

刘阳

阳渣的acm模版

目录

[图论 4](#_Toc501092854)

[二分图bfs判断 4](#_Toc501092855)

[使用优先队列优化Dijkstra 5](#_Toc501092856)

[vector版本 5](#_Toc501092857)

[邻接表版本 6](#_Toc501092858)

[SPFA 8](#_Toc501092859)

[图的割点、桥和双连通分支的基本概念 9](#_Toc501092860)

[无向图的割点和桥 11](#_Toc501092861)

[二分图匹配 13](#_Toc501092862)

[平面最近点对 14](#_Toc501092863)

[最小树形图 16](#_Toc501092864)

[生成树计数 18](#_Toc501092865)

[生成树计数（取模） 23](#_Toc501092866)

[最小生成树 24](#_Toc501092867)

[二维曼哈顿距离最小生成树 26](#_Toc501092868)

[Tarjan 30](#_Toc501092869)

[最大团 31](#_Toc501092870)

[LCA 33](#_Toc501092871)

[全局最小割 35](#_Toc501092872)

[最小割树 36](#_Toc501092873)

[ISAP 39](#_Toc501092874)

[DINIC 42](#_Toc501092875)

[费用流 44](#_Toc501092876)

[上下界网络流 46](#_Toc501092877)

[字符串 49](#_Toc501092878)

[后缀数组 49](#_Toc501092879)

[SAM（Reskip） 51](#_Toc501092880)

[扩展KMP 53](#_Toc501092881)

[AC自动机 54](#_Toc501092882)

[KMP 57](#_Toc501092883)

[Manacher 58](#_Toc501092884)

[大小写转换 59](#_Toc501092885)

[数据结构 60](#_Toc501092886)

[树状数组 60](#_Toc501092887)

[二维树状数组 60](#_Toc501092888)

[字典树 62](#_Toc501092889)

[并查集 63](#_Toc501092890)

[莫队 65](#_Toc501092891)

[DLX（精确覆盖） 66](#_Toc501092892)

[DLX（重复覆盖） 68](#_Toc501092893)

[ST表（RMQ） 70](#_Toc501092894)

[线段树区间合并 72](#_Toc501092895)

[矩形面积并 75](#_Toc501092896)

[矩形面积交 77](#_Toc501092897)

[矩形轮廓 80](#_Toc501092898)

[树链剖分（点权） 82](#_Toc501092899)

[树链剖分（边权） 86](#_Toc501092900)

[主席树 90](#_Toc501092901)

[点分治 92](#_Toc501092902)

[LCT 95](#_Toc501092903)

[Splay树 101](#_Toc501092904)

[KDTree 109](#_Toc501092905)

[Treap 113](#_Toc501092906)

[线性基 119](#_Toc501092907)

[数论 120](#_Toc501092908)

[CRT（互质） 120](#_Toc501092909)

[CRT（不互质） 122](#_Toc501092910)

[素数筛 124](#_Toc501092911)

[大区间素数筛 124](#_Toc501092912)

[素因子分解 126](#_Toc501092913)

[扩展欧几里得 127](#_Toc501092914)

[矩阵快速幂 127](#_Toc501092915)

[组合数 128](#_Toc501092916)

[高斯消元（int） 130](#_Toc501092917)

[高斯消元（double） 132](#_Toc501092918)

[FFT 133](#_Toc501092919)

[FWT 136](#_Toc501092920)

[NTT 137](#_Toc501092921)

[Lucas定理 138](#_Toc501092922)

[数位DP 139](#_Toc501092923)

[默慈金数 140](#_Toc501092924)

[错排 140](#_Toc501092925)

[单纯形 141](#_Toc501092926)

[Meisell-Lehmer算法（1e11内素数） 144](#_Toc501092927)

[计算几何 145](#_Toc501092928)

[海伦公式 145](#_Toc501092929)

[正n面体体积 145](#_Toc501092930)

[Point（三维） 146](#_Toc501092931)

[Point 147](#_Toc501092932)

[Line 147](#_Toc501092933)

[判断线段相交 148](#_Toc501092934)

[判断直线和线段相交 149](#_Toc501092935)

[点到直线距离 149](#_Toc501092936)

[点到线段距离 149](#_Toc501092937)

[多边形面积 150](#_Toc501092938)

[判断点在线段上 150](#_Toc501092939)

[判断点在凸多边形内 150](#_Toc501092940)

[数论公式 151](#_Toc501092941)

[欧拉定理 151](#_Toc501092942)

[Lucas定理 151](#_Toc501092943)

[中国剩余定理 152](#_Toc501092944)

[费马小定理 152](#_Toc501092945)

[拉姆塞理论 152](#_Toc501092946)

[范德蒙恒等式 152](#_Toc501092947)

[莫比乌斯反演 153](#_Toc501092948)

[扩展欧几里德 154](#_Toc501092949)

[勒让德定理 155](#_Toc501092950)

[二分图 155](#_Toc501092951)

[差分约束 157](#_Toc501092952)

[其他 163](#_Toc501092953)

[Vim 163](#_Toc501092954)

[JAVA大数 164](#_Toc501092955)

# 图论

## 二分图bfs判断

//用'临点填色法'判断，相邻点异色，发现同色则不成立

bool BFS()//二分图BFS判断

{

queue<int> Q;

int judge[MAXN];// 判断二分图时 0-1表

int v;

for(int i=0;i<=n;i++)

judge[i]=-1;

for(int j=1;j<=n;j++)

{

if(judge[j]==-1)

{

Q.push(j);

judge[j]=0;

uN.push\_back(j);//加入左半部分

while(!Q.empty())

{

v=Q.front();

Q.pop();

for(int i=1;i<=n;i++)

{

if(g[v][i])

{

if(judge[i]==-1)

{

judge[i]=(judge[v]+1)%2;

Q.push(i);

if(judge[i]==0)

{

uN.push\_back(i);//加入左半部分

}

else

{

vN.push\_back(i);//加入右半部分

}

}

else

{

if(judge[i]==judge[v])

return false;

}

}

}

}

}

}

return true;

}

## 使用优先队列优化Dijkstra

vector版本

/\*

\* 使用优先队列优化Dijkstra算法

\* 复杂度O(ElogE)

\* 注意对vector<Edge>E[MAXN]进行初始化后加边

\*/

const int INF=0x3f3f3f3f;

const int MAXN=1000010;

struct qnode

{

int v;

int c;

qnode(int \_v=0,int \_c=0):v(\_v),c(\_c){}

bool operator <(const qnode &r)const

{

return c>r.c;

}

};

struct Edge

{

int v,cost;

Edge(int \_v=0,int \_cost=0):v(\_v),cost(\_cost){}

};

vector<Edge>E[MAXN];

bool vis[MAXN];

int dist[MAXN];

void Dijkstra(int n,int start)//点的编号从1开始

{

memset(vis,false,sizeof(vis));

for(int i=1;i<=n;i++)dist[i]=INF;

priority\_queue<qnode>que;

while(!que.empty())que.pop();

dist[start]=0;

que.push(qnode(start,0));

qnode tmp;

while(!que.empty())

{

tmp=que.top();

que.pop();

int u=tmp.v;

if(vis[u])continue;

vis[u]=true;

for(int i=0;i<E[u].size();i++)

{

int v=E[tmp.v][i].v;

int cost=E[u][i].cost;

if(!vis[v]&&dist[v]>dist[u]+cost)

{

dist[v]=dist[u]+cost;

que.push(qnode(v,dist[v]));

}

}

}

}

void addedge(int u,int v,int w)

{

E[u].push\_back(Edge(v,w));

}

邻接表版本

typedef long long ll;

const ll INF=0x3f3f3f3f3f3f3f3f;

const int MAXN=20010;

int cnt=0;

struct qnode

{

int v;

ll c;

qnode(int \_v=0,ll \_c=0):v(\_v),c(\_c){}

bool operator <(const qnode &r)const

{

return c>r.c;

}

};

struct Edge

{

int u,v,nxt;

ll cost;

Edge(){}

Edge(int \_u,int \_v,ll \_cost,int \_nxt):u(\_u),v(\_v),cost(\_cost),nxt(\_nxt){}

};

Edge E[MAXN];

bool book[MAXN],vis[MAXN];

int head[MAXN];

ll dist[MAXN];

void Dijkstra(int n,int start)//点的编号从1开始

{

memset(vis,false,sizeof(vis));

for(int i=0;i<n;i++)dist[i]=INF;

priority\_queue<qnode>que;

while(!que.empty())que.pop();

dist[start]=0;

que.push(qnode(start,0));

qnode tmp;

while(!que.empty())

{

tmp=que.top();

que.pop();

int u=tmp.v;

if(vis[u])continue;

vis[u]=true;

for(int i=head[u];~i;i=E[i].nxt)

{

int v=E[i].v;

int cost=E[i].cost;

if(!vis[v]&&dist[v]>dist[u]+cost)

{

dist[v]=dist[u]+cost;

que.push(qnode(v,dist[v]));

}

}

}

}

void addedge(int u,int v,ll w)

{

E[cnt]=Edge(u,v,w,head[u]);

head[u]=cnt++;

E[cnt]=Edge(v,u,w,head[v]);

head[v]=cnt++;

}

## SPFA

/\*

\* 单源最短路SPFA

\* 时间复杂度 0(kE)

\* 这个是队列实现，有时候改成栈实现会更加快，很容易修改

\* 这个复杂度是不定的

\*/

const int MAXN=1010;

const int INF=0x3f3f3f3f;

struct Edge

{

int v;

int cost;

Edge(int \_v=0,int \_cost=0):v(\_v),cost(\_cost){}

};

vector<Edge>E[MAXN];

void addedge(int u,int v,int w)

{

E[u].push\_back(Edge(v,w));

}

bool vis[MAXN];//在队列标志

int cnt[MAXN];//每个点的入队列次数

int dist[MAXN];

bool SPFA(int start,int n)

{

memset(vis,false,sizeof(vis));

for(int i=1;i<=n;i++)dist[i]=INF;

vis[start]=true;

dist[start]=0;

queue<int>que;

while(!que.empty())que.pop();

que.push(start);

memset(cnt,0,sizeof(cnt));

cnt[start]=1;

while(!que.empty())

{

int u=que.front();

que.pop();

vis[u]=false;

for(int i=0;i<E[u].size();i++)

{

int v=E[u][i].v;

if(dist[v]>dist[u]+E[u][i].cost)

{

dist[v]=dist[u]+E[u][i].cost;

if(!vis[v])

{

vis[v]=true;

que.push(v);

if(++cnt[v]>n)return false;

//cnt[i]为入队列次数，用来判定是否存在负环回路

}

}

}

}

return true;

}

## 图的割点、桥和双连通分支的基本概念

[点连通度与边连通度]

在一个无向连通图中，如果有一个顶点集合，删除这个顶点集合，以及这个集合中所有顶点相关联的边以后，原图变成多个连通块，就称这个点集为割点集合。一个图的点连通度的定义为，最小割点集合中的顶点数。

类似的，如果有一个边集合，删除这个边集合以后，原图变成多个连通块，就称这个点集为割边集合。一个图的边连通度的定义为，最小割边集合中的边数。

[双连通图、割点与桥]

如果一个无向连通图的点连通度大于1，则称该图是点双连通的(point biconnected)，简称双连通或重连通。一个图有割点，当且仅当这个图的点连通度为1，则割点集合的唯一元素被称为割点(cut point)，又叫关节点(articulation point)。 如果一个无向连通图的边连通度大于1，则称该图是边双连通的(edge biconnected)，简称双连通或重连通。

一个图有桥，当且仅当这个图的边连通度为1，则割边集合的唯一元素被称为桥(bridge)，又叫关节边(articulation edge)。 可以看出，点双连通与边双连通都可以简称为双连通，它们之间是有着某种联系的，下文中提到的双连通，均既可指点双连通，又可指边双连通。

[双连通分支]

在图G的所有子图G'中，如果G'是双连通的，则称G'为双连通子图。如果一个双连通子图G'它不是任何一个双连通子图的真子集，则G'为极大双连通子图。双连通分支(biconnected component)，或重连通分支，就是图的极大双连通子图。特殊的，点双连通分支又叫做块。

[求割点与桥]

该算法是R.Tarjan发明的。对图深度优先搜索，定义DFS(u)为u在搜索树（以下简称为树）中被遍历到的次序号。定义Low(u)为u或u的子树中能通过非父子边追溯到的最早的节点，即DFS序号最小的节点。根据定义，则有：

Low(u)=Min { DFS(u) DFS(v) (u,v)为后向边(返祖边) 等价于 DFS(v)<DFS(u)且v不为u的父亲节点 Low(v) (u,v)为树枝边(父子边) }

一个顶点u是割点，当且仅当满足(1)或(2) (1) u为树根，且u有多于一个子树。 (2) u不为树根，且满足存在(u,v)为树枝边(或称父子边，即u为v在搜索树中的父亲)，使得DFS(u)<=Low(v)。

一条无向边(u,v)是桥，当且仅当(u,v)为树枝边，且满足DFS(u)<Low(v)。

[求双连通分支]

下面要分开讨论点双连通分支与边双连通分支的求法。 对于点双连通分支，实际上在求割点的过程中就能顺便把每个点双连通分支求出。建立一个栈，存储当前双连通分支，在搜索图时，每找到一条树枝边或后向边(非横叉边)，就把这条边加入栈中。如果遇到某时满足DFS(u)<=Low(v)，说明u是一个割点，同时把边从栈顶一个个取出，直到遇到了边(u,v)，取出的这些边与其关联的点，组成一个点双连通分支。割点可以属于多个点双连通分支，其余点和每条边只属于且属于一个点双连通分支。

对于边双连通分支，求法更为简单。只需在求出所有的桥以后，把桥边删除，原图变成了多个连通块，则每个连通块就是一个边双连通分支。桥不属于任何一个边双连通分支，其余的边和每个顶点都属于且只属于一个边双连通分支。

[构造双连通图]

一个有桥的连通图，如何把它通过加边变成边双连通图？方法为首先求出所有的桥，然后删除这些桥边，剩下的每个连通块都是一个双连通子图。把每个双连通子图收缩为一个顶点，再把桥边加回来，最后的这个图一定是一棵树，边连通度为1。

统计出树中度为1的节点的个数，即为叶节点的个数，记为leaf。则至少在树上添加(leaf+1)/2条边，就能使树达到边二连通，所以至少添加的边数就是(leaf+1)/2。具体方法为，首先把两个最近公共祖先最远的两个叶节点之间连接一条边，这样可以把这两个点到祖先的路径上所有点收缩到一起，因为一个形成的环一定是双连通的。然后再找两个最近公共祖先最远的两个叶节点，这样一对一对找完，恰好是(leaf+1)/2次，把所有点收缩到了一起。

## 无向图的割点和桥

/\*

\* 求 无向图的割点和桥

\* 可以找出割点和桥，求删掉每个点后增加的连通块。

\* 需要注意重边的处理，可以先用矩阵存，再转邻接表，或者进行判重

\*/·

const int MAXN = 10010;

const int MAXM = 100010;

struct Edge

{

int to,next;

bool cut;//是否为桥的标记

}edge[MAXM];

int head[MAXN],tot;

int Low[MAXN],DFN[MAXN],Stack[MAXN];

int Index,top;

bool Instack[MAXN];

bool cut[MAXN];

int add\_block[MAXN];//删除一个点后增加的连通块

int bridge;

void addedge(int u,int v)

{

edge[tot].to=v;edge[tot].next=head[u];edge[tot].cut=false;

head[u]=tot++;

}

void Tarjan(int u,int pre)

{

int v;

Low[u]=DFN[u]=++Index;

Stack[top++]=u;

Instack[u]=true;

int son=0;

for(int i=head[u];i!=-1;i=edge[i].next)

{

v=edge[i].to;

if(v==pre)continue;

if(!DFN[v])

{

son++;

Tarjan(v,u);

if(Low[u]>Low[v])Low[u]=Low[v];

//桥

//一条无向边(u,v)是桥，当且仅当(u,v)为树枝边，且满足DFS(u)<Low(v)。

if(Low[v]>DFN[u])

{

bridge++;

edge[i].cut=true;

edge[i^1].cut=true;

}

//割点

//一个顶点u是割点，当且仅当满足(1)或(2) (1) u为树根，且u有多于一个子树。

//(2) u不为树根，且满足存在(u,v)为树枝边(或称父子边，

//即u为v在搜索树中的父亲)，使得DFS(u)<=Low(v)

if(u!=pre&&Low[v]>=DFN[u])//不是树根

{

cut[u]=true;

add\_block[u]++;

}

}

else if(Low[u]>DFN[v])

Low[u]=DFN[v];

}

//树根，分支数大于1

if(u == pre && son > 1)cut[u]=true;

if(u == pre)add\_block[u]=son-1;

Instack[u]=false;

top--;

}

void Tarjan(int u,int pre,int del) //无向图有重边求桥

{

int v;

int flag=0;

Low[u]=DFN[u]=++Index;

for(int i=head[u];i!=-1;i=edge[i].next)

{

v=edge[i].to;

if(v==pre&&!flag)//考虑重边

{

flag=1;

continue;

}

if((i>>1)==del)continue;

if(!DFN[v])

{

Tarjan(v,u,del);

if(Low[u]>Low[v])Low[u]=Low[v];

if(Low[v]>DFN[u])

{

bridge[i>>1]=1;

}

}

else if(Low[u]>DFN[v])

Low[u]=DFN[v];

}

}

## 二分图匹配

/\*

\* 匈牙利算法邻接表形式

\* 使用前用init()进行初始化，给uN赋值

\* 加边使用函数addedge(u,v)

\*

\*/

const int MAXN=5010;//点数的最大值

const int MAXM=50010;//边数的最大值

struct Edge

{

int to,next;

}edge[MAXM];

int head[MAXN],tot;

void init()

{

tot=0;

memset(head,-1,sizeof(head));

}

void addedge(int u,int v)

{

edge[tot].to=v;edge[tot].next=head[u];

head[u]=tot++;

}

int linker[MAXN];

bool used[MAXN];

int uN;

bool dfs(int u)

{

for(int i=head[u];i!=-1;i=edge[i].next)

{

int v=edge[i].to;

if(!used[v])

{

used[v]=true;

if(linker[v]==-1||dfs(linker[v]))

{

linker[v]=u;

return true;

}

}

}

return false;

}

int hungary()

{

int res=0;

memset(linker,-1,sizeof(linker));

for(int u = 0; u < uN;u++)//点的编号0~uN-1

{

memset(used,false,sizeof(used));

if(dfs(u))res++;

}

return res;

}

## 平面最近点对

struct Point

{

long long x,y;

bool operator < (const Point &rhs) const

{

return x<rhs.x||(x==rhs.x&&y<rhs.y);

}

};

typedef pair<long long, pair<int, int> > llii;

Point point[100000];

int tmpt[100000];

bool cmpy(const int &a, const int &b) { return point[a].y < point[b].y; }

llii dis(int i, int j)

{

return make\_pair((point[i].x-point[j].x)\*(point[i].x-point[j].x)+(point[i].y-point[j].y)\*(point[i].y-point[j].y),make\_pair(i, j));

}

llii Closest\_Pair(int left, int right)

{

llii d=make\_pair(INF,make\_pair(-1, -1));

if(left==right)

return d;

if(left+1==right)

return dis(left, right);

int mid=(left+right)>>1;

llii d1=Closest\_Pair(left,mid);

llii d2=Closest\_Pair(mid+1,right);

d=min(d1,d2);

int i,j,k=0;

for(i=left;i<=right;i++)

{

if(abs(point[mid].x-point[i].x)<=d.first)

tmpt[k++]=i;

}

sort(tmpt,tmpt+k,cmpy);

for(i=0;i<k;i++)

{

for(j=i+1;j<k&&point[tmpt[j]].y-point[tmpt[i]].y<d.first;j++)

{

llii d3=dis(tmpt[i],tmpt[j]);

if(d>d3)

d=d3;

}

}

return d;

}

//首先先按照x排序！！！！！然后再用

## 最小树形图

#include<cstdio>

#include<cstring>

#include<iostream>

#include<algorithm>

using namespace std;

/\*

\* 最小树形图

\* int型

\* 复杂度O(NM)

\* 点从0开始

\*/

const int INF=0x3f3f3f3f;

const int MAXN=1010;

const int MAXM=40010;

struct Edge

{

int u,v,cost;

};

Edge edge[MAXM];

int pre[MAXN],id[MAXN],visit[MAXN],in[MAXN];

int zhuliu(int root,int n,int m)

{

int res=0,u,v;

while(1)

{

for(int i=0;i<n;i++)

in[i]=INF;

for(int i=0;i<m;i++)

if(edge[i].u!=edge[i].v&&edge[i].cost<in[edge[i].v])

{

pre[edge[i].v]=edge[i].u;

in[edge[i].v]=edge[i].cost;

}

for(int i=0;i<n;i++)

if(i!=root&&in[i]==INF)

return -1;//不存在最小树形图

int tn=0;

memset(id,-1,sizeof(id));

memset(visit,-1,sizeof(visit));

in[root]=0;

for(int i=0;i<n;i++)

{

res+=in[i];

v=i;

while(visit[v]!=i&&id[v]==-1&&v!=root)

{

visit[v]=i;

v=pre[v];

}

if(v!=root&&id[v]==-1)

{

for(int u=pre[v];u!=v;u=pre[u])

id[u]=tn;

id[v]=tn++;

}

}

if(tn==0)break;//没有有向环

for(int i=0;i<n;i++)

if(id[i]==-1)

id[i]=tn++;

for(int i=0;i<m;)

{

v=edge[i].v;

edge[i].u=id[edge[i].u];

edge[i].v=id[edge[i].v];

if(edge[i].u!=edge[i].v)

edge[i++].cost-=in[v];

else

swap(edge[i],edge[--m]);

}

n=tn;

root=id[root];

}

return res;

}

int g[MAXN][MAXN];

int main()

{

int n,m;

int iCase=0;

int T;

scanf("%d",&T);

while(T--)

{

iCase++;

scanf("%d%d",&n,&m);

for(int i=0;i<n;i++)

for(int j=0;j<n;j++)

g[i][j]=INF;

int u,v,cost;

while(m--)

{

scanf("%d%d%d",&u,&v,&cost);

if(u==v)continue;

g[u][v]=min(g[u][v],cost);

}

int L=0;

for(int i=0;i<n;i++)

for(int j=0;j<n;j++)

if(g[i][j]<INF)

{

edge[L].u=i;

edge[L].v=j;

edge[L++].cost=g[i][j];

}

int ans=zhuliu(0,n,L);

printf("Case #%d: ",iCase);

if(ans==-1)printf("Possums!\n");

else printf("%d\n",ans);

}

return 0;

}

## 生成树计数

/\*

\*算法引入：

\*给定一个含有N个结点M条边的无向图,求它最小生成树的个数t(G);

\*

\*算法思想：

\*抛开“最小”的限制不看,如果只要求求出所有生成树的个数,是可以利用Matrix-Tree定理解决的;

\*Matrix-Tree定理此定理利用图的Kirchhoff矩阵,可以在O(N3)时间内求出生成树的个数;

\*

\*kruskal算法：

\*将图G={V,E}中的所有边按照长度由小到大进行排序,等长的边可以按照任意顺序;

\*初始化图G’为{V,?},从前向后扫描排序后的边,如果扫描到的边e在G’中连接了两个相异的连通块,则将它插入G’中;

\*最后得到的图G’就是图G的最小生成树;

\*

\*由于kruskal按照任意顺序对等长的边进行排序,则应该将所有长度为L0的边的处理当作一个阶段来整体看待;

\*令kruskal处理完这一个阶段后得到的图为G0,如果按照不同的顺序对等长的边进行排序,得到的G0也是不同;

\*虽然G0可以随排序方式的不同而不同,但它们的连通性都是一样的,都和F0的连通性相同(F0表示插入所有长度为L0的边后形成的图);

\*

\*在kruskal算法中的任意时刻,并不需要关注G’的具体形态,而只要关注各个点的连通性如何(一般是用并查集表示);

\*所以只要在扫描进行完第一阶段后点的连通性和F0相同,且是通过最小代价到达这一状态的,接下去都能找到最小生成树;

\*

\*经过上面的分析,可以看出第一个阶段和后面的工作是完全独立的;

\*第一阶段需要完成的任务是使G0的连通性和F0一样,且只能使用最小的代价;

\*计算出这一阶段的方案数,再乘上完成后续事情的方案数,就是最终答案;

\*

\*由于在第一个阶段中,选出的边数是一定的,所有边的长又都为L0;

\*所以无论第一个阶段如何进行代价都是一样的,那么只需要计算方案数就行了;

\*此时Matrix-Tree定理就可以派上用场了,只需对F0中的每一个连通块求生成树个数再相乘即可;

\*

\*Matrix-Tree定理:

\*G的所有不同的生成树的个数等于其Kirchhoff矩阵C[G]任何一个n-1阶主子式的行列式的绝对值；

\*n-1阶主子式就是对于r(1≤r≤n),将C[G]的第r行,第r列同时去掉后得到的新矩阵,用Cr[G]表示;

\*

\*算法举例：

\*HDU4408(Minimum Spanning Tree)

\*

\*题目地址：

\*http://acm.hdu.edu.cn/showproblem.php?pid=4408

\*

\*题目大意：

\*给定一个含有N个结点M条边的无向图,求它最小生成树的个数,所得结果对p取模;

\*\*/

#include<iostream>

#include<cstdio>

#include<cmath>

#include<cstring>

#include<cstdlib>

#include<queue>

#include<algorithm>

#include<vector>

using namespace std;

const int N=111;

const int M=1111;

typedef long long LL;

struct Edges

{

int a,b,c;

bool operator<(const Edges & x)const

{

return c<x.c;

}

} edge[M];

int n,m;

int mod;

