3f3f3f3f3f;

ll v[N],cnt=0;
ll f[N][2][11][11][2][2][2];

ll dfs(int len,bool issmall,int last1,int last2,bool ok,bool
eight,bool four)
{
    ll res=0;
    if(eight&&four) return 0;
    if(len<=0) return ok;
    if(f[len][issmall][last1][last2][ok][eight][four]!=-1)
```

```
        return f[len][issmall][last1][last2][ok][eight][four];
    for(int i=0;i<10;i++)
    {
        if(!issmall&&i>v[len]) break;
        res+=dfs(len-
1,issmall||(i<v[len]),i,last1,ok||(last1==last2&&i==last1),eight||(i==
8),four||(i==4));
    }
    return f[len][issmall][last1][last2][ok][eight][four]=res;
}

ll work(ll n)
{
    cnt=0;
    if(n<1e10) return 0;
    while(n)
    {
        v[++cnt]=n%10;
        n/=10;
    }
    memset(f,-1,sizeof f);
//  ll sum=0;
    ll res=0;
    for(int i=1;i<=v[cnt];i++)
    {
        res=res+dfs(cnt-1,i<v[cnt],i,-1,0,i==8,i==4);
    }
    return res;
}

signed main()
{
    IOS;
    ll n,m;
    cin>>n>>m;
    cout<<work(m)-work(n-1);
}
```