**强连通图**

int n,m,x[N],y[N];

int head[N],len;

int dfn[N],low[N],dfs\_num;//dfn表示遍历深度，low(u)为u或u的子树能够追溯到的最早的栈中节点的次序号

int color[N],col\_num,num[N];//染色

int stack[N],vis[N],top;//栈和栈指针

struct EdgeNode{

int from,to,next,w;

}edge[N\*N];

void add(int i,int j,int w){

edge[len].from=i;

edge[len].to=j;

edge[len].w=w;

edge[len].next=head[i];

head[i]=len++;

}

void init(){

mem(vis,0),mem(dfn,0),mem(low,0),mem(head,-1),len=top=col\_num=dfs\_num=0;

}

void tarjan(int u){

dfn[u]=low[u]=++dfs\_num;

vis[u]=1;//是否在栈中

stack[++top]=u;

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

int v=edge[i].to;

if(!dfn[v]){

tarjan(v);

low[u]=min(low[u],low[v]);

}

else if(vis[v]){

low[u]=min(dfn[v],low[u]);

}

}

if(dfn[u]==low[u]){//构成强连通分量

vis[u]=0;

//染色

color[u]=++col\_num;

num[col\_num]=1;

while(stack[top]!=u){//退栈

color[stack[top]]=col\_num;

num[col\_num]++;

vis[stack[top]]=0;

top--;

}

top--;

}

}

void solve(){

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

if(!dfn[i]){

tarjan(i);

}

}

}

**LCA**

**Tarjan离线**

int head[N],len;

int qhead[N],qlen;

int F[N],ance[N],dis[N];

bool vis[N];

struct EdgeNode{

int from,to,next;

int lca,w;

}edge[2\*N],qedge[N];

void add(int i,int j,int w){

edge[len].from=i;

edge[len].to=j;

edge[len].w=w;

edge[len].next=head[i];

head[i]=len++;

}

void qadd(int i,int j){

qedge[qlen].from=i;

qedge[qlen].to=j;

qedge[qlen].lca=-1;

qedge[qlen].next=qhead[i];

qhead[i]=qlen++;

}

void init(){

mem(vis,0),mem(head,-1),mem(qhead,-1),len=qlen=0;

}

int Find(int x){

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

}

void LCA(int u){

ance[u]=F[u]=u;

vis[u]=1;

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

int temp=edge[k].to;

if(!vis[temp]){

dis[temp]=dis[u]+edge[k].w;

LCA(temp);

F[temp]=u;

}

}

for(int k=qhead[u];k!=-1;k=qedge[k].next){

int temp=qedge[k].to;

if(vis[temp]){

qedge[k].lca=qedge[k^1].lca=ance[Find(temp)];

}

}

}

int main(){

int n,m,q,u,v,w,ans;

char x;

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

init();

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

scanf("%d%d%d %c",&u,&v,&w,&x);

add(u,v,w);

add(v,u,w);

}

scanf("%d",&q);

for(int i=1;i<=q;i++){

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

qadd(u,v);

qadd(v,u);

}

dis[1]=0;

LCA(1);

for(int i=0;i<q;i++){

ans=dis[qedge[2\*i].from]+dis[qedge[2\*i].to]-2\*dis[qedge[2\*i].lca];

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

}

}

return 0;

}

**基于RMQ在线**

const int N = 100000 + 5;

const int mod = 1000000000 + 7;

const double eps = 1e-8;

int edgenum,head[N];//edgenum:边的数量 head:存兄弟结点

int id[N<<1],deep[N<<1],first[N];//id:该坐标对应的结点编号 deep:深度 first:点编号第一次出现的坐标

int dp[N<<1][30];

int tot;//深搜坐标

struct Edge{

int v,next;

Edge(int v=0,int next=0):v(v),next(next){}

}e[N<<1];

void init(){

tot=0,edgenum=0;

mem(head,-1);

}

void add(int u,int v){

e[edgenum]=Edge(v,head[u]);

head[u]=edgenum++;