LL f[N],U[N],vist[N];//f,U都是并查集，U是每组边临时使用

LL G[N][N],C[N][N];//G顶点之间的关系，C为生成树计数用的Kirchhoff矩阵

vector<int>V[N];//记录每个连通分量

int Find(int x,LL f[])

{

if(x==f[x])

return x;

else

return Find(f[x],f);

}

LL det(LL a[][N],int n)//生成树计数:Matrix-Tree定理

{

for(int i=0; i<n; i++)

for(int j=0; j<n; j++)

a[i][j]%=mod;

int ret=1;

for(int i=1; i<n; i++)

{

for(int j=i+1; j<n; j++)

while(a[j][i])

{

int t=a[i][i]/a[j][i];

for(int k=i; k<n; k++)

a[i][k]=(a[i][k]-a[j][k]\*t)%mod;

for(int k=i; k<n; k++)

swap(a[i][k],a[j][k]);

ret=-ret;

}

if(a[i][i]==0)

return 0;

ret=ret\*a[i][i]%mod;

}

return (ret+mod)%mod;

}

void Solve()

{

sort(edge,edge+m);//按权值排序

for(int i=1; i<=n; i++)//初始化并查集

{

f[i]=i;

vist[i]=0;

}

LL Edge=-1;//记录相同的权值的边

LL ans=1;

for(int k=0; k<=m; k++)

{

if(edge[k].c!=Edge||k==m)//一组相等的边,即权值都为Edge的边加完

{

for(int i=1; i<=n; i++)

{

if(vist[i])

{

LL u=Find(i,U);

V[u].push\_back(i);

vist[i]=0;

}

}

for(int i=1; i<=n; i++) //枚举每个连通分量

{

if(V[i].size()>1)

{

for(int a=1; a<=n; a++)

for(int b=1; b<=n; b++)

C[a][b]=0;

int len=V[i].size();

for(int a=0; a<len; a++) //构建Kirchhoff矩阵C

for(int b=a+1; b<len; b++)

{

int a1=V[i][a];

int b1=V[i][b];

C[a][b]=(C[b][a]-=G[a1][b1]);

C[a][a]+=G[a1][b1];//连通分量的度

C[b][b]+=G[a1][b1];

}

LL ret=(LL)det(C,len);

ans=(ans\*ret)%mod;//对V中的每一个连通块求生成树个数再相乘

for(int a=0; a<len; a++)

f[V[i][a]]=i;

}

}

for(int i=1; i<=n; i++)

{

U[i]=f[i]=Find(i,f);

V[i].clear();

}

if(k==m)

break;

Edge=edge[k].c;

}

int a=edge[k].a;

int b=edge[k].b;

int a1=Find(a,f);

int b1=Find(b,f);

if(a1==b1)

continue;

vist[a1]=vist[b1]=1;

U[Find(a1,U)]=Find(b1,U);//并查集操作

G[a1][b1]++;

G[b1][a1]++;

}

int flag=0;

for(int i=2; i<=n&&!flag; i++)

if(U[i]!=U[i-1])

flag=1;

if(m==0)

flag=1;

printf("%I64d\n",flag?0:ans%mod);

}

int main()

{

//freopen("C:\\Users\\Administrator\\Desktop\\kd.txt","r",stdin);

while(scanf("%d%d%d",&n,&m,&mod),n+m+mod)

{

memset(G,0,sizeof(G));

for(int i=1; i<=n; i++)

V[i].clear();

for(int i=0; i<m; i++)

scanf("%d%d%d",&edge[i].a,&edge[i].b,&edge[i].c);

Solve();

}

return 0;

}

## 生成树计数（取模）

const int MOD = 10007;

int INV[MOD];

//求ax = 1( mod m) 的x值，就是逆元(0<a<m)

long long inv(long long a,long long m)

{

if(a==1)return 1;

return inv(m%a,m)\*(m-m/a)%m;

}

struct Matrix

{

int mat[330][330];

void init()

{

memset(mat,0,sizeof(mat));

}

int det(int n)//求行列式的值模上MOD，需要使用逆元

{

for(int i=0;i<n;i++)

for(int j=0;j<n;j++)

mat[i][j]=(mat[i][j]%MOD+MOD)%MOD;

int res=1;

for(int i=0;i<n;i++)

{

for(int j=i;j<n;j++)

if(mat[j][i]!=0)

{

for(int k=i;k<n;k++)

swap(mat[i][k],mat[j][k]);

if(i!=j)

res=(-res+MOD)%MOD;

break;

}

if(mat[i][i]==0)

{

res = -1;//不存在(也就是行列式值为0)

break;

}

for(int j=i+1;j<n;j++)

{

//int mut = (mat[j][i]\*INV[mat[i][i]])%MOD;//打表逆元

int mut = (mat[j][i]\*inv(mat[i][i],MOD))%MOD;

for(int k = i;k < n;k++)

mat[j][k] = (mat[j][k]-(mat[i][k]\*mut)%MOD+MOD)%MOD;

}

res=(res\*mat[i][i])%MOD;

}

return res;

}

};

Matrix ret;

ret.init();

for(int i=0;i<n;i++)

for(int j=0;j<n;j++)

if(i!=j&&g[i][j])

{

ret.mat[i][j]=-1;

ret.mat[i][i]++;

}

printf("%d\n",ret.det(n-1));

## 最小生成树

/\*

\* Kruskal算法求MST

\*/

const int MAXN=110;//最大点数

const int MAXM=10000;//最大边数

int F[MAXN];//并查集使用

struct Edge

{

int u,v,w;

}edge[MAXM];//存储边的信息，包括起点/终点/权值

int tol;//边数，加边前赋值为0

void init()//初始化

{

tol=0;

memset(F,-1,sizeof(F));

memset(edge,0,sizeof(edge));

}

void addedge(int u,int v,int w)

{

edge[tol].u=u;

edge[tol].v=v;

edge[tol++].w=w;

}

bool cmp(Edge a,Edge b)//排序函数，讲边按照权值从小到大排序

{

return a.w<b.w;

}

int find(int x)//并查集使用

{

if(F[x]==-1)return x;

else return F[x]=find(F[x]);

}

int Kruskal(int n)//传入点数，返回最小生成树的权值，如果不连通返回-1

{

memset(F,-1,sizeof(F));

sort(edge,edge+tol,cmp);

int cnt=0;//计算加入的边数

int ans=0;

for(int i=0;i<tol;i++)

{

int u=edge[i].u;

int v=edge[i].v;

int w=edge[i].w;

int t1=find(u);

int t2=find(v);

if(t1!=t2)

{

ans+=w;

F[t1]=t2;

cnt++;

}

if(cnt==n-1)break;

}

if(cnt<n-1)return -1;//不连通

else return ans;

}

## 二维曼哈顿距离最小生成树

#include<bits/stdc++.h>

using namespace std;

const int inf=0x3f3f3f3f;

const int N=1e5+10;

typedef long long ll;

template <class T>

inline bool rd(T &ret)

{

char c;

int sgn;

if(c=getchar(),c==EOF) return 0;

while(c!='-'&&(c<'0'||c>'9'))

c=getchar();

sgn=(c=='-')?-1:1;

ret=(c=='-')?0:(c-'0');

while(c=getchar(),c>='0'&&c<='9')

ret=ret\*10+(c-'0');

ret\*=sgn;

return 1;

}

template <class T>

inline void pt(T x)

{

if(x<0)

{

putchar('-');

x=-x;

}

if(x>9)

pt(x/10);

putchar(x%10+'0');

}

class MST

{

struct Edge

{

int from, to, dis;

Edge(int \_from = 0,int \_to=0,int \_dis=0):from(\_from),to(\_to),dis(\_dis){}

bool operator < (const Edge &x) const

{

return dis<x.dis;

}

}edge[N<<3];

int f[N],tot;

int find(int x)

{

return x==f[x]?x:f[x]=find(f[x]);

}

bool Union(int x,int y)

{

x=find(x);y=find(y);

if(x==y)

return false;

if(x>y)

swap(x,y);

f[x]=y;

return true;

}

public:

void init(int n)

{

for(int i=0;i<=n;i++)f[i]=i;

tot=0;

}

void add(int u,int v,int dis)

{

edge[tot++]=Edge(u,v,dis);

}

ll work()

{//计算最小生成树，返回花费

sort(edge,edge+tot);

ll cost=0;

for(int i=0;i<tot;i++)

if(Union(edge[i].from,edge[i].to))

cost+=edge[i].dis;

return cost;

}

}mst;

struct Point

{//二维平面的点

int x,y,id;

bool operator < (const Point&a) const

{

return x==a.x?y<a.y:x<a.x;

}

}p[N];

class BIT

{//树状数组

int c[N],id[N],maxn;

int lowbit(int x)

{

return x&-x;

}

public:

void init(int n)

{

maxn=n+10;

fill(c,c+maxn+1,inf);

fill(id,id+maxn+1,-1);

}

void updata(int x, int val, int \_id)

{

while(x)

{

if(val<c[x])

{

c[x]=val;id[x]=\_id;

}

x-=lowbit(x);

}

}

int query(int x)

{

int val=inf,\_id=-1;

while (x<=maxn)

{

if(val>c[x])

{

val=c[x];\_id=id[x];

}

x+=lowbit(x);

}

return \_id;

}

}tree;

inline bool cmp(int \*x, int \*y)

{

return \*x<\*y;

}

class Manhattan\_MST

{//复杂度 O(max(N\*1.5,Nlog(N)))

int A[N], B[N];

public:

ll work(int l, int r)

{

mst.init(r);

for(int dir = 1; dir <= 4; dir++)

{

if(dir%2==0)

for(int i=l;i<=r;i++)

swap(p[i].x, p[i].y);

else if(dir == 3)

for(int i=l;i<=r;i++)p[i].y=-p[i].y;

sort(p+l,p+r+1);

for(int i=l;i<=r;i++)

A[i]=B[i]=p[i].y-p[i].x; //离散化

sort(B+1,B+N+1);

int sz=unique(B+1,B+N+1)-B-1;

//初始化反树状数组

tree.init(sz);

for(int i=r;i>=l;i--)

{

int pos=lower\_bound(B+1,B+sz+1, A[i])-B;

int id=tree.query(pos);

if (id!=-1)

mst.add(p[i].id,p[id].id,abs(p[i].x-p[id].x)+abs(p[i].y-p[id].y));

tree.updata(pos, p[i].x + p[i].y,i);

}

}

for(int i=l;i<=r;i++)

p[i].y=-p[i].y;

return mst.work();

}

}m\_mst;

int n;

int main()

{

int Cas=1;

while(cin>>n,n)

{

for(int i=1;i<=n;i++)

rd(p[i].x),rd(p[i].y),p[i].id=i;

printf("Case %d: Total Weight = ",Cas++);

cout<<m\_mst.work(1,n)<<endl;

}

return 0;

}

## Tarjan

/\*

\* Tarjan算法

\* 复杂度O(N+M)

\*/

const int MAXN=20010;//点数

const int MAXM=50010;//边数

struct Edge

{

int to,next;

}edge[MAXM];

int head[MAXN],tot;

int Low[MAXN],DFN[MAXN],Stack[MAXN],Belong[MAXN];//Belong数组的值是1~scc

int Index,top;

int scc;//强连通分量的个数

bool Instack[MAXN];

int num[MAXN];//各个强连通分量包含点的个数，数组编号1~scc

//num数组不一定需要，结合实际情况

void addedge(int u,int v)

{

edge[tot].to=v;edge[tot].next=head[u];head[u]=tot++;

}

void Tarjan(int u)

{

int v;

Low[u]=DFN[u]=++Index;

Stack[top++]=u;

Instack[u]=true;

for(int i = head[u];i != -1;i = edge[i].next)

{

v=edge[i].to;

if(!DFN[v])

{

Tarjan(v);

if(Low[u]>Low[v])Low[u]=Low[v];

}

else if(Instack[v]&&Low[u]>DFN[v])

` Low[u]=DFN[v];

}

if(Low[u]==DFN[u])

{

scc++;

do

{

v=Stack[--top];

Instack[v] = false;

Belong[v] = scc;

num[scc]++;

}

while(v!=u);

}

}

void solve(int N)

{

memset(DFN,0,sizeof(DFN));

memset(Instack,false,sizeof(Instack));

memset(num,0,sizeof(num));

Index=scc=top=0;

for(int i=1;i<=N;i++)

if(!DFN[i])

Tarjan(i);

}

void init()

{

tot=0;

memset(head,-1,sizeof(head));

}

## 最大团

/\*

最大团 = 补图G的最大独立集数

———>最大独立集数 = 补图G'最大团

\*/

//最大团模板

#include<bits/stdc++.h>

const int MAXN=102;

int mx;//最大团数(要初始化为0)

int x[MAXN],tuan[MAXN];

int can[MAXN][MAXN];//can[i]表示在已经确定了经选定的i个点必须在最大团内的前提下还有可能被加进最大团的结点集合

int num[MAXN];//num[i]表示由结点i到结点n构成的最大团的结点数

bool g[MAXN][MAXN];//邻接矩阵(从1开始)

int n,m;

bool dfs(int tot,int cnt)

{

int i,j,k;

if(tot==0)

{

if(cnt>mx)

{

mx=cnt;

for(i=0;i<mx;i++)

{

tuan[i]=x[i];

}

return true;

}

return false;

}

for(i=0;i<tot;i++)

{

if(cnt+(tot-i)<=mx)return false;

if(cnt+num[can[cnt][i]]<= mx)return false;

k=0;

x[cnt]=can[cnt][i];

for(j=i+1;j<tot;j++)

{

if(g[can[cnt][i]][can[cnt][j]])

{

can[cnt+1][k++]=can[cnt][j];

}

}

if(dfs(k,cnt+1))return false;

}

return false;

}

void MaxTuan()

{

int i,j,k;

mx=1;

for(i=n;i>=1;i--)

{

k=0;

x[0]=i;

for(j=i+1;j<=n;j++)

{

if(g[i][j])

{

can[1][k++]=j;

}

}

dfs(k,1);

num[i]=mx;

}

}

int main()

{

while(scanf("%d",&n)&&n)

{

int i,j;

memset(g,0,sizeof(g));

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

{

scanf("%d",&g[i][j]);

}

}

mx=0;

MaxTuan();

printf("%d\n",mx);

}

return 0;

}

## LCA

namespace lca

{

const int MAXN=10010;

const int DEG=20;

struct Edge

{

int to,next;

}edge[MAXN\*2];

int head[MAXN],tot;

void addedge(int u,int v)

{

edge[tot].to=v;

edge[tot].next=head[u];

head[u]=tot++;

}

void init()

{

tot=0;

memset(head,-1,sizeof(head));

}

int fa[MAXN][DEG];//fa[i][j]表示结点i的第2^j个祖先

int deg[MAXN];//深度数组

void BFS(int root)

{

queue<int>que;

deg[root]=0;

fa[root][0]=root;

que.push(root);

while(!que.empty())

{

int tmp=que.front();

que.pop();

for(int i=1;i<DEG;i++)

fa[tmp][i]=fa[fa[tmp][i-1]][i-1];

for(int i=head[tmp];i!=-1;i=edge[i].next)

{

int v=edge[i].to;

if(v==fa[tmp][0])continue;

deg[v]=deg[tmp]+1;

fa[v][0]=tmp;

que.push(v);

}

}

}

int LCA(int u,int v)

{

if(deg[u]>deg[v])swap(u,v);

int hu=deg[u],hv=deg[v];

int tu=u,tv=v;

for(int det=hv-hu,i=0;det;det>>=1,i++)

if(det&1)

tv=fa[tv][i];

if(tu==tv)return tu;

for(int i=DEG-1;i>=0;i--)

{

if(fa[tu][i]==fa[tv][i])

continue;

tu=fa[tu][i];

tv=fa[tv][i];

}

return fa[tu][0];

}

}

## 全局最小割

namespace SW

{

const int N=510,INF=0x3f3f3f3f;

int mpa[N][N],dis[N],v[N];//v数组是马甲数组，dis数组用来表示该点与A集合中所有点之间的边的长度之和

bool vis[N];//用来标记是否该点加入了A集合

int stoer\_wagner(int n)

{

int res=INF;

for(int i=0;i<n;i++)

v[i]=i;//初始马甲为自己

while(n>1)

{

int k,pre=0;//pre用来表示之前加入A集合的点，我们每次都以0点为第一个加入A集合的点

memset(vis,0,sizeof(vis));

memset(dis,0,sizeof(dis));

vis[v[pre]]=true;////

for(int i=1;i<n;i++)

{

k=-1;

for(int j=1;j<n;j++)//根据之前加入的点，要更新dis数组，并找到最大的dis

if(!vis[v[j]])

{

dis[v[j]]+=mpa[v[pre]][v[j]];

if(k==-1||dis[v[k]]<dis[v[j]])

k=j;

}

vis[v[k]]=true;//标记该点已经加入A集合

if(i==n-1)//最后一次加入的点就要更新答案了

{

res=min(res,dis[v[k]]);

for(int j=0;j<n;j++)//将该点合并到pre上，相应的边权就要合并

{

mpa[v[pre]][v[j]]+=mpa[v[j]][v[k]];

mpa[v[j]][v[pre]]+=mpa[v[j]][v[k]];

}

v[k]=v[--n];//删除点v[k]，把最后一个点扔到v[k]上

}

pre=k;

}

}

return res;

}

}

## 最小割树

#include <set>

#include <cstdio>

#include <cstring>

#include <algorithm>

#define Rep(i, \_begin, \_end) for(int i=(\_begin),i##\_END=(\_end);i<=(i##\_END);i++)

#define For(i, \_begin, \_end) for(int i=(\_begin),i##\_END=(\_end);i!=(i##\_END);i++)

#define Lop(i, \_begin, \_end) for(int i=(\_begin),i##\_END=(\_end);i>=(i##\_END);i--)

#define Dnt(i, \_begin, \_end) for(int i=(\_begin),i##\_END=(\_end);i!=(i##\_END);i--)

using std :: max;

using std :: min;

const int maxx=1000+25;

const int maxm=10000+25;

const int Inf=(-1u)>>1;

int n,m,x,y,z,k,num=1;

int s,t,front,back,ans;

int head[maxx],cur[maxx],quq[maxx],deep[maxx];

int A[maxx],tmp[maxx];

int to[maxm<<1],nxt[maxm<<1],cap[maxm<<1];

bool vis[maxx];

std :: set<int> Cut;

namespace Dinic

{

void Ins(int x,int y,int c)

{

to[++num]=y;nxt[num]=head[x];cap[num]=c;head[x]=num;

to[++num]=x;nxt[num]=head[y];cap[num]=c;head[y]=num;

}

bool Bfs()

{

int x;

Rep(i,1,n)

deep[i]=0;

deep[quq[front=0]=s]=back=1;

while(front!=back)

for(int i=head[x=quq[front ++]];i;i=nxt[i])

if(!deep[to[i]]&&cap[i])

{

deep[quq[back++]=to[i]]=deep[x]+1;

if(to[i]==t)

return true;

}

return deep[t];

}

int Dfs(int x, int v)

{

int f;

if(x==t||!v)

return v;

for(int &i=cur[x];i;i=nxt[i])

if(deep[to[i]]==deep[x]+1&&cap[i])

if(f=Dfs(to[i],min(v,cap[i])))

return cap[i]-=f,cap[i^1]+=f,f;

return 0;

}

int dinic()

{

int flow,ans=0;

while(Bfs())

{

memcpy(cur,head,sizeof(cur));

while(flow=Dfs(s,Inf))

ans+=flow;

}

return ans;

}

}

namespace Gomory\_Hu\_Tree

{

using namespace Dinic;

void init()

{

for(int i=2;i<=num;i+=2)

cap[i]=cap[i^1]=(cap[i]+cap[i^1])>>1;

for(int i=1;i<=n;i++)

vis[i]=0;

}

void Dfs(int x)

{

vis[x] = 1;

for(int i=head[x];i;i=nxt[i])

if(!vis[to[i]] && cap[i])

Dfs(to[i]);

}

void Get(int L, int R)

{

if(L>=R)

return;

init();

s=A[L];t=A[R];

int f=dinic();

Cut.insert(f);

int l=L,r=R;

Dfs(s);

for(int i=L;i<=R;i++)

if(vis[A[i]])

tmp[l++]=A[i];

else

tmp[r--]=A[i];

for(int i=L;i<=R;i++)

A[i]=tmp[i];

Get(L, l-1);Get(r+1, R);

}

}

using namespace Gomory\_Hu\_Tree;

int main()

{

scanf("%d%d", &n, &m);

Rep(i, 1, m) scanf("%d%d%d", &x, &y, &z),Ins(x, y, z);

Rep(i, 1, n) A[i] = i;

Get(1, n);

printf("%d\n",Cut.size());

return 0;

}

## ISAP

const int MAXN=100010;//点数的最大值

const int MAXM=400010;//边数的最大值

const int INF=0x3f3f3f3f;

namespace ISAP

{

struct Edge

{

int to,next,cap,flow;

}edge[MAXM];//注意是MAXM

int tol;

int head[MAXN];

int gap[MAXN],dep[MAXN],cur[MAXN];

void init()

{

tol=0;

memset(head,-1,sizeof(head));

}

void addedge(int u,int v,int w,int rw=0)

{

edge[tol].to=v;edge[tol].cap=w;edge[tol].flow=0;

edge[tol].next=head[u];head[u]=tol++;

edge[tol].to=u;edge[tol].cap=rw;edge[tol].flow=0;

edge[tol].next=head[v];head[v]=tol++;

}

int Q[MAXN];

void BFS(int start,int end)

{

memset(dep,-1,sizeof(dep));

memset(gap,0,sizeof(gap));

gap[0]=1;

int front=0,rear=0;

dep[end]=0;

Q[rear++]=end;

while(front!=rear)

{

int u=Q[front++];

for(int i=head[u];i!=-1;i=edge[i].next)

{

int v=edge[i].to;

if(dep[v]!=-1)continue;

Q[rear++]=v;

dep[v]=dep[u]+1;

gap[dep[v]]++;

}

}

}

int S[MAXN];

int sap(int start,int end,int N)//N总点数

{

BFS(start,end);

memcpy(cur,head,sizeof(head));

int top=0;

int u=start;

int ans=0;

while(dep[start]<N)

{

if(u==end)

{

int Min=INF;

int inser;

for(int i=0;i<top;i++)

if(Min>edge[S[i]].cap-edge[S[i]].flow)

{

Min=edge[S[i]].cap-edge[S[i]].flow;

inser=i;

}

for(int i=0;i<top;i++)

{

edge[S[i]].flow+=Min;

edge[S[i]^1].flow-=Min;

}

ans+=Min;

top=inser;

u=edge[S[top]^1].to;

continue;

}

bool flag=false;

int v;

for(int i=cur[u];i!=-1;i=edge[i].next)

{

v=edge[i].to;

if(edge[i].cap-edge[i].flow&&dep[v]+1==dep[u])

{

flag=true;

cur[u]=i;

break;

}

}

if(flag)

{

S[top++]=cur[u];

u=v;

continue;

}

int Min=N;

for(int i=head[u];i!=-1;i=edge[i].next)

if(edge[i].cap-edge[i].flow&&dep[edge[i].to]<Min)

{

Min=dep[edge[i].to];

cur[u]=i;

}

gap[dep[u]]--;

if(!gap[dep[u]])

return ans;

dep[u]=Min+1;

gap[dep[u]]++;

if(u!=start)

u=edge[S[--top]^1].to;

}

return ans;

}

}

## DINIC

namespace DINIC

{

const int MAXN=500+10;

const int MAXM=4e4+10;

const int INF=0x3f3f3f3f;

struct Edge

{

int u,v,cap,nxt;

Edge(){}

Edge(int \_u,int \_v,int \_cap,int \_nxt):u(\_u),v(\_v),cap(\_cap),nxt(\_nxt){}

}E[MAXM];

int head[MAXN],dis[MAXN],vis[MAXN];

int tol;

void init()

{

tol=0;

memset(head,-1,sizeof(head));

}

void addedge(int u,int v,int cap)

{

E[tol]=Edge(u,v,cap,head[u]);//正向边

head[u]=tol++;

E[tol]=Edge(v,u,0,head[v]);//反向边容量为0

head[v]=tol++;

}

bool BFS(int S,int T)

{

queue<int> q;

q.push(S);

memset(dis,0x3f,sizeof(dis));

dis[S]=0;

while(!q.empty())

{

int x=q.front();

q.pop();

for(int i=head[x];~i;i=E[i].nxt)

{

if(E[i].cap>0&&dis[E[i].v]==INF)

{

dis[E[i].v]=dis[x]+1;

if(E[i].v==T)

return true;

q.push(E[i].v);

}

}

}

return dis[T]<INF; //返回是否能够到达汇点

}

int dfs(int x,int maxflow,int T)

{

if(x==T||maxflow<=0)

return maxflow;

//i=vis[x]当前弧优化

int ret=0;

for(int &i=vis[x];~i;i=E[i].nxt)

{

if(dis[E[i].v]==dis[x]+1&&E[i].cap>0)

{

int flow=dfs(E[i].v,min(maxflow,E[i].cap),T);

if(flow)

{

ret+=flow;

maxflow-=flow;

E[i].cap-=flow;//正向边流量降低

E[i^1].cap+=flow; //反向边流量增加

}

if(maxflow==0)

break;

}

}

return ret;//找不到增广路退出

}

ll dinic(int S,int T,int N)

{

ll ans=0;

while(BFS(S,T))//建立分层图

{

int flow;

for(int i=1;i<=N;i++)//初始化vis

{

vis[i]=head[i];

}

while(flow=dfs(S,INF,T))//一次BFS可以进行多次增广

ans+=(ll)flow;

}

return ans;

}

}

## 费用流

//最小费用最大流，求最大费用只需要取相反数，结果取相反数即可。

//点的总数为N，点的编号0~N-1

const int MAXN=10000;

const int MAXM=100000;

const int INF=0x3f3f3f3f;

namespace MCMF

{

struct Edge

{

int to,next,cap,flow,cost;

}edge[MAXM];

int head[MAXN],tol;

int pre[MAXN],dis[MAXN];

bool vis[MAXN];

int N;//节点总个数，节点编号从0~N-1

void init(int n)

{

N=n;

tol=0;

memset(head,-1,sizeof(head));

}

void addedge(int u,int v,int cap,int cost)

{

edge[tol].to=v;edge[tol].cap=cap;edge[tol].cost=cost;edge[tol].flow=0;edge[tol].next=head[u];

head[u]=tol++;

edge[tol].to=u;edge[tol].cap=0;edge[tol].cost=-cost;edge[tol].flow=0;edge[tol].next=head[v];

head[v]=tol++;

}

bool spfa(int s,int t)

{

queue<int>q;

for(int i=0;i<N;i++)

{

dis[i]=INF;

vis[i]=false;

pre[i]=-1;

}

dis[s]=0;

vis[s]=true;

q.push(s);

while(!q.empty())

{

int u=q.front();

q.pop();

vis[u]=false;

for(int i=head[u];i!=-1;i=edge[i].next)

{

int v=edge[i].to;

if(edge[i].cap>edge[i].flow&&dis[v]>dis[u]+edge[i].cost)

{

dis[v]=dis[u]+edge[i].cost;

pre[v]=i;

if(!vis[v])

{

vis[v]=true;

q.push(v);

}

}

}

}

if(pre[t]==-1)

return false;

else

return true;

}

//返回的是最大流，cost存的是最小费用

int minCostMaxflow(int s,int t,int &cost)

{

int flow=0;

cost=0;

while(spfa(s,t))

{

int Min=INF;

for(int i=pre[t];i!=-1;i=pre[edge[i^1].to])

{

if(Min>edge[i].cap-edge[i].flow)

Min=edge[i].cap-edge[i].flow;

}

for(int i=pre[t];i!=-1;i=pre[edge[i^1].to])

{

edge[i].flow+=Min;

edge[i^1].flow-=Min;

cost+=edge[i].cost\*Min;

}

flow+=Min;

}

return flow;

}

}

## 上下界网络流

**1.无源汇有上下界可行流(也就是循环流)**

模型:一个网络,求出一个流,使得每条边的流量必须>=Li且<=Hi,每个点必须满足总流入量=总流出量(流量守恒)(这个流的特点是循环往复,无始无终).

这个算法是有上下界网络流算法的基础,只要深刻理解这个算法其他算法也就水到渠成,因此我用大篇幅力图将这个算法的思想和细节阐述清楚.

可行流算法的核心是将一个不满足流量守恒的初始流调整成满足流量守恒的流.

流量守恒,即每个点的总流入量=总流出量

如果存在一个可行流,那么一定满足每条边的流量都大于等于流量的下限.因此我们可以令每条边的流量等于流量下限,得到一个初始流,然后建出这个流的残量网络.(即:每条边的流量等于这条边的流量上限与流量下限之差)**这个初始流不一定满足流量守恒,因此最终的可行流一定是在这个初始流的基础上增大了一些边的流量使得所有点满足流量守恒.**

因此我们考虑在残量网络上求出一个另不满足流量守恒的附加流,使得这个附加流和我们的初始流合并之后满足流量守恒.即:

如果某个点在所有边流量等于下界的初始流中满足流量守恒,那么这个点在附加流中也满足流量守恒,

如果某个点在初始流中的流入量比流出量多x,那么这个点在附加流中的流出量比流入量多x.

如果某个点在初始流中的流入量比流出量少x,那么这个点在附加流中的流出量比流入量少x.

         可以认为附加流中一条从u到v的边上的一个流量代表将原图中u到v的流量增大1

X的数值可以枚举x的所有连边求出.比较方便的写法是开一个数组A[],A[i]表示i在初始流中的流入量-流出量的值,那么A[i]的正负表示流入量和流出量的大小关系,下面就用A[i]表示初始流中i的流入量-流出量

但是dinic算法能够求的是满足流量守恒的有源汇最大流,不能在原网络上直接求一个这样的无源汇且不满足流量守恒的附加流.注意到附加流是在原网络上不满足流量守恒的,这启发我们添加一些原网络之外的边和点,用这些边和点实现”原网络上流量不守恒”的限制.

具体地,如果一个点i在原网络上的附加流中需要满足流入量>流出量(初始流中流入量<流出量,A[i]<0),那么我们需要给多的流入量找一个去处,因此我们**建一条从i出发**流量=-A[i]的边.如果A[i]>0,也就是我们需要让附加流中的流出量>流入量,我们需要让多的流出量有一个来路,因此我们**建一条指向i的**流量=A[i]的边.

当然,我们所新建的从i出发的边也要有个去处,指向i的边也要有个来路,因此我们新建一个虚拟源点ss和一个虚拟汇点tt(双写字母是为了和有源汇网络流中的源点s汇点t相区分).新建的指向i的边都从ss出发,从i出发的边都指向tt.一个点要么有一条边指向tt,要么有一条边来自ss,

指向tt的边的总流量上限一定等于ss流出的边的总流量上限,因为每一条边对两个点的A[i]贡献一正一负大小相等,所以全部点的A[i]之和等于0,即小于0的A[i]之和的绝对值=大于0的A[i]之和的绝对值.

如果我们能找到一个流满足新加的边都满流,这个流在原图上的部分就是我们需要的附加流(根据我们的建图方式,“新加的边都满流”和”附加流合并上初始流得到流量平衡的流”是等价的约束条件).

那么怎样找出一个新加的边都满流的流呢?可以发现假如存在这样的方案,这样的流一定是我们所建出的图的ss-tt最大流,所以跑ss到tt的最大流即可.如果最大流的大小等于ss出发的所有边的流量上限之和(此时指向tt的边也一定满流,因为这两部分边的流量上限之和相等).

最后,每条边在可行流中的流量=容量下界+附加流中它的流量(即跑完dinic之后所加反向边的权值).

**2. 有源汇有上下界可行流**

模型:现在的网络有一个**源点s和汇点t**,求出一个流使得源点的总流出量等于汇点的总流入量,其他的点满足流量守恒,而且每条边的流量满足上界和下界限制.

源点和汇点不满足流量守恒,这让我们很难办,因此我们想办法把问题转化成容易处理的每个点都满足流量守恒的无源汇情况.

为了使源汇点满足流量守恒,我们需要有边流入源点s,有边流出汇点t.注意到源点s的流出量等于汇点t的流入量,我们就可以从汇点t向源点s连一条下界为0上界为无穷大的边,相当于把从源点s流出的流量再流回来.在这样的图中套用上面的算法求出一个可行的循环流,拆掉从汇点t到源点s的边就得到一个可行的有源汇流.

这里有一个小问题:最后得到的可行的有源汇流的流量是多少?

可以发现,循环流中一定满足s流出的总流量=流入s的总流量,假定原图中没有边流入s,那么s流出的流量就是t到s的无穷边的流量,也就是s-t可行流的流量.因此我们最后看一下t到s的无穷边的流量(即dinic跑完之后反向边的权值)即可知道原图中有源汇可行流的流量.

代码:这个可行流算法在有源汇有上下界最大流/最小流中都会用到,可以看下面两个算法的代码

**3.有源汇有上下界最大流**

模型:现在的网络有一个源点s和汇点t,求出一个流使得源点的总流出量等于汇点的总流入量,其他的点满足流量守恒,而且每条边的流量满足上界和下界限制.在这些前提下要求**总流量最大**.

首先套用上面的算法求出一个有源汇有上下界可行流.此时的流不一定最大.

接下来在残量网络上跑s-t最大流即可.

最终的最大流流量=可行流流量(即t到s的无穷边上跑出的流量)+新增广出的s-t流量

问题:会不会增广的时候使得一些边不满足流量下限?

不会.因为我们一开始建的图就是把大小等于流量下限的流量拿出去之后的残量网络,这些流量根本没有在图中出现

**4.有源汇有上下界最小流**

模型:现在的网络有一个源点s和汇点t,求出一个流使得源点的总流出量等于汇点的总流入量,其他的点满足流量守恒,而且每条边的流量满足上界和下界限制.在这些前提下要求**总流量最小**.

依然是先跑出一个有源汇可行流.这时候的流也不一定是最小的.假如我们能在残量网络上找到一条s-t的路径使得去掉这条路径上的流量之后仍然满足流量下限,我们就可以得到一个更小的流.好像我们并没有什么算法可以”找到尽可能多的能够去除流量的路径”

这时候需要我们再理解一下dinic的反向边.反向边的流量增加等价于正向边的的流量减少.因此我们在残量网络上找出t到s的流就相当于减小了s到t的流,因此我们在跑出可行流的残量网络上跑t-s最大流,用可行流的大小减去这一次t-s最大流的大小就是最小流的大小.(t-s最大流其实是尽量缩减s-t方向的流).

问题:会不会使流量缩减到不满足流量下限?

不会.和有源汇有上下限的最大流一样,我们之前从每条边上拿出了大小等于流量下限的流量构成初始流,这些流量不在我们建出的图中.最极端的情况是缩减到所有边的流量等于流量下限,不会更小了.

# 字符串

## 后缀数组

/\*后缀数组SA是一个一维数组，它保存1..n的某个排列SA[1]，

SA[2]，……，SA[n]，并且保证Suffix(SA[i])<Suffix(SA[i+1])，1≤i<n。

也就是将S的n个后缀从小到大进行排序之后把排好序的后缀的开头位置顺

次放入SA中。

名次数组Rank[i]保存的是Suffix(i)在所有后缀中从小到大排

列的“名次”。

定义height[i]=suffix(sa[i-1])和suffix(sa[i])的最长公

共前缀，也就是排名相邻的两个后缀的最长公共前缀。

其他数组都是辅助数组\*/

const int maxn=1000001;

int WA[maxn],WB[maxn],WV[maxn],WS[maxn];

int cmp(int \*r,int a,int b,int l)

{

return r[a]==r[b]&&r[a+l]==r[b+l];

}

void da(int \*r,int \*sa,int n,int m)

{

int i,j,p,\*x=WA,\*y=WB,\*t;

for(i=0;i<m;i++) WS[i]=0;

for(i=0;i<n;i++) WS[x[i]=r[i]]++;

for(i=1;i<m;i++) WS[i]+=WS[i-1];

for(i=n-1;i>=0;i--) sa[--WS[x[i]]]=i;

for(j=1,p=1;p<n;j\*=2,m=p)

{

for(p=0,i=n-j;i<n;i++) y[p++]=i;

for(i=0;i<n;i++) if(sa[i]>=j) y[p++]=sa[i]-j;

for(i=0;i<n;i++) WV[i]=x[y[i]];

for(i=0;i<m;i++) WS[i]=0;

for(i=0;i<n;i++) WS[WV[i]]++;

for(i=1;i<m;i++) WS[i]+=WS[i-1];

for(i=n-1;i>=0;i--) sa[--WS[WV[i]]]=y[i];

for(t=x,x=y,y=t,p=1,x[sa[0]]=0,i=1;i<n;i++)

x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;

}

return;

}

int Rank[maxn],height[maxn];

void calheight(int \*r,int \*sa,int n)

{

int i,j,k=0;

for(i=1;i<=n;i++)

Rank[sa[i]]=i;

for(i=0;i<n;height[Rank[i++]]=k)

for(k?k--:0,j=sa[Rank[i]-1];r[i+k]==r[j+k];k++);

return;

}

int r[maxn],sa[maxn];

int RMQ[maxn];

int mm[maxn];

int best[20][maxn];

void initRMQ(int n)

{

int i,j,a,b;

for(mm[0]=-1,i=1;i<=n;i++)

mm[i]=((i&(i-1))==0)?mm[i-1]+1:mm[i-1];

for(i=1;i<=n;i++)

best[0][i]=i;

for(i=1;i<=mm[n];i++)

for(j=1;j<=n+1-(1<<i);j++)

{

a=best[i-1][j];

b=best[i-1][j+(1<<(i-1))];

if(RMQ[a]<RMQ[b])

best[i][j]=a;

else best[i][j]=b;

}

return;

}

int askRMQ(int a,int b)

{

int t;

t=mm[b-a+1];b-=(1<<t)-1;

a=best[t][a];b=best[t][b];

return RMQ[a]<RMQ[b]?a:b;

}

int lcp(int a,int b)

{

int t;

a=Rank[a];b=Rank[b];

if(a>b)

{

t=a;a=b;b=t;

}

return(height[askRMQ(a+1,b)]);

}

## SAM（Reskip）

#define N 250005

#define CNUM 27

#define CBEG 'a'

struct node

{

char data;

int cnt;

//vector<int>debug;

int step;

int pre, next[CNUM];

void clear()

{

step = 0;

cnt = 0;

pre = -1;

memset(next, -1, sizeof(next));

}

int calc();

}nodes[N \* 2];

int node::calc()

{

if (pre == -1)

{

return 0;

}

return step - nodes[pre].step;

}

int beg, root, last;

void init()

{

beg = 1;

root = 0;

last = 0;

nodes[root].clear();

}

void generate(int x)

{

int now = beg;

beg++;

int temp = last;

nodes[now].clear();

nodes[now].step = nodes[temp].step + 1;

nodes[now].data = x;

while (temp != -1 && nodes[temp].next[x] == -1)

{

nodes[temp].next[x] = now;

temp = nodes[temp].pre;

}

if (temp == -1)

{

nodes[now].pre = root;

}

else

{

int st = nodes[temp].next[x];

if (nodes[st].step == nodes[temp].step + 1)

{

nodes[now].pre = st;

}

else

{

int news = beg;

nodes[news].clear();

beg++;

memcpy(nodes[news].next, nodes[st].next, sizeof(nodes[st].next));

nodes[news].data = nodes[st].data;

nodes[news].pre = nodes[st].pre;

nodes[news].step = nodes[temp].step + 1;

nodes[now].pre = nodes[st].pre = news;

while (temp != -1 && nodes[temp].next[x] == st)

{

nodes[temp].next[x] = news;

temp = nodes[temp].pre;

}

}

}

last = now;

return;

}

## 扩展KMP

/\*

\* 扩展KMP算法

\*/

//next[i]:x[i...m-1]与x[0...m-1]的最长公共前缀

//extend[i]:y[i...n-1]与x[0...m-1]的最长公共前缀

void pre\_EKMP(char x[],int m,int next[])

{

next[0]=m;

int j=0;

while(j+1<m&&x[j]==x[j+1])j++;

next[1]=j;

int k=1;

for(int i=2;i<m;i++)

{

int p=next[k]+k-1;

int L=next[i-k];

if(i+L<p+1)next[i]=L;

else

{

j=max(0,p-i+1);

while(i+j<m && x[i+j]==x[j])j++;

next[i]=j;

k=i;

}

}

}

void EKMP(char x[],int m,char y[],int n,int next[],int extend[])

{

pre\_EKMP(x,m,next);

int j=0;

while(j<n&&j<m&&x[j]==y[j])j++;

extend[0]=j;

int k=0;

for(int i=1;i<n;i++)

{

int p=extend[k]+k-1;

int L=next[i-k];

if(i+L<p+1)extend[i]=L;

else

{

j=max(0,p-i+1);

while(i+j<n && j<m && y[i+j]==x[j])j++;

extend[i]=j;

k=i;

}

}

}

## AC自动机

#include<stdio.h>

#include<algorithm>

#include<iostream>

#include<string.h>

#include<queue>

using namespace std;

struct Trie

{

int next[500010][26],fail[500010],end[500010];

int root,L;

int newnode()

{

for(int i=0;i<26;i++)

next[L][i]=-1;

end[L++]=0;

return L-1;

}

void init()

{

L=0;

root=newnode();

}

void insert(char buf[])

{

int len=strlen(buf);

int now=root;

for(int i=0;i<len;i++)

{

if(next[now][buf[i]-'a'] == -1)

next[now][buf[i]-'a'] = newnode();

now=next[now][buf[i]-'a'];

}

end[now]++;

}

void build()

{

queue<int>Q;

fail[root]=root;

for(int i=0;i<26;i++)

if(next[root][i]==-1)

next[root][i]=root;

else

{

fail[next[root][i]]=root;

Q.push(next[root][i]);

}

while(!Q.empty())

{

int now=Q.front();

Q.pop();

for(int i=0;i<26;i++)

if(next[now][i]==-1)

next[now][i]=next[fail[now]][i];

else

{

fail[next[now][i]]=next[fail[now]][i];

Q.push(next[now][i]);

}

}

}

int query(char buf[])

{

int len=strlen(buf);

int now=root;

int res=0;

for(int i=0;i<len;i++)

{

now=next[now][buf[i]-'a'];

int temp=now;

while(temp!=root)

{

res += end[temp];

end[temp]=0;

temp=fail[temp];

}

}

return res;

}

void debug()

{

for(int i=0;i<L;i++)

{

printf("id = %3d,fail = %3d,end = %3d,chi = [",i,fail[i],end[i]);

for(int j=0;j<26;j++)

printf("%2d",next[i][j]);

printf("]\n");

}

}

}ac;

char buf[1000010];

int main()

{

int T;

int n;

scanf("%d",&T);

while(T--)

{

scanf("%d",&n);

ac.init();

for(int i = 0;i < n;i++)

{

scanf("%s",buf);

ac.insert(buf);

}

ac.build();

scanf("%s",buf);

printf("%d\n",ac.query(buf));

}

return 0;

}

## KMP

/\*

\* Next[]的含义：x[i-Next[i]...i-1]=x[0...Next[i]-1]

\* Next[i]为满足x[i-z...i-1]=x[0...z-1]的最大z值（就是x的自身匹配）

\*/

void kmp\_pre(char x[],int m,int Next[])

{

int i,j;

j=Next[0]=-1;

i=0;

while(i<m)

{

while(-1!=j && x[i]!=x[j])j=Next[j];

Next[++i]=++j;

}

}

/\*

\* kmpNext[]的意思：Next'[i]=Next[Next[...[Next[i]]]] (直到Next'[i]<0或者x[Next'[i]]!=x[i])

\* 这样的预处理可以快一些

\*/

void preKMP(char x[],int m,int kmpNext[])

{

int i,j;

j=kmpNext[0]=-1;

i=0;

while(i<m)

{

while(-1!=j && x[i]!=x[j])j=kmpNext[j];

if(x[++i]==x[++j])kmpNext[i]=kmpNext[j];

else kmpNext[i]=j;

}

}

/\*

\* 返回x在y中出现的次数，可以重叠

\*/

int Next[10010];

int KMP\_Count(char x[],int m,char y[],int n)//x是模式串，y是主串

{

int i,j;

int ans=0;

preKMP(x,m,Next);

//kmp\_pre(x,m,Next);

i=j=0;

while(i<n)

{

while(-1!=j&&y[i]!=x[j])j=Next[j];

i++;j++;

if(j>=m)

{

ans++;

j=Next[j];

}

}

return ans;

}

## Manacher

/\*

\* 求最长回文子串

\*/

const int MAXN=110010;

char Ma[MAXN\*2];

int Mp[MAXN\*2];

void Manacher(char s[],int len)

{

int l=0;

Ma[l++]='$';

Ma[l++]='#';

for(int i=0;i<len;i++)

{

Ma[l++]=s[i];

Ma[l++]='#';

}

Ma[l]=0;

int mx=0,id=0;

for(int i=0;i<l;i++)

{

Mp[i]=mx>i?min(Mp[2\*id-i],mx-i):1;

while(Ma[i+Mp[i]]==Ma[i-Mp[i]])Mp[i]++;

if(i+Mp[i]>mx)

{

mx=i+Mp[i];

id=i;

}

}

}

/\*

\* abaaba

\* i: 0 1 2 3 4 5 6 7 8 9 10 11 12 13

\* Ma[i]: $ # a # b # a # a # b # a #

\* Mp[i]: 1 1 2 1 4 1 2 7 2 1 4 1 2 1

\*/

char s[MAXN];

int main()

{

while(scanf("%s",s)==1)

{

int len=strlen(s);

Manacher(s,len);

int ans=0;

for(int i=0;i<2\*len+2;i++)

ans=max(ans,Mp[i]-1);

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

}

return 0;

}

## 大小写转换

string a;

cin>>a;

transform(a.begin(),a.end(),a.begin(),::tolower);

transform(a.begin(),a.end(),a.begin(),::toupper);

# 数据结构

## 树状数组

struct Bit

{

vector<int> a;

int sz;

void init(int n)

{

sz=n;

for(int i=1;i<=n+5;i++)

a.push\_back(0);

}

int lowbit(int x)

{

return x&(-x);

}

int query(int x)

{

int ans = 0;

for(;x;x-=lowbit(x))ans+=a[x];

return ans;

}

void update(int x,int v)

{

for(;x<sz;x+=lowbit(x))

a[x]+=v;

}

}bit;

## 二维树状数组

//v1:(最值)

const int maxn=3e2+10, INF=0x3f3f3f3f;

struct BIT

{

int n;

int bit[maxn][maxn];

int tim[maxn][maxn];

void init(int tn)

{

n=tn;

memset(bit,0x3f,sizeof(bit));

memset(tim,0,sizeof(tim));

}

void update(int nt,int x,int y,int val)

{

for(int i=x;i<=n;i+=i&-i)

for(int j=y;j<=n;j+=j&-j)

{

if(tim[i][j]!=nt)

{

tim[i][j]=nt;

bit[i][j]=INF;

}

bit[i][j]=min(bit[i][j],val);

}

}

int query(int nt,int x,int y)

{

int res=INF;

for(int i=x;i>0;i-=i&-i)

for(int j=y;j>0;j-=j&-j)

{

if(tim[i][j]==nt)

res=min(res,bit[i][j]);

}

return res;

}

};

//

struct BIT

{

ll d[MAXN][MAXN];

int lowbit(int x)

{

return x&(-x);

}

void update(int x,int y,int w)

{

for(int i=x;i<MAXN;i+=lowbit(i))

for(int j=y;j<MAXN;j+=lowbit(j))

d[i][j]+=w;

}

ll get(int x,int y)

{

long long ans=0;

for(int i=x;i;i-=lowbit(i))

for(int j=y;j;j-=lowbit(j))

ans+=d[i][j];

return ans;

}

}B;

## 字典树

typedef long long ll;

struct Tri

{

int ch[maxn][2];

int sz[maxn][2];

int tot;

void init()

{

memset(ch,0,sizeof(ch));

memset(sz,0,sizeof(sz));

tot=2;

}

void insert(int x)

{

int u=1;

for(int i=30;i>=0;i--)

{

int p=(x>>i)&1;

if(!ch[u][p])ch[u][p]=tot++;

sz[u][p]++;

u=ch[u][p];

}

}

void del(int x)

{

int p;

int u=1;

for(int i=30;i>=0;i--)

{

p=(x>>i)&1;

sz[u][p]--;

if(sz[u][p]==0)

ch[u][p]=0;

u=ch[u][p];

}

}

ll get(int x)

{

int u=1;

ll ans=0;

for(int i=30;i>=0;i--)

{

int p=(x>>i)&1;

if(ch[u][p^1])

{

ans^=1<<i;

u=ch[u][p^1];

}

else

{

u=ch[u][p];

}

}

return ans;

}

}T;

## 并查集

int par[MAXN],Rank[MAXN];

void init(int n)

{

for(int i=0;i<n;i++)

{

par[i]=i;

Rank[i]=0;

}

}

int find(int x)

{

if(par[x]==x)

return x;

else

return par[x]=find(par[x]);

}

void unite(int x,int y)

{

x=find(x);

y=find(y);

if(x==y)

return;

if(Rank[x]<Rank[y])

par[x]=y;

else

{

par[y]=x;

if(Rank[x]==Rank[y])

Rank[x]++;

}

}

bool same(int x,int y)

{

return find(x)==find(y);

}

//Q

int fa[MAXN];

void Init(int n)

{

for(int i=1;i<=n;i++)fa[i]=i;

}

int Find(int x)

{

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

}

bool Union(int x,int y)

{

x=Find(x),y=Find(y);

if(x==y)return 0;

return fa[x]=y,1;

}

## 莫队

#include<bits/stdc++.h>

using namespace std;

const int maxn = 1000005;

inline int read()

{

int x=0,f=1;char ch=getchar();

while(ch>'9'||ch<'0'){if(ch=='-')f=-1;ch=getchar();}

while(ch>='0'&&ch<='9'){x=x\*10+ch-'0';ch=getchar();}

return x\*f;

}

int a[maxn],pos[maxn],c[maxn],Ans[maxn];

int ans,n,m;

struct query

{

int l,r,id;

}Q[maxn];

bool cmp(query a,query b)

{

if(pos[a.l]==pos[b.l])

return a.r<b.r;

return pos[a.l]<pos[b.l];

}

void Update(int x)

{

c[x]++;

if(c[x]==2)ans++;

}

void Delete(int x)

{

c[x]--;

if(c[x]==1)ans--;

}

int main()

{

n=read(),m=read(),m=read();

int sz =ceil(sqrt(1.0\*n));

for(int i=1;i<=n;i++)

{

a[i]=read();

pos[i]=(i-1)/sz;

}

for(int i=1;i<=m;i++)

{

Q[i].l=read();

Q[i].r=read();

Q[i].id = i;

}

sort(Q+1,Q+1+m,cmp);

int L=1,R=0;ans=0;

for(int i=1;i<=m;i++)

{

int id = Q[i].id;

while(R<Q[i].r)R++,Update(a[R]);

while(L>Q[i].l)L--,Update(a[L]);

while(R>Q[i].r)Delete(a[R]),R--;

while(L<Q[i].l)Delete(a[L]),L++;

Ans[id]=ans;

}

for(int i=1;i<=m;i++)

printf("%d\n",Ans[i]);

}

## DLX（精确覆盖）

const int maxnode = 100010;

const int MaxM = 1010;

const int MaxN = 1010;

struct DLX

{

int n,m,size;

int U[maxnode],D[maxnode],R[maxnode],L[maxnode],Row[maxnode],Col[maxnode];

int H[MaxN], S[MaxM];

int ansd, ans[MaxN];

void init(int \_n,int \_m)

{

n = \_n;

m = \_m;

for(int i = 0;i <= m;i++)

{

S[i] = 0;

U[i] = D[i] = i;

L[i] = i-1;

R[i] = i+1;

}

R[m] = 0; L[0] = m;

size = m;

for(int i = 1;i <= n;i++)

H[i] = -1;

}

void Link(int r,int c)

{

++S[Col[++size]=c];

Row[size] = r;

D[size] = D[c];

U[D[c]] = size;

U[size] = c;

D[c] = size;

if(H[r] < 0)H[r] = L[size] = R[size] = size;

else

{

R[size] = R[H[r]];

L[R[H[r]]] = size;

L[size] = H[r];

R[H[r]] = size;

}

}

void remove(int c)

{

L[R[c]] = L[c]; R[L[c]] = R[c];

for(int i = D[c];i != c;i = D[i])

for(int j = R[i];j != i;j = R[j])

{

U[D[j]] = U[j];

D[U[j]] = D[j];

--S[Col[j]];

}

}

void resume(int c)

{

for(int i = U[c];i != c;i = U[i])

for(int j = L[i];j != i;j = L[j])

++S[Col[U[D[j]]=D[U[j]]=j]];

L[R[c]] = R[L[c]] = c;

}

//d为递归深度

bool Dance(int d)

{

if(R[0] == 0)

{

ansd = d;

return true;

}

int c = R[0];

for(int i = R[0];i != 0;i = R[i])

if(S[i] < S[c])

c = i;

remove(c);

for(int i = D[c];i != c;i = D[i])

{

ans[d] = Row[i];

for(int j = R[i]; j != i;j = R[j])remove(Col[j]);

if(Dance(d+1))return true;

for(int j = L[i]; j != i;j = L[j])resume(Col[j]);

}

resume(c);

return false;

}

};

## DLX（重复覆盖）

const int maxnode,MaxN,MaxM;

struct DLX

{

int n,m,size;

int U[maxnode],D[maxnode],R[maxnode],L[maxnode],Row[maxnode],Col[maxnode];

int H[MaxN],S[MaxM];

int ansd;

void init(int \_n,int \_m)

{

n = \_n;

m = \_m;

for(int i = 0;i <= m;i++)

{

S[i] = 0;

U[i] = D[i] = i;

L[i] = i-1;

R[i] = i+1;

}

R[m] = 0; L[0] = m;

size = m;

for(int i = 1;i <= n;i++)H[i] = -1;

}

void Link(int r,int c)//连接（第r行可以覆盖第c列）

{

++S[Col[++size]=c];

Row[size] = r;

D[size] = D[c];

U[D[c]] = size;

U[size] = c;

D[c] = size;

if(H[r] < 0)H[r] = L[size] = R[size] = size;

else

{

R[size] = R[H[r]];

L[R[H[r]]] = size;

L[size] = H[r];

R[H[r]] = size;

}

}

void remove(int c)//删除

{

for(int i = D[c];i != c;i = D[i])

L[R[i]] = L[i], R[L[i]] = R[i];

}

void resume(int c)//恢复

{

for(int i = U[c];i != c;i = U[i])

L[R[i]] = R[L[i]] = i;

}

bool v[MaxM];

int f()//估价函数

{

int ret = 0;

for(int c = R[0]; c != 0;c = R[c])v[c] = true;

for(int c = R[0]; c != 0;c = R[c])

if(v[c])

{

ret++;

v[c] = false;

for(int i = D[c];i != c;i = D[i])

for(int j = R[i];j != i;j = R[j])

v[Col[j]] = false;

}

return ret;

}

void Dance(int d)//递归深度

{

if(d + f() >= ansd)return;

if(R[0] == 0)

{

if(d < ansd)ansd = d;

return;

}

int c = R[0];

for(int i = R[0];i != 0;i = R[i])

if(S[i] < S[c])

c = i;

for(int i = D[c];i != c;i = D[i])

{

remove(i);

for(int j = R[i];j != i;j = R[j])remove(j);

Dance(d+1);

for(int j = L[i];j != i;j = L[j])resume(j);

resume(i);

}

}

};

## ST表（RMQ）

/\*

状态转移方程的含义是：先更新所有长度为F[i,0]即1个元素，然后通过2个1个元素的最值，

获得所有长度为F[i,1]即2个元素的最值，然后再通过2个2个元素的最值，获得所有长度为

F[i,2]即4个元素的最值，以此类推更新所有长度的最值。

然后是查询。

假如我们需要查询的区间为(i,j)，那么我们需要找到覆盖这个闭区间(左边界取i，右边界取j)的最小幂（可以重复，比如查询5，6，7，8，9，我们可以查询5678和6789）。

因为这个区间的长度为j - i + 1,所以我们可以取k=log2( j - i + 1)，则有：RMQ(A, i, j)=max{F[i , k], F[ j - 2 ^ k + 1, k]}。

举例说明，要求区间[2，8]的最大值，k = log2（8 - 2 + 1）= 2，即求max(F[2, 2]，F[8 - 2 ^ 2 + 1, 2]) = max(F[2, 2]，F[5, 2])；

\*/

struct RMQ

{

const static int RMQ\_size = maxn;

int n;

int ArrayMax[RMQ\_size][21];

int ArrayMin[RMQ\_size][21];

void build\_rmq()

{

for(int j = 1 ; (1<<j) <= n ; ++ j)

for(int i = 0 ; i + (1<<j) - 1 < n ; ++ i)

{

ArrayMax[i][j]=max(ArrayMax[i][j-1],ArrayMax[i+(1<<(j-1))][j-1]);

ArrayMin[i][j]=min(ArrayMin[i][j-1],ArrayMin[i+(1<<(j-1))][j-1]);

}

}

int QueryMax(int L,int R)

{

int k = 0;

while( (1<<(k+1)) <= R-L+1) k ++ ;

return max(ArrayMax[L][k],ArrayMax[R-(1<<k)+1][k]);

}

int QueryMin(int L,int R)

{

int k = 0;

while( (1<<(k+1)) <= R-L+1) k ++ ;

return min(ArrayMin[L][k],ArrayMin[R-(1<<k)+1][k]);

}

void init(int \* a,int sz)

{

n = sz ;

for(int i = 0 ; i < n ; ++ i) ArrayMax[i][0] = ArrayMin[i][0] = a[i];

build\_rmq();

}

}s1,s2;

## 线段树区间合并

#include<cstdio>

#include<algorithm>

#define lson l,m,rt<<1

#define rson m+1,r,rt<<1|1

using namespace std;

const int maxn=50000;

int n,mNum,op,a,b;

struct tree

{

int sum,lsum,rsum,cover;

}Tree[maxn<<2];

void BuildTree(int l,int r,int rt)

{

Tree[rt].cover=-1;

Tree[rt].lsum=Tree[rt].rsum=Tree[rt].sum=r-l+1;

if(l==r)

return;

int m=(l+r)>>1;

BuildTree(lson);

BuildTree(rson);

}/\* BuildTree \*/

void PushDown(int rt,int k)

{

if (Tree[rt].cover!=-1)

{ /\* Lazy Tag \*/

Tree[rt<<1].cover=Tree[rt<<1|1].cover=Tree[rt].cover;

Tree[rt<<1].lsum=Tree[rt<<1].rsum=Tree[rt<<1].sum=Tree[rt].cover?0:(k-(k>>1));

Tree[rt<<1|1].lsum=Tree[rt<<1|1].rsum=Tree[rt<<1|1].sum=Tree[rt].cover?0:(k>>1);

Tree[rt].cover=-1;

}

}/\* PushDown \*/

int query(int w,int l,int r,int rt)//查询最小的满足条件的点

{

if(l==r)

return 1;

PushDown(rt,r-l+1); // Push Down

int m=(l+r)>>1;

if(Tree[rt<<1].sum>=w) // 左连续区间长度

return query(w,lson);

else if(Tree[rt<<1].rsum+Tree[rt<<1|1].lsum>=w) // 左区间后半部分与右区间左半部分长度

return m-Tree[rt<<1].rsum+1;

else // 右连续区间长度

return query(w,rson);

}/\* query \*/

/\*int query(int w,int l,int r,int rt)//查询包含w的最长的区间长度

{

if(l==r||Tree[rt].sum==r-l+1||Tree[rt].sum==0)

return Tree[rt].sum;

PushDown(rt,r-l+1); // Push Down

int m=(l+r)>>1;

if(w<=m)

{

if(w>=m-Tree[rt<<1].rsum+1)//因为w<=m，看左子树，m-Tree[rt<<1].rsum+1代表左子树右边连续区间的左边界值，如果w在左子树的右区间内，则要看右子树的左区间有多长并返回

return query(w,lson)+query(m+1,rson);

else

return query(w,lson);

}

else

{

if(w<=m+Tree[rt<<1|1].lsum)//同理

return query(m,lson)+query(w,rson);

else

return query(w,rson);

}

}\*/

void PushUp(int rt,int k)

{

Tree[rt].lsum=Tree[rt<<1].lsum; /\* 左区间的左半部分 \*/

Tree[rt].rsum=Tree[rt<<1|1].rsum; /\* 右区间的右半部分 \*/

if (Tree[rt].lsum==k-(k>>1))

Tree[rt].lsum+=Tree[rt<<1|1].lsum;

if (Tree[rt].rsum==k>>1)

Tree[rt].rsum+=Tree[rt<<1].rsum;

Tree[rt].sum=max(Tree[rt<<1].rsum+Tree[rt<<1|1].lsum,max(Tree[rt<<1].sum,Tree[rt<<1|1].sum));

}/\* PushUp \*/

void UpData(int L,int R,int c,int l,int r,int rt)

{

if(L<=l&&r<=R)

{

Tree[rt].lsum=Tree[rt].rsum=Tree[rt].sum=c?0:r-l+1;

Tree[rt].cover=c;

return ;

}/\* End of If \*/

PushDown(rt,r-l+1);/\* Push Down \*/

int m=(l+r)>>1;

if(L<=m)

UpData(L,R,c,lson);

if(R>m)

UpData(L,R,c,rson);

PushUp(rt,r-l+1);/\* Push Up \*/

}/\* Updata \*/

int main()

{

scanf("%d %d",&n,&mNum);

BuildTree(1,n,1); //BuildTree

for (int i=1;i<=mNum;++i)

{

scanf("%d",&op);

if (op==1)

{

scanf("%d",&a);

if (Tree[1].sum<a)

printf("0\n");

else

{

int pos=query(a,1,n,1);

printf("%d\n",pos);

UpData(pos,pos+a-1,1,1,n,1);

}

}

else

{

scanf("%d %d",&a,&b);

UpData(a,a+b-1,0,1,n,1);

}

}

return 0;

}

## 矩形面积并

#include<cstdio>

#include<cstring>

#include<algorithm>

#include<cmath>

#include<iostream>

using namespace std;

const int maxn=100+5;

const double esp=1e-8;

int lenx=0,ls=0,cover[maxn<<3]={0},Case=1;

double sum[maxn<<3]={0},x[maxn<<1]={0},ans=0;

struct sege

{

int flag;

double x1,x2,h;

sege (){}

sege(double a,double b,double c,int d) :x1(a),x2(b),h(c),flag(d) {}

friend bool operator < (sege a,sege b)

{

return a.h<b.h;

}

}seg[maxn<<1];

void init()

{

lenx=0,ls=0,ans=0;

memset(cover,0,sizeof(cover));

memset(sum,0,sizeof(sum));

}

void work(int l,int r,int root)

{

if(cover[root]>0) sum[root]=x[r]-x[l];

else if(l==r-1) sum[root]=0;

else sum[root]=sum[root<<1]+sum[root<<1|1];

}

void merge()

{

int len=lenx,i;

lenx=1;

for(i=1;i<len;i++)

{

if(x[i]-x[i-1]>=esp)

{

x[lenx++]=x[i];

}

}

}

void query(int L,int R,int l,int r,int root,int flag)

{

int m=(l+r)>>1;

if(L<=l&&r<=R)

{

cover[root]+=flag;

work(l,r,root);

return ;

}

if(l==r-1) return ;

if(m>=L) query(L,R,l,m,root<<1,flag);

if(m<R) query(L,R,m,r,root<<1|1,flag);

work(l,r,root);

}

int main()

{

int n,i;

double x1,x2,y1,y2;

while(~scanf("%d",&n)&&n)

{

init();

for(i=0;i<n;i++)

{

scanf("%lf%lf%lf%lf",&x1,&y1,&x2,&y2);

if(x1>x2||y1>y2)

{

swap(x1,x2);

swap(y1,y2);

}

seg[ls++]=sege(x1,x2,y1,1);

seg[ls++]=sege(x1,x2,y2,-1);

x[lenx++]=x1;

x[lenx++]=x2;

}

sort(x,x+lenx);

merge();

sort(seg,seg+ls);

for(i=0;i<ls-1;i++)

{

x1=lower\_bound(x,x+lenx,seg[i].x1)-x;

x2=lower\_bound(x,x+lenx,seg[i].x2)-x;

query(x1,x2,0,lenx-1,1,seg[i].flag);

ans+=(seg[i+1].h-seg[i].h)\*sum[1];

}

printf("Test case #%d\nTotal explored area: %.2lf\n\n",Case++,ans);

}

return 0;

}

## 矩形面积交

#include<cstdio>

#include<cstring>

#include<algorithm>

#include<cmath>

#include<iostream>

#define xx first

#define yy second

using namespace std;

const int maxn=1000+5;

const double esp=1e-8;

int lenx=0,ls=0,cover[maxn<<3]={0};

double x[maxn<<1]={0},ans=0;

pair<double,double> sum[maxn<<3];

struct sege

{

int flag;

double x1,x2,h;

sege (){}

sege(double a,double b,double c,int d) :x1(a),x2(b),h(c),flag(d) {}

friend bool operator < (sege a,sege b)

{

return a.h<b.h;

}

}seg[maxn<<1];

void init()

{

lenx=0,ls=0,ans=0;

memset(cover,0,sizeof(cover));

memset(sum,0,sizeof(sum));

}

void work(int l,int r,int root)

{

if(cover[root]>0)

{

sum[root].xx=x[r]-x[l];

if(cover[root]>1)

{

sum[root].yy=sum[root].xx;

}

else

{

if(l==r-1) sum[root].yy=0;

else sum[root].yy=sum[root<<1].xx+sum[root<<1|1].xx;

}

}

else if(l==r-1) sum[root].xx=sum[root].yy=0;

else sum[root].xx=sum[root<<1].xx+sum[root<<1|1].xx,sum[root].yy=sum[root<<1].yy+sum[root<<1|1].yy;

}

void merge()

{

int len=lenx,i;

lenx=1;

for(i=1;i<len;i++)

{

if(x[i]-x[i-1]>=esp)

{

x[lenx++]=x[i];

}

}

}

void query(int L,int R,int l,int r,int root,int flag)

{

int m=(l+r)>>1;

if(L<=l&&r<=R)

{

cover[root]+=flag;

work(l,r,root);

return ;

}

if(l==r-1) return ;

if(m>=L) query(L,R,l,m,root<<1,flag);

if(m<R) query(L,R,m,r,root<<1|1,flag);

work(l,r,root);

}

int main()

{

int n,i,T;

double x1,x2,y1,y2;

scanf("%d",&T);

while(T--)

{

init();

scanf("%d",&n);

for(i=0;i<n;i++)

{

scanf("%lf%lf%lf%lf",&x1,&y1,&x2,&y2);

if(x1>x2||y1>y2)

{

swap(x1,x2);

swap(y1,y2);

}

seg[ls++]=sege(x1,x2,y1,1);

seg[ls++]=sege(x1,x2,y2,-1);

x[lenx++]=x1;

x[lenx++]=x2;

}

sort(x,x+lenx);

merge();

sort(seg,seg+ls);

for(i=0;i<ls-1;i++)

{

x1=lower\_bound(x,x+lenx,seg[i].x1)-x;

x2=lower\_bound(x,x+lenx,seg[i].x2)-x;

query(x1,x2,0,lenx-1,1,seg[i].flag);

ans+=(seg[i+1].h-seg[i].h)\*sum[1].yy;

}

printf("%.2lf\n",ans);

}

return 0;

}

## 矩形轮廓

#include<cstdio>

#include<cstring>

#include<algorithm>

#include<cmath>

#include<iostream>

#define xx first

#define yy second

using namespace std;

const int maxn=5000+5;

int lenx=0,ls=0,cover[maxn<<3]={0};

int x[maxn<<1]={0},ans=0,last;

pair<int,int> sum[maxn<<3];

bool lcover[maxn<<3],rcover[maxn<<3];

struct sege

{

int x1,x2,h,flag;

sege (){}

sege(int a,int b,int c,int d) :x1(a),x2(b),h(c),flag(d) {}

friend bool operator < (sege a,sege b)

{

return a.h<b.h;

}

}seg[maxn<<1];

void init()

{

lenx=0,ls=0,ans=0,last=0;

memset(cover,0,sizeof(cover));

memset(sum,0,sizeof(sum));

memset(lcover,false,sizeof(lcover));

memset(rcover,false,sizeof(rcover));

}

void work(int l,int r,int root)

{

if(cover[root]>0)

{

sum[root].xx=x[r]-x[l];

sum[root].yy=1;

lcover[root]=rcover[root]=true;

}

else if(l==r-1)

{

sum[root].xx=sum[root].yy=0;

lcover[root]=rcover[root]=false;

}

else

{

sum[root].xx=sum[root<<1].xx+sum[root<<1|1].xx;

sum[root].yy=sum[root<<1].yy+sum[root<<1|1].yy;

lcover[root]=lcover[root<<1];

rcover[root]=rcover[root<<1|1];

if(rcover[root<<1]&&lcover[root<<1|1])

{

sum[root].yy--;

}

}

}

void merge()

{

int len=lenx,i;

lenx=1;

for(i=1;i<len;i++)

{

if(x[i]!=x[i-1])

{

x[lenx++]=x[i];

}

}

}

void query(int L,int R,int l,int r,int root,int flag)

{

int m=(l+r)>>1;

if(L<=l&&r<=R)

{

cover[root]+=flag;

work(l,r,root);

return ;

}

if(l==r-1) return ;

if(m>=L) query(L,R,l,m,root<<1,flag);

if(m<R) query(L,R,m,r,root<<1|1,flag);

work(l,r,root);

}

int main()

{

int x1,x2,y1,y2,n,i;

while(~scanf("%d",&n)&&n)

{

init();

for(i=0;i<n;i++)

{

scanf("%d%d%d%d",&x1,&y1,&x2,&y2);

if(x1>x2||y1>y2)

{

swap(x1,x2);

swap(y1,y2);

}

seg[ls++]=sege(x1,x2,y1,1);

seg[ls++]=sege(x1,x2,y2,-1);

x[lenx++]=x1;

x[lenx++]=x2;

}

sort(x,x+lenx);

merge();

sort(seg,seg+ls);

for(i=0;i<ls-1;i++)

{

x1=lower\_bound(x,x+lenx,seg[i].x1)-x;

x2=lower\_bound(x,x+lenx,seg[i].x2)-x;

query(x1,x2,0,lenx-1,1,seg[i].flag);

ans+=(sum[1].yy\*2\*(seg[i+1].h-seg[i].h)+abs(sum[1].xx-last));

last=sum[1].xx;

}

x1=lower\_bound(x,x+lenx,seg[ls-1].x1)-x;

x2=lower\_bound(x,x+lenx,seg[ls-1].x2)-x;

query(x1,x2,0,lenx-1,1,seg[ls-1].flag);

ans+=abs(sum[1].xx-last);

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

}

return 0;

}

## 树链剖分（点权）

const int MAXN=50010;

struct Edge

{

int to,next;

}edge[MAXN\*2];

int head[MAXN],tot;

int top[MAXN];//top[v] 表示v所在的重链的顶端节点

int fa[MAXN];//父亲节点

int deep[MAXN];//深度

int num[MAXN];//num[v] 表示以v为根的子树的节点数

int p[MAXN];//p[v]表示v对应的位置

int fp[MAXN];//fp和p数组相反

int son[MAXN];//重儿子

int pos;

void init()

{

tot=0;

memset(head,-1,sizeof(head));

pos=1;//使用树状数组，编号从头1开始

memset(son,-1,sizeof(son));

}

void addedge(int u,int v)

{

edge[tot].to=v;edge[tot].next=head[u];head[u]=tot++;

}

void dfs1(int u,int pre,int d)

{

deep[u]=d;

fa[u]=pre;

num[u]=1;

for(int i=head[u];i!=-1;i=edge[i].next)

{

int v=edge[i].to;

if(v!=pre)

{

dfs1(v,u,d+1);

num[u]+=num[v];

if(son[u]==-1||num[v]>num[son[u]])

son[u]=v;

}

}

}

void getpos(int u,int sp)

{

top[u]=sp;

p[u]=pos++;

fp[p[u]]=u;

if(son[u]==-1) return;

getpos(son[u],sp);

for(int i=head[u];i!=-1;i=edge[i].next)

{

int v=edge[i].to;

if(v!=son[u]&&v!=fa[u])

getpos(v,v);

}

}

//树状数组

int lowbit(int x)

{

return x&(-x);

}

int c[MAXN];

int n;

int sum(int i)

{

int s=0;

while(i>0)

{

s+=c[i];

i-=lowbit(i);

}

return s;

}

void add(int i,int val)

{

while(i<=n)

{

c[i]+=val;

i+=lowbit(i);

}

}

void Change(int u,int v,int val)//u->v的路径上点的值改变val

{

int f1=top[u],f2=top[v];

int tmp=0;

while(f1!=f2)

{

if(deep[f1]<deep[f2])

{

swap(f1,f2);

swap(u,v);

}

add(p[f1],val);

add(p[u]+1,-val);

u=fa[f1];

f1=top[u];

}

if(deep[u]>deep[v])

swap(u,v);

add(p[u],val);

add(p[v]+1,-val);

}

int a[MAXN];

int main()

{

int M,P;

while(scanf("%d%d%d",&n,&M,&P)==3)

{

int u,v;

int C1,C2,K;

char op[10];

init();

for(int i=1;i<=n;i++)

{

scanf("%d",&a[i]);

}

while(M--)

{

scanf("%d%d",&u,&v);

addedge(u,v);

addedge(v,u);

}

dfs1(1,0,0);

getpos(1,1);

memset(c,0,sizeof(c));

for(int i=1;i<=n;i++)

{

add(p[i],a[i]);

add(p[i]+1,-a[i]);

}

while(P--)

{

scanf("%s",op);

if(op[0]=='Q')

{

scanf("%d",&u);

printf("%d\n",sum(p[u]));

}

else

{

scanf("%d%d%d",&C1,&C2,&K);

if(op[0]=='D')

K=-K;

Change(C1,C2,K);

}

}

}

return 0;

}

## 树链剖分（边权）

//基于边权，修改单条边权，查询路径边权最大值（SPOJ QTREE 树链剖分+线段树 ）

const int MAXN=10010;

struct Edge

{

int to,next;

}edge[MAXN\*2];

int head[MAXN],tot;

int top[MAXN];//top[v]表示v所在的重链的顶端节点

int fa[MAXN]; //父亲节点

int deep[MAXN];//深度

int num[MAXN];//num[v]表示以v为根的子树的节点数

int p[MAXN];//p[v]表示v与其父亲节点的连边在线段树中的位置

int fp[MAXN];//和p数组相反

int son[MAXN];//重儿子

int pos;

void init()

{

tot=0;

memset(head,-1,sizeof(head));

pos=0;

memset(son,-1,sizeof(son));

}

void addedge(int u,int v)

{

edge[tot].to=v;edge[tot].next=head[u];head[u]=tot++;

}

void dfs1(int u,int pre,int d) //第一遍dfs求出fa,deep,num,son

{

deep[u]=d;

fa[u]=pre;

num[u]=1;

for(int i=head[u];i!=-1;i=edge[i].next)

{

int v=edge[i].to;

if(v!=pre)

{

dfs1(v,u,d+1);

num[u]+=num[v];

if(son[u]==-1||num[v]>num[son[u]])

son[u]=v;

}

}

}

void getpos(int u,int sp) //第二遍dfs求出top和p

{

top[u]=sp;

p[u]=pos++;

fp[p[u]]=u;

if(son[u]==-1)

return;

getpos(son[u],sp);

for(int i=head[u];i!=-1;i=edge[i].next)

{

int v=edge[i].to;

if(v!=son[u]&&v!=fa[u])

getpos(v,v);

}

}

//线段树

struct Node

{

int l,r;

int Max;

}segTree[MAXN\*3];

void build(int i,int l,int r)

{

segTree[i].l=l;

segTree[i].r=r;

segTree[i].Max=0;

if(l==r)

return;

int mid=(l+r)/2;

build(i<<1,l,mid);

build((i<<1)|1,mid+1,r);

}

void push\_up(int i)

{

segTree[i].Max=max(segTree[i<<1].Max,segTree[(i<<1)|1].Max);

}

void update(int i,int k,int val) // 更新线段树的第k个值为val

{

if(segTree[i].l==k&&segTree[i].r==k)

{

segTree[i].Max=val;

return;

}

int mid=(segTree[i].l+segTree[i].r)/2;

if(k<=mid)update(i<<1,k,val);

else update((i<<1)|1,k,val);

push\_up(i);

}

int query(int i,int l,int r) //查询线段树中[l,r] 的最大值

{

if(segTree[i].l==l&&segTree[i].r==r)

return segTree[i].Max;

int mid=(segTree[i].l+segTree[i].r)/2;

if(r<=mid)return query(i<<1,l,r);

else if(l>mid)return query((i<<1)|1,l,r);

else return max(query(i<<1,l,mid),query((i<<1)|1,mid+1,r));

}

int find(int u,int v)//查询u->v边的最大值

{

int f1=top[u],f2=top[v];

int tmp=0;

while(f1!=f2)

{

if(deep[f1]<deep[f2])

{

swap(f1,f2);

swap(u,v);

}

tmp=max(tmp,query(1,p[f1],p[u]));

u=fa[f1];f1=top[u];

}

if(u==v)

return tmp;

if(deep[u]>deep[v])

swap(u,v);

return max(tmp,query(1,p[son[u]],p[v]));

}

int e[MAXN][3];

int main()

{

//freopen("in.txt","r",stdin);

//freopen("out.txt","w",stdout);

int T;

int n;

scanf("%d",&T);

while(T--)

{

init();

scanf("%d",&n);

for(int i=0;i<n-1;i++)

{

scanf("%d%d%d",&e[i][0],&e[i][1],&e[i][2]);

addedge(e[i][0],e[i][1]);

addedge(e[i][1],e[i][0]);

}

dfs1(1,0,0);

getpos(1,1);

build(1,0,pos-1);

for(int i=0;i<n-1;i++)

{

if(deep[e[i][0]]>deep[e[i][1]])

swap(e[i][0],e[i][1]);

update(1,p[e[i][1]],e[i][2]);

}

char op[10];

int u,v;

while(scanf("%s",op)==1)

{

if(op[0]=='D')break;

scanf("%d%d",&u,&v);

if(op[0]=='Q')

printf("%d\n",find(u,v));//查询u->v路径上边权的最大值

else update(1,p[e[u-1][1]],v);//修改第u条边的长度为v

}

}

return 0;

}

## 主席树

/\*

\* 给出一个序列，查询区间内有多少个不相同的数

\*/

const int MAXN=30010;

const int M=MAXN\*100;

int n,q,tot;

int a[MAXN];

int T[MAXN],lson[M],rson[M],c[M];

int build(int l,int r)

{

int root=tot++;

c[root]=0;

if(l!=r)

{

int mid=(l+r)>>1;

lson[root]=build(l,mid);

rson[root]=build(mid+1,r);

}

return root;

}

int update(int root,int pos,int val)

{

int newroot=tot++,tmp=newroot;

c[newroot]=c[root]+val;

int l=1,r=n;

while(l<r)

{

int mid=(l+r)>>1;

if(pos<=mid)

{

lson[newroot]=tot++;rson[newroot]=rson[root];

newroot=lson[newroot];root=lson[root];

r=mid;

}

else

{

rson[newroot]=tot++;lson[newroot]=lson[root];

newroot=rson[newroot];root=rson[root];

l=mid+1;

}

c[newroot]=c[root]+val;

}

return tmp;

}

int query(int root,int pos)

{

int ret=0;

int l=1,r=n;

while(pos<r)

{

int mid=(l+r)>>1;

if(pos<=mid)

{

r=mid;

root=lson[root];

}

else

{

ret+=c[lson[root]];

root=rson[root];

l=mid+1;

}

}

return ret+c[root];

}

int main()

{

//freopen("in.txt","r",stdin);

//freopen("out.txt","w",stdout);

while(scanf("%d",&n) == 1)

{

tot=0;

for(int i=1;i<=n;i++)

scanf("%d",&a[i]);

T[n+1]=build(1,n);

map<int,int>mp;

for(int i=n;i>=1;i--)

{

if(mp.find(a[i])==mp.end())

{

T[i]=update(T[i+1],i,1);

}

else

{

int tmp=update(T[i+1],mp[a[i]],-1);

T[i]=update(tmp,i,1);

}

mp[a[i]]=i;

}

scanf("%d",&q);

while(q--)

{

int l,r;

scanf("%d%d",&l,&r);

printf("%d\n",query(T[l],r));

}

}

return 0;

}

## 点分治

#include<bits/stdc++.h>

using namespace std;

const int MAXN=10010;

int N,K;

int ans,root,Max;

struct node

{

int v,next,w;

}edge[MAXN\*2];

int head[MAXN],tot;

int size[MAXN];//树的大小

int maxv[MAXN];//最大孩子节点的size

int vis[MAXN];

int dis[MAXN];

int num;

void init()

{

tot=0;

ans=0;

memset(head,-1,sizeof(head));

memset(vis,0,sizeof(vis));

}

void add\_edge(int u,int v,int w)

{

edge[tot].v=v;

edge[tot].w=w;

edge[tot].next=head[u];

head[u]=tot++;

}

//处理子树的大小

void dfssize(int u,int f)

{

size[u]=1;

maxv[u]=0;

for(int i=head[u];i!=-1;i=edge[i].next)

{

int v=edge[i].v;

if(v==f||vis[v])continue;

dfssize(v,u);

size[u]+=size[v];

if(size[v]>maxv[u])maxv[u]=size[v];

}

}

//找重心

void dfsroot(int r,int u,int f)

{

if(size[r]-size[u]>maxv[u])//size[r]-size[u]是u上面部分的树的尺寸，跟u的最大孩子比，找到最大孩子的最小差值节点

maxv[u]=size[r]-size[u];

if(maxv[u]<Max)Max=maxv[u],root=u;

for(int i=head[u];i!=-1;i=edge[i].next)

{

int v=edge[i].v;

if(v==f||vis[v])continue;

dfsroot(r,v,u);

}

}

//求每个点离重心的距离

void dfsdis(int u,int d,int f)

{

dis[num++]=d;

for(int i=head[u];i!=-1;i=edge[i].next)

{

int v=edge[i].v;

if(v!=f&&!vis[v])

dfsdis(v,d+edge[i].w,u);

}

}

//计算以u为根的子树中有多少点对的距离小于等于K

int calc(int u,int d)

{

int ret=0;

num=0;

dfsdis(u,d,0);

sort(dis,dis+num);

int i=0,j=num-1;

while(i<j)

{

while(dis[i]+dis[j]>K&&i<j)j--;

ret+=j-i;

i++;

}

return ret;

}

void dfs(int u)

{

Max=N;

dfssize(u,0);

dfsroot(u,u,0);

ans+=calc(root,0);

vis[root]=1;

for(int i=head[root];i!=-1;i=edge[i].next)

{

int v=edge[i].v;

if(!vis[v])

{

ans-=calc(v,edge[i].w);

dfs(v);

}

}

}

int main()

{

while(scanf("%d%d",&N,&K)!=EOF)

{

if(!N&&!K)break;

int u,v,w;

init();

for(int i=1;i<N;i++)

{

scanf("%d%d%d",&u,&v,&w);

add\_edge(u,v,w);

add\_edge(v,u,w);

}

dfs(1);

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

}

return 0;

}

## LCT

//动态维护一组森林，要求支持一下操作:

//link(a,b) : 如果a,b不在同一颗子树中，则通过在a,b之间连边的方式，连接这两颗子树

//cut(a,b) : 如果a,b在同一颗子树中，且a!=b,则将a视为这颗子树的根以后，切断b与其父亲结点的连接

//ADD(a,b,w): 如果a,b在同一颗子树中，则将a,b之间路径上所有点的点权增加w

//query(a,b): 如果a,b在同一颗子树中，返回a,b之间路径上点权的最大值

const int MAXN = 300010;

int ch[MAXN][2],pre[MAXN],key[MAXN];

int add[MAXN],rev[MAXN],Max[MAXN];

bool rt[MAXN];

void Update\_Add(int r,int d)

{

if(!r)

return;

key[r]+=d;

add[r]+=d;

Max[r]+=d;

}

void Update\_Rev(int r)

{

if(!r)

return;

swap(ch[r][0],ch[r][1]);

rev[r]^=1;

}

void push\_down(int r)

{

if(add[r])

{

Update\_Add(ch[r][0],add[r]);

Update\_Add(ch[r][1],add[r]);

add[r]=0;

}

if(rev[r])

{

Update\_Rev(ch[r][0]);

Update\_Rev(ch[r][1]);

rev[r]=0;

}

}

void push\_up(int r)

{

Max[r]=max(max(Max[ch[r][0]],Max[ch[r][1]]),key[r]);

}

void Rotate(int x)

{

int y=pre[x],kind=ch[y][1]==x;

ch[y][kind]=ch[x][!kind];

pre[ch[y][kind]]=y;

pre[x]=pre[y];

pre[y]=x;

ch[x][!kind]=y;

if(rt[y])

rt[y]=false,rt[x]=true;

else

ch[pre[x]][ch[pre[x]][1]==y]=x;

push\_up(y);

}

//P函数先将根结点到r的路径上所有的结点的标记逐级下放

void P(int r)

{

if(!rt[r])

P(pre[r]);

push\_down(r);

}

void Splay(int r)

{

P(r);

while(!rt[r])

{

int f=pre[r],ff=pre[f];

if(rt[f])

Rotate(r);

else if((ch[ff][1]==f)==(ch[f][1]==r))

Rotate(f),Rotate(r);

else

Rotate(r),Rotate(r);

}

push\_up(r);

}

int Access(int x)

{

int y=0;

for (;x;x=pre[y=x])

{

Splay(x);

rt[ch[x][1]]=true,rt[ch[x][1]=y]=false;

push\_up(x);

}

return y;

}

//判断是否是同根(真实的树，非splay)

bool judge(int u, int v)

{

while(pre[u])

u=pre[u];

while(pre[v])

v=pre[v];

return u==v;

}

//使r成为它所在的树的根

void mroot(int r)

{

Access(r);

Splay(r);

Update\_Rev(r);

}

//调用后u是原来u和v的lca,v和ch[u][1]分别存着lca的2个儿子

//(原来u和v所在的2颗子树)

void lca(int &u, int &v)

{

Access(v),v=0;

while(u)

{

Splay(u);

if(!pre[u])

return;

rt[ch[u][1]]=true;

rt[ch[u][1]=v]=false;

push\_up(u);

u=pre[v=u];

}

}

void link(int u, int v)

{

if(judge(u,v))

{

puts("-1");

return;

}

mroot(u);

pre[u]=v;

}

//使u成为u所在树的根，并且v和它父亲的边断开

void cut(int u, int v)

{

if(u==v||!judge(u,v))

{

puts("-1");

return;

}

mroot(u);

Splay(v);

pre[ch[v][0]]=pre[v];

pre[v]=0;

rt[ch[v][0]]=true;

ch[v][0]=0;

push\_up(v);

}

void ADD(int u,int v,int w)

{

if(!judge(u,v))

{

puts("-1");

return;

}

lca(u,v);

Update\_Add(ch[u][1],w);

Update\_Add(v,w);

key[u]+=w;

push\_up(u);

}

void query(int u,int v)

{

if(!judge(u,v))

{

puts("-1");

return;

}

lca(u,v);

printf("%d\n",max(max(Max[v],Max[ch[u][1]]),key[u]));

}

struct Edge

{

int to,next;

}edge[MAXN\*2];

int head[MAXN],tot;

void addedge(int u,int v)

{

edge[tot].to=v;

edge[tot].next=head[u];

head[u]=tot++;

}

void dfs(int u)

{

for (int i=head[u];i!=-1;i=edge[i].next)

{

int v=edge[i].to;

if(pre[v]!=0)

continue;

pre[v]=u;

dfs(v);

}

}

int main()

{

//freopen("in.txt","r",stdin);

//freopen("out.txt","w",stdout);

int n,q,u,v;

while(scanf("%d",&n)==1)

{

tot=0;

for (int i=0;i<=n;i++)

{

head[i]=-1;

pre[i]=0;

ch[i][0]=ch[i][1]=0;

rev[i]=0;

add[i]=0;

rt[i]=true;

}

Max[0]=-2000000000;

for(int i=1;i<n;i++)

{

scanf("%d%d",&u,&v);

addedge(u,v);

addedge(v,u);

}

for(int i=1;i<=n;i++)

{

scanf("%d",&key[i]);

Max[i]=key[i];

}

scanf("%d",&q);

pre[1]=-1;

dfs(1);

pre[1]=0;

int op;

while(q--)

{

scanf("%d",&op);

if (op==1)

{

int x,y;

scanf("%d%d",&x,&y);

link(x,y);

}

else if(op==2)

{

int x, y;

scanf("%d%d",&x,&y);

cut(x,y);

}

else if(op==3)

{

int w,x,y;

scanf("%d%d%d",&w,&x,&y);

ADD(x,y,w);

}

else

{

int x, y;

scanf("%d%d",&x,&y);

query(x,y);

}

}

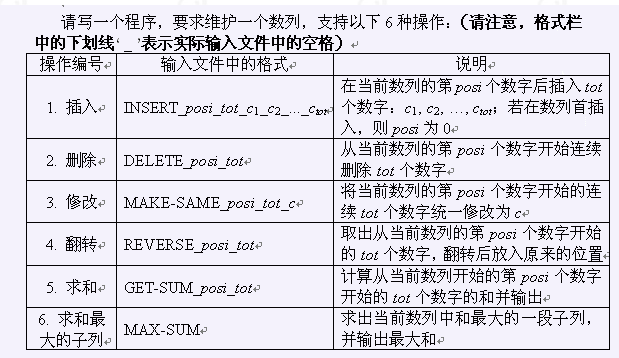
printf("\n");

}

return 0;

}

## Splay树



//题目：维修数列。

//经典题，插入、删除、修改、翻转、求和、求和最大的子序列

const int MAXN=500010;

const int INF=0x3f3f3f3f;

int a[MAXN];

int n,q;

namespace Splay\_Tree

{

#define Key\_value ch[ch[root][1]][0]

int pre[MAXN],ch[MAXN][2],key[MAXN],size[MAXN];

int root,tot1;

int sum[MAXN],rev[MAXN],same[MAXN];

int lx[MAXN],rx[MAXN],mx[MAXN];

int s[MAXN],tot2;//内存池和容量

//debug部分\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void Treavel(int x)

{

if(x)

{

Treavel(ch[x][0]);

printf("结点：%2d: 左儿子 %2d 右儿子 %2d 父结点 %2d size = % 2d\n",x,ch[x][0],ch[x][1],pre[x],size[x]);

Treavel(ch[x][1]);

}

}

void debug()

{

printf("root:%d\n", root);

Treavel(root);

}

//以上是debug部分\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void NewNode(int &r,int father,int k)

{

if(tot2)

r=s[tot2--];//取的时候是tot2--,存的时候就是++tot2

else

r=++tot1;

pre[r]=father;

ch[r][0]=ch[r][1]=0;

key[r]=k;

sum[r]=k;

rev[r]=same[r]=0;

lx[r]=rx[r]=mx[r]=k;

size[r]=1;

}

void Update\_Rev(int r)

{

if(!r)

return;

swap(ch[r][0],ch[r][1]);

swap(lx[r],rx[r]);

rev[r]^=1;

}

void Update\_Same(int r, int v)

{

if(!r)

return;

key[r]=v;

sum[r]=v\*size[r];

lx[r]=rx[r]=mx[r]=max(v,v\*size[r]);

same[r]=1;

}

void push\_up(int r)

{

int lson=ch[r][0],rson=ch[r][1];

size[r]=size[lson]+size[rson]+1;

sum[r]=sum[lson]+sum[rson]+key[r];

lx[r]=max(lx[lson],sum[lson]+key[r]+max(0,lx[rson]));

rx[r]=max(rx[rson],sum[rson]+key[r]+max(0,rx[lson]));

mx[r]=max(0,rx[lson])+key[r]+max(0,lx[rson]);

mx[r]=max(mx[r],max(mx[lson],mx[rson]));

}

void push\_down(int r)

{

if(same[r])

{

Update\_Same(ch[r][0],key[r]);

Update\_Same(ch[r][1],key[r]);

same[r]=0;

}

if(rev[r])

{

Update\_Rev(ch[r][0]);

Update\_Rev(ch[r][1]);

rev[r]=0;

}

}

void Build(int &x,int l,int r,int father)

{

if(l>r)

return;

int mid=(l+r)/2;

NewNode(x,father,a[mid]);

Build(ch[x][0],l,mid-1,x);

Build(ch[x][1],mid+1,r,x);

push\_up(x);

}

void Init()

{

root=tot1=tot2=0;

ch[root][0]=ch[root][1]=size[root]=pre[root]=0;

same[root]=rev[root]=sum[root]=key[root]=0;

lx[root]=rx[root]=mx[root]=-INF;

NewNode(root,0,-1);

NewNode(ch[root][1],root,-1);

for(int i=0;i<n;i++)

scanf("%d",&a[i]);

Build(Key\_value,0,n-1,ch[root][1]);

push\_up(ch[root][1]);

push\_up(root);

}

//旋转,0为左旋，1为右旋

void Rotate(int x,int kind)

{

int y=pre[x];

push\_down(y);

push\_down(x);

ch[y][!kind]=ch[x][kind];

pre[ch[x][kind]]=y;

if(pre[y])

ch[pre[y]][ch[pre[y]][1]==y]=x;

pre[x]=pre[y];

ch[x][kind]=y;

pre[y]=x;

push\_up(y);

}

//Splay调整，将r结点调整到goal下面

void Splay(int r,int goal)

{

push\_down(r);

while(pre[r]!=goal)

{

if(pre[pre[r]]==goal)

{

push\_down(pre[r]);

push\_down(r);

Rotate(r,ch[pre[r]][0]==r);

}

else

{

push\_down(pre[pre[r]]);

push\_down(pre[r]);

push\_down(r);

int y=pre[r];

int kind=ch[pre[y]][0]==y;

if(ch[y][kind]==r)

{

Rotate(r,!kind);

Rotate(r,kind);

}

else

{

Rotate(y,kind);

Rotate(r,kind);

}

}

}

push\_up(r);

if(goal==0)

root=r;

}

//获得第K个位置数

int Get\_kth(int r,int k)

{

push\_down(r);

int t=size[ch[r][0]]+1;

if(t==k)

return r;

if(t>k)

return Get\_kth(ch[r][0],k);

else

return Get\_kth(ch[r][1],k-t);

}

//在第pos个数后面插入tot个数

void Insert(int pos,int tot)

{

for(int i=0;i<tot;i++)

scanf("%d",&a[i]);

Splay(Get\_kth(root,pos+1),0);

Splay(Get\_kth(root,pos+2),root);

Build(Key\_value,0,tot-1,ch[root][1]);

push\_up(ch[root][1]);

push\_up(root);

}

//删除子树

void erase(int r)

{

if(!r)

return;

s[++tot2]=r;

erase(ch[r][0]);

erase(ch[r][1]);

}

//从第pos个数开始连续删除tot个数

void Delete(int pos,int tot)

{

Splay(Get\_kth(root,pos),0);

Splay(Get\_kth(root,pos+tot+1),root);

erase(Key\_value);

pre[Key\_value]=0;

Key\_value=0;

push\_up(ch[root][1]);

push\_up(root);

}

//将从第pos个数开始的连续的tot个数修改为c

void Make\_Same(int pos,int tot,int c)

{

Splay(Get\_kth(root,pos),0);

Splay(Get\_kth(root,pos+tot+1),root);

Update\_Same(Key\_value,c);

push\_up(ch[root][1]);

push\_up(root);

}

//将第pos个数开始的连续tot个数进行反转

void Reverse(int pos,int tot)

{

Splay(Get\_kth(root,pos),0);

Splay(Get\_kth(root,pos+tot+1),root);

Update\_Rev(Key\_value);

push\_up(ch[root][1]);

push\_up(root);

}

//得到第pos个数开始的tot个数的和

int Get\_Sum(int pos,int tot)

{

Splay(Get\_kth(root,pos),0);

Splay(Get\_kth(root,pos+tot+1),root);

return sum[Key\_value];

}

//得到第pos个数开始的tot个数中最大的子段和

int Get\_MaxSum(int pos,int tot)

{

Splay(Get\_kth(root,pos),0);

Splay(Get\_kth(root,pos+tot+1),root);

return mx[Key\_value];

}

void InOrder(int r)

{

if(!r)

return;

push\_down(r);

InOrder(ch[r][0]);

printf("%d",key[r]);

InOrder(ch[r][1]);

}

//找前驱(需要push\_down)

int Get\_pre(int r)

{

push\_down(r);

if(ch[r][0]==0)return -1;//不存在

r=ch[r][0];

while(ch[r][1])

{

r=ch[r][1];

push\_down(r);

}

return r;

}

//找后继(需要push\_down)

int Get\_next(int r)

{

push\_down(r);

if(ch[r][1]==0)return -1;

r=ch[r][1];

while(ch[r][0])

{

r=ch[r][0];

push\_down(r);

}

return r;

}

}

using namespace Splay\_Tree;

int main()

{

//freopen("in.txt","r",stdin);

//freopen("out.txt","w",stdout);

while(scanf("%d%d", &n, &q)==2)

{

Init();

char op[20];

int x,y,z;

while (q--)

{

scanf("%s",op);

if(strcmp(op,"INSERT")==0)

{

scanf("%d%d",&x,&y);

Insert(x,y);

}

else if(strcmp(op,"DELETE")==0)

{

scanf("%d%d",&x,&y);

Delete(x,y);

}

else if(strcmp(op,"MAKE-SAME")==0)

{

scanf("%d%d%d",&x,&y,&z);

Make\_Same(x,y,z);

}

else if(strcmp(op,"REVERSE")==0)

{

scanf("%d%d",&x,&y);

Reverse(x,y);

}

else if(strcmp(op,"GET-SUM")==0)

{

scanf("%d%d",&x,&y);

printf("%d\n",Get\_Sum(x,y));

}

else if(strcmp(op,"MAX-SUM")==0)

printf("%d\n",Get\_MaxSum(1,size[root]-2));

}

}

return 0;

}

## KDTree

#define x first

#define y second

const int MAXN=2e5+5;

const int inf=2e9;

namespace KD\_Tree

{

struct node

{

node \*ch[2];

int d[2],mx[2],my[2],size;//d表示这个点的坐标[0]->x,[1]->y,mx表示这个平面的x的范围，my表示这个平面的y的范围

inline void push\_up()

{

size=1;

for(int i=0;i<=1;i++)

{

if(ch[i])

{

mx[0]=min(mx[0],ch[i]->mx[0]);

mx[1]=max(mx[1],ch[i]->mx[1]);

my[0]=min(my[0],ch[i]->my[0]);

my[1]=max(my[1],ch[i]->my[1]);

size+=ch[i]->size;

}

}

}

}pool\_node[MAXN],\*pool\_top=pool\_node;

node \*del\_pool[MAXN],\*\*del\_top=del\_pool;

inline node\* newnode()

{

return del\_top==del\_pool?++pool\_top:\*(del\_top--);

}

typedef pair<int,int> Point;

inline bool cmp\_x(const Point &a,const Point &b) {return a.x<b.x;}

inline bool cmp\_y(const Point &a,const Point &b) {return a.y<b.y;}

node \*\*rebuild\_need;

int rebuild\_d;

Point stk[MAXN];

int point\_cnt;

//用于最开始建树，可以不用直接插点，方法为build(1,point\_cnt,0),保证点在stk数组中

node\* build(int l,int r,bool f)

{

int mid=(l+r)>>1;

node \*o=newnode();

nth\_element(stk+l,stk+mid,stk+r+1,!f?cmp\_x:cmp\_y);

o->d[0]=o->mx[0]=o->mx[1]=stk[mid].x;

o->d[1]=o->my[0]=o->my[1]=stk[mid].y;

o->ch[0]=l<mid?build(l,mid-1,f^1):0;

o->ch[1]=mid<r?build(mid+1,r,f^1):0;

o->push\_up();

return o;

}

void remove(node \*o)

{

if(o->ch[0])

remove(o->ch[0]);

if(o->ch[1])

remove(o->ch[1]);

stk[++point\_cnt]=Point(o->d[0],o->d[1]);

\*(++del\_top)=o;

}

/\*

插入以及重构代码如下

insert(root,x,y,0);

if(rebuild\_need)

rebuild();

\*/

void rebuild()

{

point\_cnt=0;

remove(\*rebuild\_need);

\*rebuild\_need=build(1,point\_cnt,rebuild\_d);

rebuild\_need=0;

}

void insert(node \*&o,int x,int y,bool f)

{

if(!o)

{

o=newnode();

o->d[0]=o->mx[0]=o->mx[1]=x;

o->d[1]=o->my[0]=o->my[1]=y;

}

else if(o->d[0]!=x||o->d[1]!=y)

{

int d=!f?o->d[0]<x:o->d[1]<y;

insert(o->ch[d],x,y,f^1);

o->push\_up();

if(o->ch[d]->size\*10>=o->size\*7)

rebuild\_need=&o,rebuild\_d=f;

}

}

Point P;

int ans;

int calc\_mx(node \*o)//同下calc\_mn

{

int ret=0;

ret+=max(abs(o->mx[0]-P.x),abs(o->mx[1]-P.x));

ret+=max(abs(o->my[0]-P.y),abs(o->my[1]-P.y));

return ret;

}

int calc\_mn(node \*o)//Manhattan距离下进行计算，如果是欧氏距离，则为max(P.x-o->mx[1],0)+max(o->mx[0]-P.x,0)的平方（本处表示到这个子树平面的最短距离）

{

int ret=0;

ret+=max(P.x-o->mx[1],0)+max(o->mx[0]-P.x,0);

ret+=max(P.y-o->my[1],0)+max(o->my[0]-P.y,0);

return ret;

}

void Query\_Max(node \*o)//查询平面中到目标点最远的点

{

ans=max(ans,abs(o->d[0]-P.x)+abs(o->d[1]-P.y));

int dl=o->ch[0]?calc\_mx(o->ch[0]):-inf;

int dr=o->ch[1]?calc\_mx(o->ch[1]):-inf;

if(dl>dr)

{

if(dl>ans)

Query\_Max(o->ch[0]);

if(dr>ans)

Query\_Max(o->ch[1]);

}

else

{

if(dr>ans)

Query\_Max(o->ch[1]);

if(dl>ans)

Query\_Max(o->ch[0]);

}

}

void Query\_Min(node \*o)//查询平面中到目标点最近的点

{

ans=min(ans,abs(o->d[0]-P.x)+abs(o->d[1]-P.y));

int dl=o->ch[0]?calc\_mn(o->ch[0]):inf;

int dr=o->ch[1]?calc\_mn(o->ch[1]):inf;

if(dl<dr)

{

if(dl<ans)

Query\_Min(o->ch[0]);

if(dr<ans)

Query\_Min(o->ch[1]);

}

else

{

if(dr<ans)

Query\_Min(o->ch[1]);

if(dl<ans)

Query\_Min(o->ch[0]);

}

}

node \*root;

int Query(int x,int y,bool f)

{

P=Point(x,y);

if(f)

ans=-inf,Query\_Max(root);

else

ans=inf,Query\_Min(root);

return ans;

}

//以下三个函数为本人手写，所以可靠性有待验证，不过过过题....

void debug(node \*&a)

{

if(a->ch[0])

{

debug(a->ch[0]);

}

printf("%d %d\n",a->d[0],a->d[1]);

if(a->ch[1])

{

debug(a->ch[1]);

}

}

void del(node \*&a)//用于清空整颗树

{

for(int i=0;i<=1;i++)

{

if(a->ch[i])

{

del(a->ch[i]);

}

}

a=NULL;

}

void init()//用于初始化

{

del\_top=del\_pool,pool\_top=pool\_node;

}

}

## Treap

struct Treap

{

struct data

{

int l,r,v,size,rnd,w;

}tr[100005];

int n,size,root,ans;

void update(int k)//更新结点信息

{

tr[k].size=tr[tr[k].l].size+tr[tr[k].r].size+tr[k].w;

}

//旋转

void rturn(int &k)

{

int t=tr[k].l;tr[k].l=tr[t].r;tr[t].r=k;

tr[t].size=tr[k].size;update(k);k=t;

}

void lturn(int &k)

{

int t=tr[k].r;tr[k].r=tr[t].l;tr[t].l=k;

tr[t].size=tr[k].size;update(k);k=t;

}

void insert(int &k,int x)//插入x

{

if(k==0)

{

size++;k=size;

tr[k].size=tr[k].w=1;tr[k].v=x;tr[k].rnd=rand();

return;

}

tr[k].size++;

if(tr[k].v==x)

tr[k].w++;//每个结点顺便记录下与该节点相同值的数的个数

else if(x>tr[k].v)

{

insert(tr[k].r,x);

if(tr[tr[k].r].rnd<tr[k].rnd)

lturn(k);//维护堆性质

}

else

{

insert(tr[k].l,x);

if(tr[tr[k].l].rnd<tr[k].rnd)

rturn(k);

}

}

void del(int &k,int x)//删除x

{

if(k==0)

return;

if(tr[k].v==x)

{

if(tr[k].w>1)

{

tr[k].w--;tr[k].size--;

return;//若不止相同值的个数有多个，删去一个

}

if(tr[k].l\*tr[k].r==0)

k=tr[k].l+tr[k].r;//有一个儿子为空

else if(tr[tr[k].l].rnd<tr[tr[k].r].rnd)

rturn(k),del(k,x);

else

lturn(k),del(k,x);

}

else if(x>tr[k].v)

tr[k].size--,del(tr[k].r,x);

else

tr[k].size--,del(tr[k].l,x);

}

int query\_rank(int k,int x)//x的排名

{

if(k==0)

return 0;

if(tr[k].v==x)

return tr[tr[k].l].size+1;

else if(x>tr[k].v)

return tr[tr[k].l].size+tr[k].w+query\_rank(tr[k].r,x);

else

return query\_rank(tr[k].l,x);

}

int query\_num(int k,int x)//排名为x的数

{

if(k==0)

return 0;

if(x<=tr[tr[k].l].size)

return query\_num(tr[k].l,x);

else if(x>tr[tr[k].l].size+tr[k].w)

return query\_num(tr[k].r,x-tr[tr[k].l].size-tr[k].w);

else

return tr[k].v;

}

void query\_pro(int k,int x)//前驱

{

if(k==0)

return;

if(tr[k].v<x)

{

ans=k;

query\_pro(tr[k].r,x);

}

else

query\_pro(tr[k].l,x);

}

void query\_sub(int k,int x)//后继

{

if(k==0)

return;

if(tr[k].v>x)

{

ans=k;

query\_sub(tr[k].l,x);

}

else

query\_sub(tr[k].r,x);

}

};

#include<bits/stdc++.h>

#define xx first

#define yy second

using namespace std;

const int MAXN=1e5+5;

struct Treap

{

struct node

{

int fix,key,size;

int l,r;

node(){}

node(int \_key):fix(rand()),key(\_key),l(0),r(0),size(1){}

}tr[MAXN];

typedef pair<int,int> Droot;//用来Split返回两个根

inline void updata(int o)

{

tr[o].size=1+(tr[o].l?tr[tr[o].l].size:0)+(tr[o].r?tr[tr[o].r].size:0);

}

inline int Size(int x)

{

return x?tr[x].size:0;

}//这样求size可以防止访问空指针

int tot,root;

void init()

{

tot=0;

}

inline int newnode(int val)

{

int ret=++tot;

tr[tot]=node(val);

return ret;

}

//合并操作

int Merge(int A,int B)

{

if(!A)

return B;

if(!B)

return A;

if(tr[A].fix<tr[B].fix)

{

tr[A].r=Merge(tr[A].r,B);

updata(A);

return A;

}

else

{

tr[B].l=Merge(A,tr[B].l);

updata(B);

return B;

}

}

//拆分操作

Droot Split(int x,int k)

{

if(!x)

return Droot(0,0);

Droot ret;

if(Size(tr[x].l)>=k)

{

ret=Split(tr[x].l,k);

tr[x].l=ret.yy;

updata(x);

ret.yy=x;

}

else

{

ret=Split(tr[x].r,k-Size(tr[x].l)-1);

tr[x].r=ret.xx;

updata(x);

ret.xx=x;

}

return ret;

}

//建造操作

int stack[MAXN],x,last;

int Build(int \*a,int n)

{

int p=0;

for(int i=1;i<=n;i++)

{

x=newnode(a[i]);

last=0;

while(p&&tr[stack[p]].fix>tr[x].fix)

{

updata(stack[p]);

last=stack[p];

stack[p--]=0;

}

if(p)

tr[stack[p]].r=x;

tr[x].l=last;

stack[++p]=x;

}

while(p)

updata(stack[p--]);

return stack[1];

}

//查找第K小

int Findkth(int k)

{

Droot x=Split(root,k-1);

Droot y=Split(x.yy,1);

int ret=y.xx;

root=Merge(Merge(x.xx,ret),y.yy);

return tr[ret].key;

}

//询问一个数是第几大

int Getkth(int x,int v)

{

if(!x)

return 0;

return v<tr[x].key?Getkth(tr[x].l,v):Getkth(tr[x].r,v)+Size(tr[x].l)+1;

}

//插入操作

void Insert(int v)

{

int k=Getkth(root,v);

Droot x=Split(root,k);

int o=newnode(v);

root=Merge(Merge(x.xx,o),x.yy);

}

//删除操作

void Delete(int k)

{

Droot x=Split(root,k-1);

Droot y=Split(x.yy,1);

root=Merge(x.xx,y.yy);

}

};

## 线性基

const int MAXL=50;//位数

typedef long long ll;

struct LinearBasis

{

ll v[MAXL+1];

int sz,n;

LinearBasis()

{

memset(v,0,sizeof(v));

}

void add(ll x)

{

for(int i=MAXL;i>=0;i--)

{

if(x&(1LL<<i))

{

if(!v[i])

{

v[i]=x;

break;

}

x^=v[i];

}

}

}

ll queryMax()

{

ll x=0;

for(int i=MAXL;i>=0;i--)

if((x^v[i])>x)

x^=v[i];

return x;

}

//第k大预处理

void init(int \_n)

{

n=\_n;sz=0;

for(int i=0;i<=MAXL;i++)

for(int j=i+1;j<=MAXL;j++)

if((v[j]>>i)&1)

v[j]^=v[i];

for(int i=0;i<=MAXL;i++)

if(v[i])

v[sz++]=v[i];

}

ll query(ll k)

{

if(sz!=n)

k--;

if(k>(1ll<<sz)-1)

return -1;

ll ans=0;

for(int i=0;i<=MAXL;i++)

if(k&(1LL<<i))

ans^=v[i];

return ans;

}

}lb;

# 数论

## CRT（互质）

void exgcd(ll a,ll b,ll& d,ll& x,ll& y)

{

if(!b)

{

d=a;x=1;y=0;

}

else

{

exgcd(b,a%b,d,y,x);

y-=x\*(a/b);

}

}

ll inv(ll a, ll n) //mod n a逆元

{

ll d,x,y;

exgcd(a,n,d,x,y);

return d==1?(x+n)%n:-1;

}

/\*

x = ai(mod mi)

m1 \* m2 \* ... \* mi = M

mi之间互素

x = sigma(ai \* Mi \* Mi(-1)) % M

Mi = M / mi

Mi(-1) = Mi 模mi的逆

\*/

ll mul(ll a,ll b,ll M)

{

ll ret=0;

while(b)

{

if(b&1)

ret=(ret+a)%M;

a=a\*2%M;

b>>=1;

}

return ret;

}

ll CRT(ll a[],ll m[],ll n)//a余数，m模数

{

ll M=1;

ll ans=0;

for(int i=1;i<=n;i++)

M\*=m[i];

for(int i=1;i<=n;i++)

{

ll Mi=M/m[i];

ll x,y,d;

exgcd(Mi,m[i],d,x,y);

ans=(ans+mul(x,mul(a[i],Mi,M),M))%M;

}

if(ans<0)

ans+=M;

return ans;

}

## CRT（不互质）

#include<bits/stdc++.h>

using namespace std;

typedef long long ll;

const int MAXN=1005;

ll a[MAXN],m[MAXN];

ll gcd(ll a,ll b)

{

return b?gcd(b,a%b):a;

}

void extend\_Euclid(ll a,ll b,ll &x,ll &y)

{

if(b==0)

{

x=1;

y=0;

return;

}

extend\_Euclid(b,a%b,x,y);

ll tmp=x;

x=y;

y=tmp-(a/b)\*y;

}

ll Inv(ll a,ll b)

{

ll d=gcd(a,b);

if(d!=1) return -1;

ll x,y;

extend\_Euclid(a,b,x,y);

return (x%b+b)%b;

}

bool merge(ll a1,ll m1,ll a2,ll m2,ll &a3,ll &m3)

{

ll d=gcd(m1,m2);

ll c=a2-a1;

if(c%d) return false;

c=(c%m2+m2)%m2;

m1/=d;

m2/=d;

c/=d;

c\*=Inv(m1,m2);

c%=m2;

c\*=m1\*d;

c+=a1;

m3=m1\*m2\*d;

a3=(c%m3+m3)%m3;

return true;

}

ll CRT(ll a[],ll m[],int n)

{

ll a1=a[1];

ll m1=m[1];

for(int i=2; i<=n; i++)

{

ll a2=a[i];

ll m2=m[i];

ll m3,a3;

if(!merge(a1,m1,a2,m2,a3,m3))

return -1;

a1=a3;

m1=m3;

}

return (a1%m1+m1)%m1;

}

int main()

{

int n;

while(scanf("%d",&n)!=EOF)

{

for(int i=1;i<=n;i++)

scanf("%I64d%I64d",&m[i],&a[i]);

ll ans=CRT(a,m,n);

printf("%I64d\n",ans);

}

return 0;

}

## 素数筛

typedef long long ll;

int pri[MAXN],phi[MAXN];

bool vis[MAXN];

int tot;

ll sum[MAXN];

void init()

{

int n=MAXN;

tot=0;

memset(vis,false,sizeof(vis));

phi[1]=1;

for(int i=2;i<n;i++)

{

if(!vis[i])

{

pri[tot++]=i;

phi[i]=i-1;

}

for(int j=0;j<tot && i\*pri[j]<n;j++)

{

vis[i\*pri[j]]=true;

if(i%pri[j]==0)

{

phi[i\*pri[j]]=phi[i]\*pri[j];

break;

}

else phi[i\*pri[j]]=phi[i]\*(pri[j]-1);

}

}

sum[0]=0;

for(int i=1;i<MAXN;i++)

sum[i]=(sum[i-1]+phi[i])%M;

}

## 大区间素数筛

/\*

\* POJ 2689 Prime Distance

\* 给出一个区间[L,U]，找出区间内容、相邻的距离最近的两个素数和

\* 距离最远的两个素数。

\* 1<=L<U<=2,147,483,647 区间长度不超过1,000,000

\* 就是要筛选出[L,U]之间的素数

\*/

#include <stdio.h>

#include <algorithm>

#include <iostream>

#include <string.h>

using namespace std;

const int MAXN=100010;

int prime[MAXN+1];

void getPrime()

{

memset(prime,0,sizeof(prime));

for(int i=2;i<=MAXN;i++)

{

if(!prime[i])prime[++prime[0]]=i;

for(int j=1;j<=prime[0]&&prime[j]<=MAXN/i;j++)

{

prime[prime[j]\*i]=1;

if(i%prime[j]==0)break;

}

}

}

bool notprime[1000010];

int prime2[1000010];

void getPrime2(int L,int R)

{

memset(notprime,false,sizeof(notprime));

if(L<2)L=2;

for(int i=1;i<=prime[0]&&(long long)prime[i]\*prime[i]<=R;i++)

{

int s=L/prime[i]+(L%prime[i]>0);

if(s==1)s=2;

for(int j=s;(long long)j\*prime[i]<=R;j++)

if((long long)j\*prime[i]>=L)

notprime[j\*prime[i]-L]=true;

}

prime2[0]=0;

for(int i=0;i<=R-L;i++)

if(!notprime[i])

prime2[++prime2[0]]=i+L;

}

int main()

{

getPrime();

int L,U;

while(scanf("%d%d",&L,&U)==2)

{

getPrime2(L,U);

if(prime2[0]<2)printf("There are no adjacent primes.\n");

else

{

int x1=0,x2=100000000,y1=0,y2=0;

for(int i=1;i<prime2[0];i++)

{

if(prime2[i+1]-prime2[i]<x2-x1)

{

x1=prime2[i];

x2=prime2[i+1];

}

if(prime2[i+1]-prime2[i]>y2-y1)

{

y1=prime2[i];

y2=prime2[i+1];

}

}

printf("%d,%d are closest, %d,%d are most distant.\n",x1,x2,y1,y2);

}

}

}

## 素因子分解

void pri(ll x)//寻找x的所有素因子

{

int i;

for(i=2; i\*i <= x; i++)

{

if(x%i==0)

gcd.push\_back(i);

while(x%i==0)

x=x/i;

}

if(x!=1)

gcd.push\_back(x);

gcd.erase(unique(gcd.begin(), gcd.end()), gcd.end());//清除重复元素

}

## 扩展欧几里得

ll exgcd(ll a,ll b,ll &x,ll &y)//extended Euclidean algorithm

{

ll r,t;

if(b==0)

{

x=1;

y=0;

return a;

}

r=exgcd(b,a%b,x,y);

t=x;

x=y;

y=t-a/b\*y;

return r;

}

ll cal(ll a,ll b,ll c)//the minimum solution of ax+by=c also can use to find the inverse element

{

ll x,y;

ll gcd=e\_gcd(a,b,x,y);

if(c%gcd!=0) return -1;

x\*=c/gcd;

b/=gcd;

if(b<0) b=-b;

ll ans=x%b;

if(ans<=0) ans+=b;

return ans;

}

## 矩阵快速幂

struct Matrix

{

ll a[205][205];

Matrix()

{

for(int i=0;i<=200;i++)

for(int j=0;j<=200;j++)

a[i][j]=-INF;

}

Matrix operator \* (const Matrix &B)const

{

Matrix C;

for(int i=0;i<=200;i++)

for(int k=0;k<=200;k++)

for(int j=0;j<=200;j++)

C.a[i][j]=a[i][k]+B.a[k][j];

return C;

}

Matrix operator ^ (const ll &t)const

{

Matrix A=(\*this),res=A;

ll p=t-1;

while(p)

{

if(p&1)res=res\*A;

A=A\*A;

p>>=1;

}

return res;

}

};

## 组合数

ll fac[MAXN];

ll qpow(ll a,ll b)

{

ll ans=1;a%=mod;

for(ll i=b;i;i>>=1,a=a\*a%mod)

if(i&1)ans=ans\*a%mod;

return ans;

}

ll C(ll n,ll m)

{

if(m>n||m<0)return 0;

ll s1=fac[n],s2=fac[n-m]\*fac[m]%mod;

return s1\*qpow(s2,mod-2)%mod;//费马小定理求逆元

}

void init()

{

fac[0]=1;

for(int i=1;i<MAXN;i++)//阶乘打表

fac[i]=fac[i-1]\*i%mod;

}

ll Fac(ll n)//递推公式（预处理）

{

CC[0]=1;

for(ll i=1;i<MAXN;i++)

{

f[i]=qpow(i,mod-2);

}

for(ll i=1;i<=k;i++){

CC[i]=(CC[i-1]\*(n-i+1)%mod)\*f[i]%mod;

}

}

ll \_inv(int x)//求逆元

{

if(x == 1)

return 1;

return ll(mod-mod/x)\*\_inv(mod%x)%mod;

}

void init(){

jie[0]=1;

for(int i=1;i<=64;i++)

{

jie[i]=((LL)jie[i-1]\*i)%mod;

}

for(int i=0;i<=64;i++)

{

zuhe[i][0]=1;

for(int j=1;j<=i;j++)

{

zuhe[i][j]=zuhe[i-1][j]+zuhe[i-1][j-1];

zuhe[i][j]%=mod;

}

}

}

## 高斯消元（int）

#include<bits/stdc++.h>

using namespace std;

const int MOD=3;

const int MAXN=1005;

int a[MAXN][MAXN];

int sv[MAXN],x[MAXN];

int Gauss(int equ,int var)

{

int row,max\_r,col;

for(row=0,col=0;col<var&&row<equ;col++,row++)

{

max\_r=row;

for(int i=row+1;i<equ;i++)

{

if(abs(a[i][col])>abs(a[max\_r][col]))

{

max\_r=i;

}

}

if(!a[max\_r][col])

{

row--;

continue;

}

if(row!=max\_r)

{

for(int j=0;j<=var;j++)

{

swap(a[row][j],a[max\_r][j]);

}

}

for(int i=row+1;i<equ;i++)

{

if(a[i][col])

{

int lcm=a[i][col]/\_\_gcd(a[i][col],a[row][col])\*a[row][col];

int t1=lcm/a[i][col];

int t2=lcm/a[row][col];

for(int j=col;j<=var;j++)

{

a[i][j]=((a[i][j]\*t1-a[row][j]\*t2)%MOD+MOD)%MOD;

}

}

}

}

for(int i=var-1;i>=0;i--)

{

x[i]=a[i][var];

for(int j=i+1;j<var;j++)

{

x[i]=((x[i]-a[i][j]\*x[j])%MOD+MOD)%MOD;

}

x[i]=a[i][i]\*x[i]%MOD;//mod 3涓嬬殑閫嗗厓灏辨槸鍏舵湰韬?

}

}

int all;

bool check(int u)

{

return u>=0&&u<all;

}

int main()

{

int T;

scanf("%d",&T);

while(T--)

{

int n,m;

scanf("%d%d",&n,&m);

for(int i=0;i<n;i++)

{

for(int j=0;j<m;j++)

{

scanf("%d",&sv[i\*m+j]);

}

}

all=m\*n;

memset(a,0,sizeof(a));

for(int i=0;i<all;i++)

{

int d=i-m;

int u=i+m;

int l=i-1;

int r=i+1;

a[i][i]=2;

a[i][all]=(3-sv[i])%MOD;

if(check(u))

{

a[i][u]=1;

}

if(check(d))

{

a[i][d]=1;

}

if(check(l)&&i%m!=0)

{

a[i][l]=1;

}

if(check(r)&&i%m!=m-1)

{

a[i][r]=1;

}

}

Gauss(all,all);

vector<int> ans;

for(int i=0;i<all;i++)

{

while(x[i])

{

x[i]--;

ans.push\_back(i);

}

}

printf("%d\n",ans.size());

for(int i=0;i<ans.size();i++)

{

int x=ans[i]/m;

int y=ans[i]%m;

printf("%d %d\n",x+1,y+1);

}

}

}

## 高斯消元（double）

const double eps=1e-9;

const int MAXN=220;

double a[MAXN][MAXN],x[MAXN];//方程的左边的矩阵和等式右边的值，求解之后x存的就是结果

int equ,var;//方程数和未知数个数

/\*

\*返回0表示无解，1表示有解

\*/

int Gauss()

{

int i,j,k,col,max\_r;

for(k=0,col=0;k<equ&&col<var;k++,col++)

{

max\_r=k;

for(i=k+1;i<equ;i++)

if(fabs(a[i][col])>fabs(a[max\_r][col]))

max\_r=i;

if(fabs(a[max\_r][col])<eps)return 0;

if(k!=max\_r)

{

for(j=col;j<var;j++)

swap(a[k][j],a[max\_r][j]);

swap(x[k],x[max\_r]);

}

x[k]/=a[k][col];

for(j=col+1;j<var;j++)a[k][j]/=a[k][col];

a[k][col]=1;

for(i=0;i<equ;i++)

if(i!=k)

{

x[i]-=x[k]\*a[i][k];

for(j=col+1;j<var;j++)a[i][j]-=a[k][j]\*a[i][col];

a[i][col]=0;

}

}

return 1;

}

## FFT

const int MAXN=50005\*4;//玄学：开成两倍会有问题(1<<17=131072,1<<18=262144)

const double PI=acos(-1.0);

namespace fft

{

struct Complex

{

double x,y;//实部和虚部 x+yi

Complex(double \_x = 0.0,double \_y = 0.0)

{

x=\_x;

y=\_y;

}

Complex operator -(const Complex &b)const

{

return Complex(x-b.x,y-b.y);

}

Complex operator +(const Complex &b)const

{

return Complex(x+b.x,y+b.y);

}

Complex operator \*(const Complex &b)const

{

return Complex(x\*b.x-y\*b.y,x\*b.y+y\*b.x);

}

}x1[MAXN],x2[MAXN];

/\*

\* 进行FFT和IFFT前的反转变换。

\* 位置i和 （i二进制反转后位置）互换

\* len必须去2的幂

\*/

void change(Complex y[],int len)

{

int i,j,k;

for(i=1,j=len/2;i<len-1;i++)

{

if(i<j)swap(y[i],y[j]);

//交换互为小标反转的元素，i<j保证交换一次

//i做正常的+1，j左反转类型的+1,始终保持i和j是反转的

k=len/2;

while(j>=k)

{

j-=k;

k/=2;

}

if(j<k)j+=k;

}

}

/\*

\* 做FFT

\* len必须为2^k形式，

\* on==1时是DFT，on==-1时是IDFT

\*/

void fft(Complex y[],int len,int on)

{

change(y,len);

for(int h=2;h<=len;h<<=1)

{

Complex wn(cos(-on\*2\*PI/h),sin(-on\*2\*PI/h));

for(int j=0;j<len;j+=h)

{

Complex w(1,0);

for(int k=j;k<j+h/2;k++)

{

Complex u=y[k];

Complex t=w\*y[k+h/2];

y[k]=u+t;

y[k+h/2]=u-t;

w=w\*wn;

}

}

}

if(on==-1)

for(int i=0;i<len;i++)

y[i].x/=len;

}

void cal(char \*str1,char \*str2,int \*sum)

{

int len1=strlen(str1);

int len2=strlen(str2);

int len=1;

while(len<len1\*2||len<len2\*2)len<<=1;

for(int i=0;i<len1;i++)

x1[i]=Complex(str1[len1-1-i]-'0',0);

for(int i=len1;i<len;i++)

x1[i]=Complex(0,0);

for(int i=0;i<len2;i++)

x2[i]=Complex(str2[len2-1-i]-'0',0);

for(int i=len2;i<len;i++)

x2[i]=Complex(0,0);

//求DFT

fft(x1,len,1);

fft(x2,len,1);

for(int i = 0;i<len;i++)

x1[i]=x1[i]\*x2[i];

fft(x1,len,-1);

for(int i=0;i<len;i++)

sum[i]=(int)(x1[i].x+0.5);//取整误差处理

}

}

char str1[MAXN/2],str2[MAXN/2];

int sum[MAXN];

## FWT

void FWT(int a[],int n)

{

for(int d=1;d<n;d<<=1)

for(int m=d<<1,i=0;i<n;i+=m)

for(int j=0;j<d;j++)

{

int x=a[i+j],y=a[i+j+d];

a[i+j]=(x+y)%mod,a[i+j+d]=(x-y+mod)%mod;

//xor:a[i+j]=x+y,a[i+j+d]=(x-y+mod)%mod;

//and:a[i+j]=x+y;

//or:a[i+j+d]=x+y;

}

}

void UFWT(int a[],int n)

{

for(int d=1;d<n;d<<=1)

for(int m=d<<1,i=0;i<n;i+=m)

for(int j=0;j<d;j++)

{

int x=a[i+j],y=a[i+j+d];

a[i+j]=1LL\*(x+y)\*rev%mod,a[i+j+d]=(1LL\*(x-y)\*rev%mod+mod)%mod;

//xor:a[i+j]=(x+y)/2,a[i+j+d]=(x-y)/2;

//and:a[i+j]=x-y;

//or:a[i+j+d]=y-x;

}

}

void solve(int a[],int b[],int n)

{

FWT(a,n);

FWT(b,n);

for(int i=0;i<n;i++) a[i]=1LL\*a[i]\*b[i]%mod;

UFWT(a,n);

}

## NTT

namespace NTT

{

typedef long long ll;

const ll MOD=998244353;

const ll g=3;

ll qpow(ll a,ll k)

{

ll res=1LL;

while(k>0)

{

if(k&1)res=res\*a%MOD;

a=a\*a%MOD;

k>>=1;

}

return res;

}

void change(ll y[],int len)

{

for(int i=1,j=len/2;i<len-1;i++)

{

if(i<j)swap(y[i],y[j]);

int k=len/2;

while(j>=k)

{

j-=k;

k/=2;

}

if(j<k)j+=k;

}

}

void ntt(ll y[],int len,int on)

{

change(y,len);

for(int h=2;h<=len;h<<=1)

{

ll wn=qpow(g,(MOD-1)/h);

if(on==-1)wn=qpow(wn,MOD-2);

for(int j=0;j<len;j+=h)

{

ll w=1LL;

for(int k=j;k<j+h/2;k++)

{

ll u=y[k];

ll t=w\*y[k+h/2]%MOD;

y[k]=(u+t)%MOD;

y[k+h/2]=(u-t+MOD)%MOD;

w=w\*wn%MOD;

}

}

}

if(on==-1)

{

ll t=qpow(len,MOD-2);

for(int i=0;i<len;i++)

y[i]=y[i]\*t%MOD;

}

}

void cal(ll x1[],ll x2[],ll tot)

{

int len=1;

while(len<tot)len<<=1;

ntt(x1,len,1);

ntt(x2,len,1);

for(int i=0;i<len;i++)

x1[i]=x1[i]\*x2[i]%MOD;

ntt(x1,len,-1);

}

}

const LL P = 50000000001507329LL; //190734863287 \* 2 ^ 18 + 1, g = 3

const LL P = 4179340454199820289LL; // 29 \* (2 ^ 57), 4e18, g = 3

const LL P = 1945555039024054273LL; // 27 \* (2 ^ 56), 1e18, g = 5

## Lucas定理

ll qpow(ll x,ll k,ll mod)

{

int res=1;

while(k)

{

if(k&1)

res=res\*x%mod;

x=x\*x%mod;

k>>=1;

}

return res;

}

ll fact[mod+10],inv[mod+10];

void init()

{

fact[0]=1;

inv[0]=1;

for(int i=1;i<=mod;i++)

{

fact[i]=fact[i-1]\*i%mod;

inv[i]=qpow(fact[i],mod-2,mod);

}

}

ll Lucas(ll n,ll m,ll mod)

{

ll a,b,res=1LL;

while(n&&m)

{

a=n%mod,b=m%mod;

if(a<b)

return 0LL;

res=res\*fact[a]%mod\*inv[b]%mod\*inv[a-b]%mod;

n/=mod,m/=mod;

}

return res;

}

## 数位DP

int dfs(int i,int s,bool e)

{

if(i==-1)

return s==target\_s;

if(!e&&~f[i][s])

return f[i][s];

int res=0;

int u=e?num[i]:9;

for(int d=first?1:0;d<=u;++d)

res+=dfs(i-1,new\_s(s,d),e&&d==u);

return e?res:f[i][s]=res;

}

f为记忆化数组；

i为当前处理串的第i位（权重表示法，也即后面剩下i+1位待填数）；

s为之前数字的状态（如果要求后面的数满足什么状态，也可以再记一个目标状态t之类，for的时候枚举下t）；

e表示之前的数是否是上界的前缀（即后面的数能否任意填）。

## 默慈金数

#include<cstdio>

#include<iostream>

#include<algorithm>

using namespace std;

const long long int mod=1e9+7,maxn=1e6+5;

long long int f[maxn]={0,1},mo[maxn]={0};

int main()

{

int i,T,n;

for(i=2;i<=1000000;i++)

{

f[i]=(((mod-mod/i)%mod)\*(f[mod%i]%mod))%mod;

}

mo[0]=1,mo[1]=1;

for(i=2;i<=1000000;i++)

{

mo[i]=(((((2\*i+1)\*f[i+2])%mod)\*mo[i-1])%mod+(((3\*(i-1)\*f[i+2])%mod)\*mo[i-2])%mod)%mod;

}

}

## 错排

void init(int n)

{

dp[0]=1;dp[1]=0;dp[2]=1;

for(int i=3;i<MAXN;i++)

{

dp[i]=(i-1)\*(dp[i-1]+dp[i-2]);

}

}

## 单纯形

//-std=c99 or g++

//输入 标准型 线性规划，输出目标函数最优解

//输入格式:

/\*

第一行两个整数n, m，表示n个自变量，m个约束

第二行n个正数，表示目标函数自变量前的系数

接下来m行，每行n+1个整数

前n个正数描述了这个约束的自变量前的系数，最后一个正数描述了这个约束 <=号 右边的数值

例:

输入(讲义里面讲单纯形法时用的例子):

3 3

3 1 2

1 1 3 30

2 2 5 24

4 1 2 36

输出:

28.00000000

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdarg.h>

#include <math.h>

#include <ctype.h>

#define MAXN 50

#define MAXM 50

//MAXN是变量数的上限，MAXM是约束条件数的上限

const double eps = 1e-9;

int dcmp(double x)

{

if(x < -eps)return -1;

else if(x > eps)return 1;

else return 0;

}

void swap(int \*a, int \*b)

{

int c = \*a;

\*a = \*b;

\*b = c;

}

int n, m; //n个变量，m个约束条件

double A[MAXN][MAXM];

int base[MAXN+MAXM], unbase[MAXM+MAXN]; //基本变量与非基本变量

void pivot(int x, int y) //转动

{

swap(base+x, unbase+y);

double k = A[x][y];

A[x][y] = 1.0; //前面系数直接变成1.0

for(int i = 0; i <= n; i++)A[x][i] /= k; //这一行都约掉k

for(int i = 0; i <= m; i++) //替换

{

if(i != x && dcmp(k = A[i][y]) /\* \*/)

{

A[i][0] += (i?-1:1)\*k\*A[x][0];

A[i][y] = 0;

for(int j = 1; j <= n; j++)

A[i][j] -= k\*A[x][j];

}

}

}

int init\_simplex()

{

for(int i = 1; i <= n; i++)unbase[i] = i; //最开始的n个非基本变量

for(int i = 1; i <= m; i++)base[i] = n+i; //额外的m个松弛变量

for(int x = 0, y = 0; ; x = y = 0)

{

for(int i = 1; i <= m; i++)

if(dcmp(A[i][0]) < 0)x = i;

if(!x)return 1;

for(int i = 1; i <= n; i++)

if(dcmp(A[x][i]) < 0)y = i;

if(!y)return 0; //发现存在b(i) < 0而变量前系数 都大于等于0(注意是 全都 大于等于0)

//由于变量具有非负约束，此时整个约束系统无解。

pivot(x, y);

}

}

int simplex()

{

if(!init\_simplex())return 0;

for(int x = 0, y = 0; ; x = y = 0)

{

for(int i = 1; i <= n; i++)

if(dcmp(A[0][i]) > 0) //找到一个目标函数里前面系数为正数的一个变量

{ // \_\_\_\_\_\_\_/

y = i; //

break; //

} //

if(!y)return 1; //若找不到说明已经找到最优解，返回

double inf = 1e15;

for(int i = 1; i <= m; i++) //找到对y约束最紧的变量

{

double t = A[i][0]/A[i][y];

if(dcmp(A[i][y]) > 0 && (!x || t < inf))

{

inf = t;

x = i;

}

}

if(!x)return -1; //无法约束y,此时整个目标函数发散

pivot(x, y);

}

}

void test()

{

freopen("test\_data.txt", "r", stdin); //这个文件里面就是讲义里面的例子。

scanf("%d %d", &n, &m);

for(int i = 1; i <= n; i++)

scanf("%lf", A[0]+i); //A[0]存目标函数，因此最后A[0][0]为目标函数最大值

for(int i = 1; i <= m; scanf("%lf", A[i++])) //b(i)存到A[i][0]里

for(int j = 1; j <= n; j++)scanf("%lf", A[i]+j);

switch(simplex())

{

case 1:

printf("%.8lf\n", A[0][0]);

break;

case 0:

puts("No solution");

break;

case -1:

puts("Infinity");

break;

}

}

int main()

{

test();

return 0;

}

## Meisell-Lehmer算法（1e11内素数）

#define MAXN 100

#define MAXM 100010

#define MAXP 666666

#define MAX 10000010

#define clr(ar) memset(ar, 0, sizeof(ar))

#define read() freopen("lol.txt", "r", stdin)

#define dbg(x) cout << #x << " = " << x << endl

#define chkbit(ar, i) (((ar[(i) >> 6]) & (1 << (((i) >> 1) & 31))))

#define setbit(ar, i) (((ar[(i) >> 6]) |= (1 << (((i) >> 1) & 31))))

#define isprime(x) (( (x) && ((x)&1) && (!chkbit(ar, (x)))) || ((x) == 2))

using namespace std;

namespace pcf{

long long dp[MAXN][MAXM];

unsigned int ar[(MAX >> 6) + 5] = {0};

int len = 0, primes[MAXP], counter[MAX];

void Sieve(){

setbit(ar, 0), setbit(ar, 1);

for (int i = 3; (i \* i) < MAX; i++, i++){

if (!chkbit(ar, i)){

int k = i << 1;

for (int j = (i \* i); j < MAX; j += k) setbit(ar, j);

}

}

for (int i = 1; i < MAX; i++){

counter[i] = counter[i - 1];

if (isprime(i)) primes[len++] = i, counter[i]++;

}

}

void init(){

Sieve();

for (int n = 0; n < MAXN; n++){

for (int m = 0; m < MAXM; m++){

if (!n) dp[n][m] = m;

else dp[n][m] = dp[n - 1][m] - dp[n - 1][m / primes[n - 1]];

}

}

}

long long phi(long long m, int n){

if (n == 0) return m;

if (primes[n - 1] >= m) return 1;

if (m < MAXM && n < MAXN) return dp[n][m];

return phi(m, n - 1) - phi(m / primes[n - 1], n - 1);

}

long long Lehmer(long long m){

if (m < MAX) return counter[m];

long long w, res = 0;

int i, a, s, c, x, y;

s = sqrt(0.9 + m), y = c = cbrt(0.9 + m);

a = counter[y], res = phi(m, a) + a - 1;

for (i = a; primes[i] <= s; i++) res = res - Lehmer(m / primes[i]) + Lehmer(primes[i]) - 1;

return res;

}

}

# 计算几何

## 海伦公式

//三角形面积公式

s=(a+b+c)/ 2;

area=sqrt(s\*(s-a)\*(s-b)\*(s-c));

## 正n面体体积

typedef double db;

const db PI=acos(-1.0);

db get(db l,int n)//边长,正n面体

{

db r=l/2/sin(PI/n);

db h=sqrt(l\*l-r\*r);

return n\*r\*r\*sin(2\*PI/n)/2\*h/3;

}

## Point（三维）

struct Point

{

double x,y,z;

Point(){}

Point(double \_x,double \_y,double \_z)

{

x=\_x;y=\_y;z=\_z;

}

int input()

{

return scanf("%lf%lf%lf",&x,&y,&z);

}

Point operator - (const Point b)const

{

return Point((x-b.x),(y-b.y),(z-b.z));

}

double operator % (const Point b)const//两点距离

{

return sqrt((x-b.x)\*(x-b.x)+(y-b.y)\*(y-b.y)+(z-b.z)\*(z-b.z));

}

double operator \* (const Point b)const//内积

{

return (x\*b.x)+(y\*b.y)+(z\*b.z);

}

Point operator ^ (const Point b)const//外积

{

return Point((y\*b.z-z\*b.y),(z\*b.x-x\*b.z),(x\*b.y-y\*b.x));

}

}a,b,c,d;

## Point

//1.1 Point定义

const double eps=1e-8;

const double PI=acos(-1.0);

int sgn(double x)

{

if(fabs(x)<eps)return 0;

if(x < 0)return -1;

else return 1;

}

struct Point

{

double x,y;

Point(){}

Point(double \_x,double \_y)

{

x=\_x;y=\_y;

}

Point operator - (const Point &b)const

{

return Point(x-b.x,y-b.y);

}

double operator ^ (const Point &b)const//叉积

{

return x\*b.y-y\*b.x;

}

double operator \* (const Point &b)const//点积

{

return x\*b.x+y\*b.y;

}

void transXY(double B)//绕原点旋转角度B（弧度值），后x,y的变化

{

double tx=x,ty=y;

x=tx\*cos(B)-ty\*sin(B);

y=tx\*sin(B)+ty\*cos(B);

}

};

## Line

//1.2 Line定义

struct Line

{

Point s,e;

Line(){}

Line(Point \_s,Point \_e)

{

s = \_s;e = \_e;

}

//两直线相交求交点

//第一个值为0表示直线重合，为1表示平行, 为2是相交

//只有第一个值为2时，交点才有意义

pair<int,Point> operator &(const Line &b)const

{

Point res=s;

if(sgn((s-e)^(b.s-b.e)) == 0)

{

if(sgn((s-b.e)^(b.s-b.e)) == 0)

return make\_pair(0,res);//重合

else return make\_pair(1,res);//平行

}

double t=((s-b.s)^(b.s-b.e))/((s-e)^(b.s-b.e));

res.x+=(e.x-s.x)\*t;

res.y+=(e.y-s.y)\*t;

return make\_pair(2,res);

}

};

## 判断线段相交

//1.4 判断：线段相交

bool inter(Line l1,Line l2)

{

return

max(l1.s.x,l1.e.x)>=min(l2.s.x,l2.e.x) &&

max(l2.s.x,l2.e.x)>=min(l1.s.x,l1.e.x) &&

max(l1.s.y,l1.e.y)>=min(l2.s.y,l2.e.y) &&

max(l2.s.y,l2.e.y)>=min(l1.s.y,l1.e.y) &&

sgn((l2.s-l1.e)^(l1.s-l1.e))\*sgn((l2.e-l1.e)^(l1.s-l1.e)) <= 0 &&

sgn((l1.s-l2.e)^(l2.s-l2.e))\*sgn((l1.e-l2.e)^(l2.s-l2.e)) <= 0;

}

## 判断直线和线段相交

//1.5 判断：直线和线段相交

bool Seg\_inter\_line(Line l1,Line l2) //判断直线l1和线段l2是否相交

{

return sgn((l2.s-l1.e)^(l1.s-l1.e))\*sgn((l2.e-l1.e)^(l1.s-l1.e))<=0;

}

## 点到直线距离

//1.6 点到直线距离

//点到直线距离

//返回为result,是点到直线最近的点

Point PointToLine(Point P,Line L)

{

Point result;

double t=((P-L.s)\*(L.e-L.s))/((L.e-L.s)\*(L.e-L.s));

result.x=L.s.x+(L.e.x-L.s.x)\*t;

result.y=L.s.y+(L.e.y-L.s.y)\*t;

return result;

}

## 点到线段距离

//1.7 点到线段距离

//点到线段的距离

//返回点到线段最近的点

Point NearestPointToLineSeg(Point P,Line L)

{

Point result;

double t=((P-L.s)\*(L.e-L.s))/((L.e-L.s)\*(L.e-L.s));

if(t>=0&&t<=1)

{

result.x=L.s.x+(L.e.x-L.s.x)\*t;

result.y=L.s.y+(L.e.y-L.s.y)\*t;

}

else

{

if(dist(P,L.s)<dist(P,L.e))

result=L.s;

else result=L.e;

}

return result;

}

## 多边形面积

//1.8 计算多边形面积

//计算多边形面积

//点的编号从0~n-1

double CalcArea(Point p[],int n)

{

double res=0;

for(int i=0;i<n;i++)

res+=(p[i]^p[(i+1)%n])/2;

return fabs(res);

}

## 判断点在线段上

//1.9 判断点在线段上

//\*判断点在线段上

bool OnSeg(Point P,Line L)

{

return

sgn((L.s-P)^(L.e-P))==0 &&

sgn((P.x-L.s.x)\*(P.x-L.e.x))<=0 &&

sgn((P.y-L.s.y)\*(P.y-L.e.y))<=0;

}

## 判断点在凸多边形内

//1.10 判断点在凸多边形内

//\*判断点在凸多边形内

//点形成一个凸包，而且按逆时针排序（如果是顺时针把里面的<0改为>0）

//点的编号:0~n-1

//返回值：

//-1:点在凸多边形外

//0:点在凸多边形边界上

//1:点在凸多边形内

int inConvexPoly(Point a,Point p[],int n)

{

for(int i=0;i<n;i++)

{

if(sgn((p[i]-a)^(p[(i+1)%n]-a)) < 0)return -1;

else if(OnSeg(a,Line(p[i],p[(i+1)%n])))return 0;

}

return 1;

}

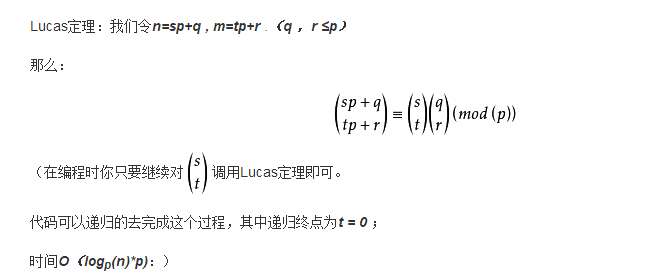
# 数论公式

## 欧拉定理

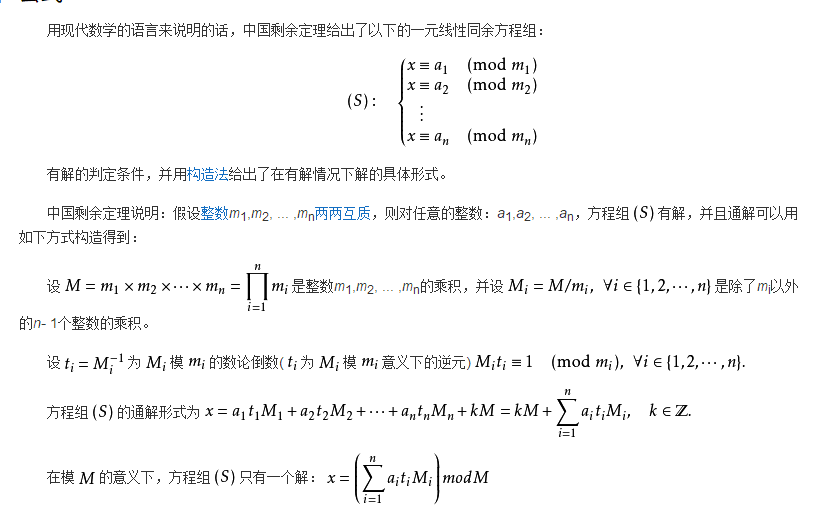
在数论中，欧拉定理,（也称费马-欧拉定理）是一个关于同余的性质。欧拉定理表明，若n,a为正整数，且n,a互质，则:

http://c.hiphotos.baidu.com/baike/s%3D116/sign=b4a8aa2a7dd98d1072d40830173eb807/0823dd54564e92584b7cba389d82d158cdbf4e9f.jpg

## Lucas定理



## 中国剩余定理



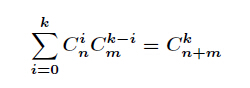
## 费马小定理

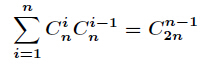
**费马小定理(Fermat Theory)**是[数论](http://baike.baidu.com/view/17568.htm)中的一个重要[定理](http://baike.baidu.com/subview/434662/11140650.htm)，其内容为： 假如p是[质数](http://baike.baidu.com/view/10626.htm)，且gcd(a,p)=1，那么 a(p-1)≡1（mod p）。即：假如a是[整数](http://baike.baidu.com/view/71484.htm)，p是[质数](http://baike.baidu.com/view/10626.htm)，且a,p[互质](http://baike.baidu.com/view/731400.htm)(即两者只有一个[公约数](http://baike.baidu.com/view/58629.htm)1)，那么a的(p-1)次方除以p的[余数](http://baike.baidu.com/view/1068391.htm)[恒等](http://baike.baidu.com/view/244913.htm)于1。

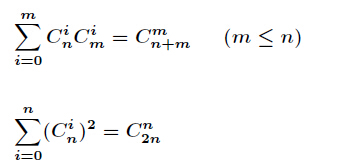
## 拉姆塞理论

拉姆塞理论可以用通常的语言来表述。在一个集会上，两个人或者彼此认识，或者彼此不认识，拉姆塞得出结果是说，当集会人数大于或等于6时，则必定有3个人，他们或者彼此者认识或者彼此都不认识。

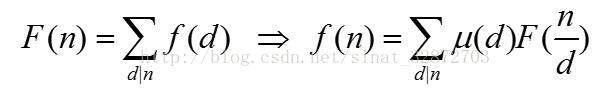
## 范德蒙恒等式

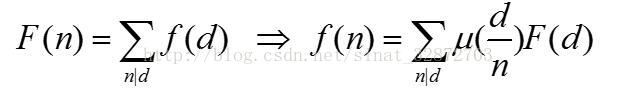


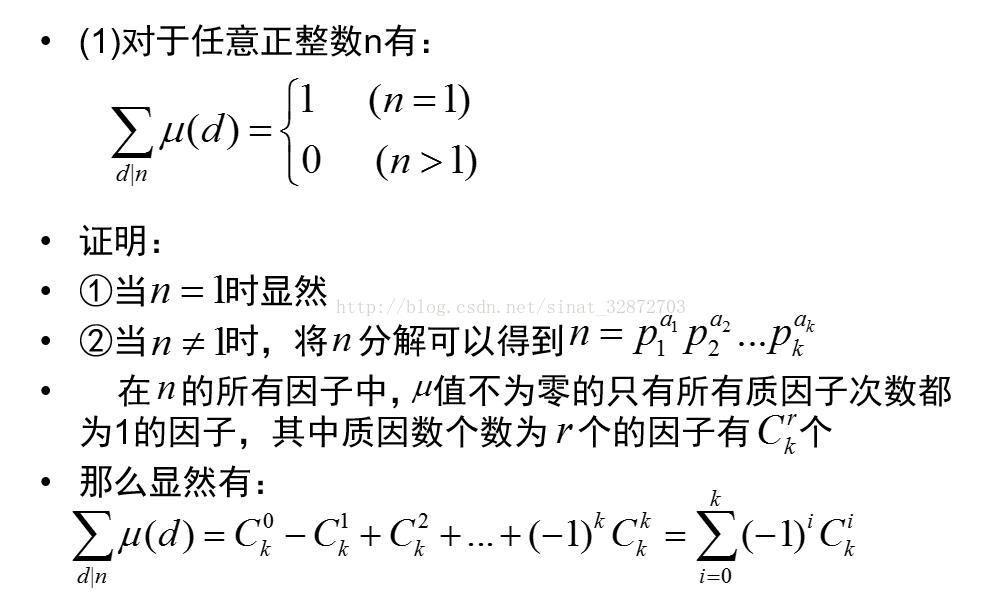


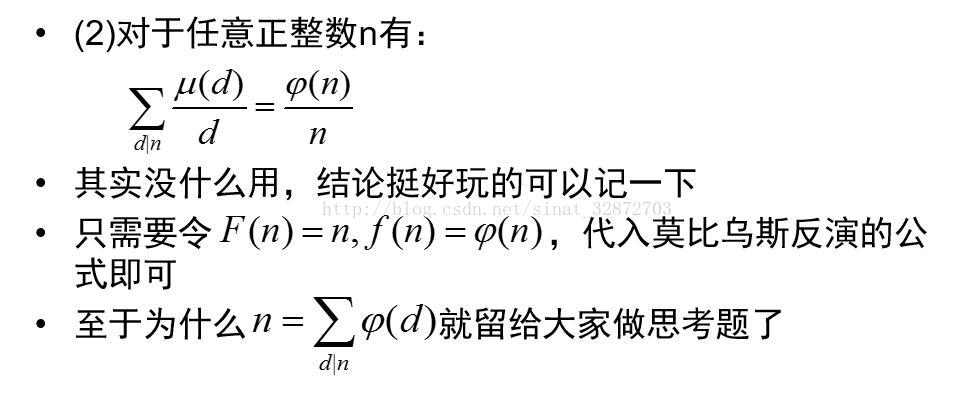


## 莫比乌斯反演

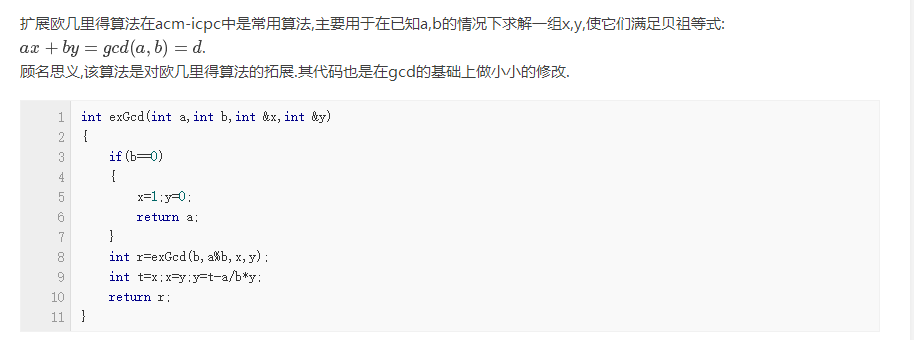


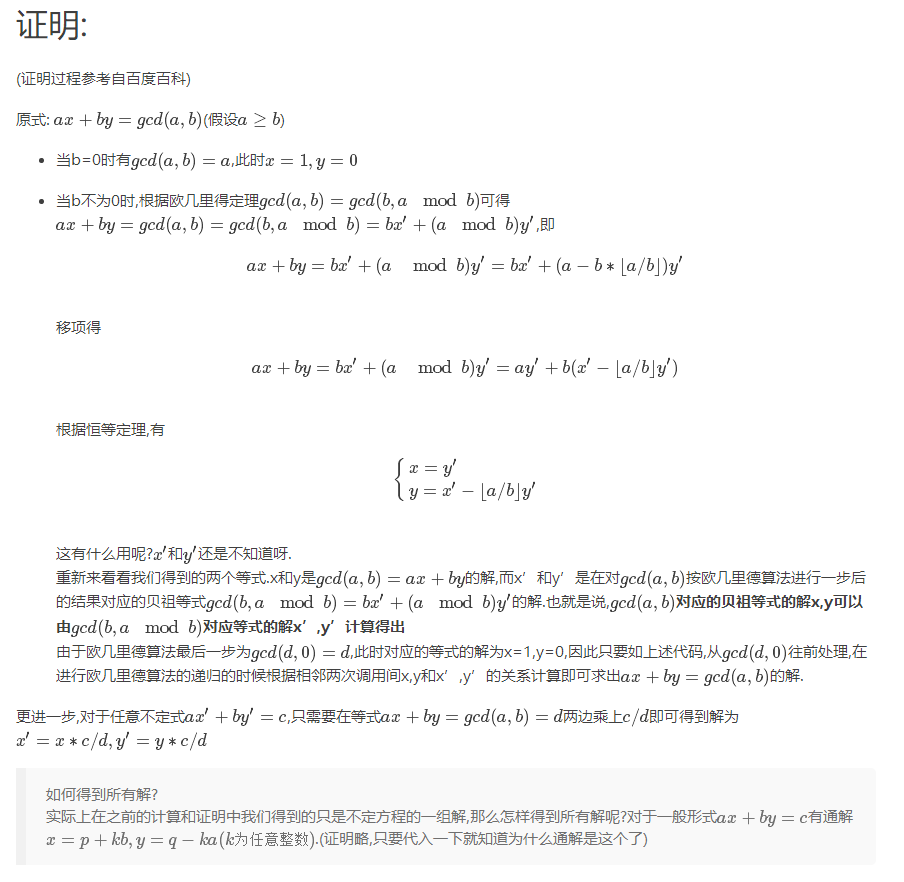




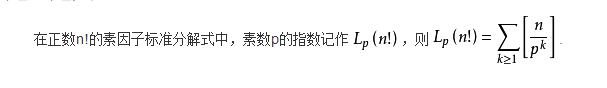


## 扩展欧几里德





## 勒让德定理



## 二分图

一、二分图最大匹配

定义：匹配是图中一些边的集合，且集合中任意两条边都没有公共点，所有的匹配中，边数最多的就是最大匹配。

算法：用匈牙利算法可以在O(V\*E)的复杂度内求出二分图的最大匹配，具体可以看byvoid神犇的blog，讲的很详细，不过想真正完全证明这个算法，得去看组合数学。

二、二分图最小点覆盖

定义：点覆盖是图中一些点的集合，且对于图中所有的边，至少有一个端点属于点覆盖，点数最小的覆盖就是最小点覆盖。

定理：最小点覆盖=最大匹配。

简单证明：首先必然有，最小覆盖>=最大匹配。于是只要证明不等式可以取等号，我们可在最大匹配的基础上构造出一组点覆盖。对右边每一个未匹配的点进行dfs找增广路，标记所有dfs过程中访问到的点，左边标记的点+右边未标记的点就是这个图的一个点覆盖。因为对于任意一条边，如果他的左边没标记，右边被标记了，那么我们就可找到一条新的增广路，所以每一条边都至少被一个点覆盖。再来证明：最大匹配=左边标记的点+右边未标记的点。对于每条匹配边，只有一个点属于点覆盖。如果这条边在dfs过程中被访问了，那么就左端点属于点覆盖，右端点不属于，否则就有左端点不属于点覆盖，右端点属于点覆盖。除此之外，不可能存在其它的点属于最小覆盖了，不然就必然可以找到增广路。所以：左边标记的点+右边未标记的点=最大匹配，对于任意的二分图，我们总能在最大匹配的基础上构造出一组点数等于最大匹配的点覆盖，所以：最小点覆盖=最大匹配。

三、二分图最小边覆盖

定义：边覆盖是图中一些边的集合，且对于图中所有的点，至少有一条集合中的边与其相关联，边数最小的覆盖就是最小边覆盖。

定理：最小边覆盖=图中点的个数-最大匹配。

简单证明：先贪心的选一组最大匹配的边加入集合，对于剩下的每个未匹配的点，随便选一条与之关联的边加入集合，得到的集合就是最小边覆盖，所以有：最小边覆盖=最大匹配+图中点的个数-2\*最大匹配=图中点的个数-最大匹配。

四、二分图最大独立集

定义：独立集是图中一些点的集合，且图中任意两点之间都不存在边，点数最大的就是最大独立集。

定理：最大独立集=图中点的个数-最大匹配。

简单证明：可以这样来理解，先把所有的点都加入集合，删除最少的点和与其关联的边使得剩下的点相互之间不存在边，我们就得到了最大独立集。所以有：最大独立集=图中点的个数-最小点覆盖=图中点的个数-最大匹配。

五、有向无环图最小不相交路径覆盖

定义：用最少的不相交路径覆盖所有顶点。

定理：把原图中的每个点V拆成Vx和Vy，如果有一条有向边A->B，那么就加边Ax-By。这样就得到了一个二分图，最小路径覆盖=原图的节点数-新图最大匹配。

简单证明：一开始每个点都独立的为一条路径，总共有n条不相交路径。我们每次在二分图里加一条边就相当于把两条路径合成了一条路径，因为路径之间不能有公共点，所以加的边之间也不能有公共点，这就是匹配的定义。所以有：最小路径覆盖=原图的节点数-新图最大匹配。

六、有向无环图最小可相交路径覆盖

定义：用最小的可相交路径覆盖所有顶点。

算法：先用floyd求出原图的传递闭包，即如果a到b有路，那么就加边a->b。然后就转化成了最小不相交路径覆盖问题。

七、偏序集的最大反链

定义：偏序集中的最大独立集。

Dilworth定理：对于任意偏序集都有，最大独立集（最大反链）=最小链的划分（最小不相交路径覆盖）。

通过Dilworth定理，我们就可以把偏序集的最大独立集问题转化为最小不相交路径覆盖问题了。

八、二分图带权最大匹配

定义：每个边都有一组权值，边权之和最大的匹配就是带权最大匹配。

算法：KM算法，复杂度为O（V^3）。具体就不说了，网上有不少资料。

要注意的是，KM算法求的是最佳匹配，即在匹配是完备的基础上权值之和最大。这和带权最大匹配是不一样的，不过我们可以加入若干条边权为0的边使得KM求出来的最佳匹配等于最大权匹配。具体实现的时候最好用矩阵来存图，因为一般点的个数都是10^2级别，并且这样默认任意两点之间都存在边权为0的边，写起来很方便。如果要求最小权匹配，我们可以用一个很大数减去每条边的边权。

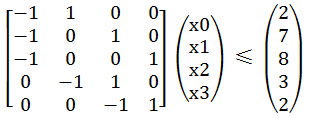
## 差分约束

**一、差分约束**

**1、数形结合**

      介绍完最短路，回到之前提到的那个不等式组的问题上来，我们将它更加系统化。

      如若一个系统由n个变量和m个不等式组成，并且这m个不等式对应的系数矩阵中每一行有且仅有一个1和-1，其它的都为0，这样的系统称为**差分约束( difference constraints )**系统。引例中的不等式组可以表示成如图三-1-1的系数矩阵。



图三-1-1

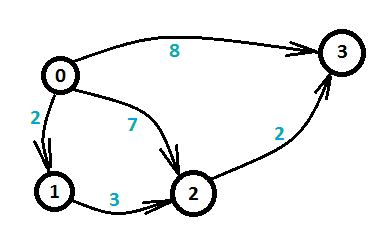
      然后继续回到单个不等式上来，观察 x[i] - x[j] <= a[k]， 将这个不等式稍稍变形，将x[j]移到不等式右边，则有x[i] <= x[j] + a[k]，然后我们令a[k] = w(j, i)，再将不等式中的i和j变量替换掉，i = v， j = u，将x数组的名字改成d（以上都是等价变换，不会改变原有不等式的性质），则原先的不等式变成了以下形式：d[u] + w(u, v) >= d[v]。

      这时候联想到SPFA中的一个松弛操作：

    if(d[u] + w(u, v) < d[v]) {  
        d[v] = d[u] + w(u, v);  
    }

      对比上面的不等式，两个不等式的不等号正好相反，但是再仔细一想，其实它们的逻辑是一致的，因为SPFA的松弛操作是在满足小于的情况下进行松弛，力求达到d[u] + w(u, v) >= d[v]，而我们之前令a[k] = w(j, i)，所以我们可以将每个不等式转化成图上的有向边：

**对于每个不等式 x[i] - x[j] <= a[k]，对结点 j 和 i 建立一条 j -> i的有向边，边权为a[k]，求x[n-1] - x[0] 的最大值就是求 0 到n-1的最短路。**



图三-1-2

      图三-1-2 展示了 图三-1-1的不等式组转化后的图。

**2、三角不等式**

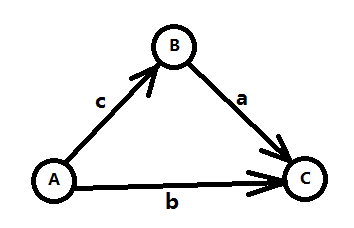
      如果还没有完全理解，我们可以先来看一个简单的情况，如下三个不等式：

B - A <= c      (1)

C - B <= a      (2)

C - A <= b      (3)

      我们想要知道C - A的最大值，通过(1) + (2)，可以得到 C - A <= a + c，所以这个问题其实就是求min{b, a+c}。将上面的三个不等式按照 **三-1 数形结合** 中提到的方式建图，如图三-2-1所示。

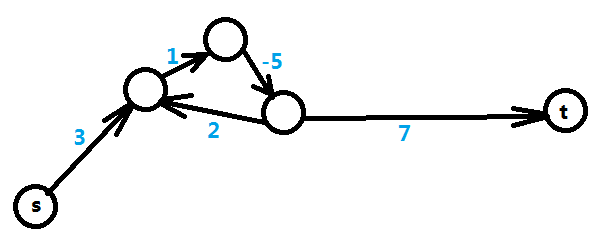


图三-2-1

      我们发现min{b, a+c}正好对应了A到C的最短路，而这三个不等式就是著名的**三角不等式**。将三个不等式推广到m个，变量推广到n个，就变成了n个点m条边的最短路问题了。

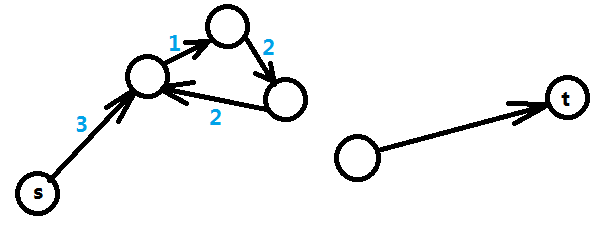
**3、解的存在性**

      上文提到最短路的时候，会出现负权圈或者根本就不可达的情况，所以在不等式组转化的图上也有可能出现上述情况，先来看负权圈的情况，如图三-3-1，下图为5个变量5个不等式转化后的图，需要求得是X[t] - X[s]的最大值，可以转化成求s到t的最短路，但是路径中出现负权圈，则表示最短路无限小，即不存在最短路，那么在不等式上的表现即X[t] - X[s] <= T中的T无限小，得出的结论就是 X[t] - X[s]的最大值 不存在。



图三-3-1

      再来看另一种情况，即从起点s无法到达t的情况，如图三-3-2，表明X[t]和X[s]之间并没有约束关系，这种情况下X[t] - X[s]的最大值是无限大，这就表明了X[t]和X[s]的取值有无限多种。



图三-3-2

      在实际问题中这两种情况会让你给出不同的输出。综上所述，差分约束系统的解有三种情况：1、有解；2、无解；3、无限多解；

**4、最大值 => 最小值**

      然后，我们将问题进行一个简单的转化，将原先的"<="变成">="，转化后的不等式如下：

B - A >= c      (1)

C - B >= a      (2)

C - A >= b      (3)

      然后求C - A的最小值，类比之前的方法，需要求的其实是max{b, c+a}，于是对应的是图三-2-1从A到C的最长路。同样可以推广到n个变量m个不等式的情况。

**5、不等式标准化**

如果给出的不等式有"<="也有">="，又该如何解决呢？很明显，首先需要关注最后的问题是什么，如果需要求的是两个变量差的最大值，那么需要将所有不等式转变成"<="的形式，建图后求最短路；相反，如果需要求的是两个变量差的最小值，那么需要将所有不等式转化成">="，建图后求最长路。

      如果有形如：A - B = c 这样的等式呢？我们可以将它转化成以下两个不等式：

A - B >= c      (1)

A - B <= c      (2)

       再通过上面的方法将其中一种不等号反向，建图即可。

       最后，如果这些变量都是整数域上的，那么遇到A - B < c这样的不带等号的不等式，我们需要将它转化成"<="或者">="的形式，即 A - B <= c - 1。

**二、差分约束的经典应用**

**1、线性约束**

        线性约束一般是在一维空间中给出一些变量（一般定义位置），然后告诉你某两个变量的约束关系，求两个变量a和b的差值的最大值或最小值。

**【例题1】N个人编号为1-N，并且按照编号顺序排成一条直线，任何两个人的位置不重合，然后给定一些约束条件。**

**X(X <= 100000)组约束Ax Bx Cx(1 <= Ax < Bx <= N)，表示Ax和Bx的距离不能大于Cx。**

**Y(X <= 100000)组约束Ay By Cy(1 <= Ay < By <= N)，表示Ay和By的距离不能小于Cy。**

**如果这样的排列存在，输出1-N这两个人的最长可能距离，如果不存在，输出-1，如果无限长输出-2。**

      像这类问题，N个人的位置在一条直线上呈线性排列，某两个人的位置满足某些约束条件，最后要求第一个人和最后一个人的最长可能距离，这种是最直白的差分约束问题，因为可以用距离作为变量列出不等式组，然后再转化成图求最短路。

      令第x个人的位置为d[x]（不妨设d[x]为x的递增函数，即随着x的增大，d[x]的位置朝着x正方向延伸）。

      那么我们可以列出一些约束条件如下：

      1、对于所有的Ax Bx Cx，有 d[Bx] - d[Ax] <= Cx；

      2、对于所有的Ay By Cy，有 d[By] - d[Ay] >= Cy；

      3、然后根据我们的设定，有 d[x] >= d[x-1] + 1 (1 < x <= N)  （这个条件是表示任何两个人的位置不重合）

**而我们需要求的是d[N] - d[1]的最大值，即表示成d[N] - d[1] <= T，要求的就是这个T。**

     于是我们将所有的不等式都转化成d[x] - d[y] <= z的形式，如下：

      1、d[Bx]  -  d[Ax]    <=    Cx

      2、d[Ay]  -  d[By]    <=  -Cy

      3、d[x-1] -    d[x]    <=    -1

     对于d[x] - d[y] <= z，令z = w(y, x)，那么有 d[x] <= d[y] + w(y, x)，所以当d[x] > d[y] + w(y, x)，我们需要更新d[x]的值，这对应了最短路的松弛操作，于是问题转化成了求1到N的最短路。

**对于所有满足d[x] - d[y] <= z的不等式，从y向x建立一条权值为z的有向边。**

      然后从起点1出发，利用SPFA求到各个点的最短路，如果1到N不可达，说明最短路(即上文中的T)无限长，输出-2。如果某个点进入队列大于等于N次，则必定存在一条负环，即没有最短路，输出-1。否则T就等于1到N的最短路。

     **2、区间约束**

**【例题2】给定n（n <= 50000）个整点闭区间和这个区间中至少有多少整点需要被选中，每个区间的范围为[ai, bi]，并且至少有ci个点需要被选中，其中0 <= ai <= bi <= 50000，问[0, 50000]至少需要有多少点被选中。**

**例如3 6 2 表示[3, 6]这个区间至少需要选择2个点，可以是3,4也可以是4,6（总情况有 C(4, 2)种 ）。**

      这类问题就没有线性约束那么明显，需要将问题进行一下转化，考虑到最后需要求的是一个完整区间内至少有多少点被选中，试着用d[i]表示[0, i]这个区间至少有多少点能被选中，根据定义，可以抽象出 d[-1] = 0，对于每个区间描述，可以表示成d[ bi ]  - d[ ai - 1 ] >= ci，而我们的目标要求的是 d[ 50000 ] - d[ -1 ] >= T 这个不等式中的T，将所有区间描述转化成图后求-1到50000的最长路。

      这里忽略了一些要素，因为d[i]描述了一个求和函数，所以对于d[i]和d[i-1]其实是有自身限制的，考虑到每个点有选和不选两种状态，所以d[i]和d[i-1]需要满足以下不等式：  0 <= d[i] - d[i-1] <= 1   （即第i个数选还是不选）

      这样一来，还需要加入 50000\*2 = 100000 条边，由于边数和点数都是万级别的，所以不能采用单纯的Bellman-Ford ，需要利用SPFA进行优化，由于-1不能映射到小标，所以可以将所有点都向x轴正方向偏移1个单位（即所有数+1）。

**3、未知条件约束**

      未知条件约束是指在不等式的右边不一定是个常数，可能是个未知数，可以通过枚举这个未知数，然后对不等式转化成差分约束进行求解。

**【例题3】**

**在一家超市里，每个时刻都需要有营业员看管，R(i)  (0 <= i < 24)表示从i时刻开始到i+1时刻结束需要的营业员的数目，现在有N(N <= 1000)个申请人申请这项工作，并且每个申请者都有一个起始工作时间 ti，如果第i个申请者被录用，那么他会连续工作8小时。**

**现在要求选择一些申请者进行录用，使得任何一个时刻i，营业员数目都能大于等于R(i)。**

       i = 0 1 2 3 4 5 6 ... 20 21 22 23 23，分别对应时刻 [i, i+1)，特殊的，23表示的是[23, 0)，并且有些申请者的工作时间可能会“跨天”。

       a[i] 表示在第i时刻**开始工作**的人数，是个**未知量**

       b[i] 表示在第i时刻能够**开始工作**人数的上限， 是个**已知量**

       R[i] 表示在第i时刻必须值班的人数，也是**已知量**

       那么第i时刻到第i+1时刻还在工作的人满足下面两个不等式（利用每人工作时间8小时这个条件）：

       当 i >= 7,        a[i-7] + a[i-6] + ... + a[i] >= R[i]                                     (1)

       当 0 <= i < 7,  (a[0] + ... + a[i]) + (a[i+17] + ... + a[23]) >= R[i]              (2)

       对于从第i时刻开始工作的人，满足以下不等式：

       0<= i < 24,    0 <= a[i] <= b[i]                                                            (3)

       令 s[i] = a[0] + ... + a[i]，特殊地，s[-1] = 0

       上面三个式子用s[i]来表示，如下：

       s[i]-s[i-8]>=R[i]                           (i >= 7)                                      (1)

       s[i] + s[23] - s[i+16] >= R[i]               (0 <= i < 7)                                  (2)

       0 <= s[i] - s[i-1] <= b[i]                     (0 <= i < 24)                                (3)

      仔细观察不等式(2)，有三个未知数，这里的s[23]就是**未知条件**，所以还无法转化成差分约束求解，但是和i相关的变量只有两个，对于s[23]的值我们可以进行枚举，令s[23] = T, 则有以下几个不等式：

      s[i] - s[i-8] >= R[i]

      s[i] - s[i+16] >= R[i] - T

      s[i] - s[i-1] >= 0

      s[i-1] - s[i] >= -b[i]

      对于所有的不等式 s[y] - s[x] >= c，建立一条权值为c的边 x->y，于是问题转化成了求从原点-1到终点23的最长路。

      但是这个问题比较特殊，我们还少了一个条件，即：s[23] = T，它并不是一个不等式，我们需要将它也转化成不等式，由于设定s[-1] = 0，所以 s[23] - s[-1] = T，它可以转化成两个不等式：

      s[23] - s[-1] >= T

      s[-1] - s[23] >= -T

      将这两条边补到原图中，求出的最长路s[23]等于T，表示T就是满足条件的一个解，由于T的值时从小到大枚举的（T的范围为0到N），所以第一个满足条件的解就是答案。

      最后，观察申请者的数量，当i个申请者能够满足条件的时候，i+1个申请者必定可以满足条件，所以申请者的数量是满足单调性的，可以对T进行二分枚举，将枚举复杂度从O(N)降为O(logN)。

# 其他

## Vim

syntax on

set nu

set history=1000000

set tabstop=4

set shiftwidth=4

set smarttab

set cindent

colo torte

set backspace=indent,eol,start

set nobackup

set noswapfile

set mouse=a

map <F9> : call CR()<CR>

func! CR()

exec "w"

exec "!g++ % -o %<"

endfunc

imap <c-]> {<cr>}<c-o><up><cr>

map <F2> :call SetTitle()<CR>

func SetTitle()

let l=0

let l=l+1 | call setline(l,'#include<bits/stdc++.h>')

let l=l+1 | call setline(l,'using namespace std;')

let l=l+1 | call setline(l,'')

let l=l+1 | call setline(l,'int main()')

let l=l+1 | call setline(l,'{')

let l=l+1 | call setline(l,' //freopen("in.txt","r",stdin);')

let l=l+1 | call setline(l,' //freopen("out.txt","w",stdout);')

let l=l+1 | call setline(l,'')

let l=l+1 | call setline(l,' return 0;')

let l=l+1 | call setline(l,'}')

endfunc

## JAVA大数

在Java中有两个类BigInteger和BigDecimal分别表示大整数类和大浮点数类，至于两个类的对象能表示最大范围不清楚，理论上能够表示无线大的数，只要计算机内存足够大。   
这两个类都在java.math.\*包中，因此每次必须在开头处引用该包。

Ⅰ基本函数：   
1.valueOf(parament); 将参数转换为制定的类型   
比如 int a=3;   
BigInteger b=BigInteger.valueOf(a);   
则b=3;   
String s=”12345”;   
BigInteger c=BigInteger.valueOf(s);   
则c=12345；

2.add(); 大整数相加   
BigInteger a=new BigInteger(“23”);   
BigInteger b=new BigInteger(“34”);   
a. add(b);

3.subtract(); 相减   
4.multiply(); 相乘   
5.divide(); 相除取整   
6.remainder(); 取余   
7.pow(); a.pow(b)=a^b   
8.gcd(); 最大公约数   
9.abs(); 绝对值   
10.negate(); 取反数   
11.mod(); a.mod(b)=a%b=a.remainder(b);   
12.max(); min();   
13.punlic int comareTo();   
14.boolean equals(); 是否相等   
15.BigInteger构造函数：   
一般用到以下两种：   
BigInteger(String val);   
将指定字符串转换为十进制表示形式；   
BigInteger(String val,int radix);   
将指定基数的 BigInteger 的字符串表示形式转换为 BigInteger   
Ⅱ.基本常量：   
A=BigInteger.ONE 1   
B=BigInteger.TEN 10   
C=BigInteger.ZERO 0

Ⅲ.基本操作   
1. 读入：   
用Scanner类定义对象进行控制台读入,Scanner类在java.util.\*包中

Scanner cin=new Scanner(System.in);// 读入

while(cin.hasNext()){ // 等同于!=EOF

int n;

BigInteger m;

n = cin.nextInt(); // 读入一个int;

m = cin.BigInteger();// 读入一个BigInteger;

System.out.print(m.toString());

}

Ⅳ.运用   
四则预算：

import java.util.Scanner;

import java.math.\*;

import java.text.\*;

public class Main {

public static void main(String args[]) {

Scanner cin = new Scanner(System.in);

BigInteger a, b;

int c;

char op;

String s;

while (cin.hasNext()) {

a = cin.nextBigInteger();

s = cin.next();

op = s.charAt(0);

if (op == '+') {

b = cin.nextBigInteger();

System.out.println(a.add(b));

} else if (op == '-') {

b = cin.nextBigInteger();

System.out.println(a.subtract(b));

} else if (op == '\*') {

b = cin.nextBigInteger();

System.out.println(a.multiply(b));

} else {

BigDecimal a1, b1, eps;

String s1, s2, temp;

s1 = a.toString();

a1 = new BigDecimal(s1);

b = cin.nextBigInteger();

s2 = b.toString();

b1 = new BigDecimal(s2);

c = cin.nextInt();

eps = a1.divide(b1, c, 4);

// System.out.println(a + " " + b + " " + c);

// System.out.println(a1.doubleValue() + " " + b1.doubleValue()

// + " " + c);

System.out.print(a.divide(b) + " " + a.mod(b) + " ");

if (c != 0) {

temp = "0.";

for (int i = 0; i < c; i++)

temp += "0";

DecimalFormat gd = new DecimalFormat(temp);

System.out.println(gd.format(eps));

} else

System.out.println(eps);

}

}

}

}

//=====================================================================================  
//PKU1311八进制浮点数化为十进制浮点数，高精度

import java.io.\*;

import java.util.\*;

import java.math.\*;

public class Main {

public static void main(String[] args) {

Scanner cin = new Scanner(System.in);

BigDecimal temp, sum, ans, num; // java大数

String str;

int i, len;

while (cin.hasNext()) {

str = cin.next();

len = str.length();

temp = BigDecimal.valueOf(8.0);

sum = BigDecimal.ONE;

ans = BigDecimal.ZERO;

for (i = 2; i < len; i++) {

int val = str.charAt(i) - '0';

num = BigDecimal.valueOf(val);

sum = sum.multiply(temp); // 8的n次幂

ans = ans.add(num.divide(sum)); // 按权累加

}

System.out.printf("%s [8] = ", str);

System.out.println(ans + " [10]");

}

}

}