}

void dfs(int u,int fa,int dep){

id[++tot]=u;//tot坐标对应结点为u

first[u]=tot;//第一次出现u结点是在tot坐标

deep[tot]=dep;//更新tot坐标的深度

for(int k=head[u];k!=-1;k=e[k].next){

int v=e[k].v;

if(v==fa) continue;

dfs(v,u,dep+1);

//下面两句表示dfs还要回溯到父亲结点

deep[++tot]=dep;

id[tot]=u;

}

}

void getst(int n){//RMQ前的初始化

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

dp[i][0]=i;

}

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

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

int ll=dp[i][j-1],rr=dp[i+(1<<(j-1))][j-1];

dp[i][j]=deep[ll]<deep[rr]?ll:rr;

}

}

}

int rmq(int l,int r){

int k=(int)log(r-l+1)/log(2.0);

int ll=dp[l][k],rr=dp[r-(1<<k)+1][k];

return deep[ll]<deep[rr]?ll:rr;//返回最小深度的坐标

}

int lca(int u,int v){

u=first[u],v=first[v];//结点第一次出现坐标

if(u>v) swap(u,v);

int res=rmq(u,v);//求最小深度的坐标

return id[res];//返回对应的结点

}

int main(){

int n,q,u,v,LCA;

init();

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

for(int i=2;i<=n;i++){

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

add(u,v),add(v,u);

}

dfs(1,1,0);

getst(tot);

while(q--){

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

LCA=lca(u,v);

printf("%d %d的LCA:%d\n",u,v,LCA);

}

return 0;

}

**拓扑排序**

int n,m,len;

int head[N],indeg[N],ans[N];

struct node{

int to,w,next;

}edge[N];

void add(int i,int j,int w){

indeg[j]++;

edge[len].to=j;

edge[len].w=w;

edge[len].next=head[i];

head[i]=len++;

}

bool topo(){

mem(ans,0);

int cnt=0;

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

if(indeg[i]==0){

ans[++cnt]=i;

}

}

for(int i=1;i<=cnt;i++){

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

indeg[edge[k].to]--;

if(indeg[edge[k].to]==0){

ans[++cnt]=edge[k].to;

}

}

}

return cnt==n;

}

void init(){

mem(head,-1),mem(edge,0),mem(indeg,0),len=1;

}

**最短路**

**Dijkatra队列优化**

int n,m,dis[N];

vector<node> edge[N\*N];

struct node{//邻接表结点

int to,w;//终点和权值

node(){}

node(int to,int w):to(to),w(w){}

bool operator< (const node &a) const{

if(w==a.w) return to<a.to;

else return w>a.w;

}

};

void Dijkstra(int s){//起点

for(int i=0;i<=n;i++) dis[i]=MAX\_INT;

dis[s]=0;

priority\_queue<node> q;

q.push(node(s,dis[s]));

while(!q.empty()){

node x=q.top();q.pop();

for(int i=0;i<edge[x.to].size();i++){

node y=edge[x.to][i];

if(dis[y.to]>x.w+y.w){

dis[y.to]=x.w+y.w;

q.push(node(y.to,dis[y.to]));

}

}

}

}

int main(){

int u,v,w;

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

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

edge[i].clear();

}

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

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

edge[u].pb(node(v,w));

edge[v].pb(node(u,w));

}

return 0;

}

**SPFA(判断负环)**

int head[N],len,n,m;

int dis[N],cnt[N];

bool vis[N];

struct node{

int to,next,w;

}edge[N];

void add(int i,int j,int w){

edge[len].to=j;

edge[len].w=w;

edge[len].next=head[i];

head[i]=len++;

}

void init(){

mem(head,-1),len=0;

}

bool SPFA(int s,int n,int head[],node edge[],int dis[]){

//vis标记是否在队列中

mem(vis,0),mem(cnt,0);

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

dis[i]=MAX\_INT;

}

dis[s]=0;

queue<int> q;

q.push(s);vis[s]=1;cnt[s]=1;

while(!q.empty()){

//队头元素出队，并且消除标记

int x=q.front();q.pop();vis[x]=0;

for(int k=head[x];k!=-1;k=edge[k].next){

int y=edge[k].to;

if(dis[x]+edge[k].w<dis[y]){

dis[y]=dis[x]+edge[k].w;//松弛

if(!vis[y]){//点y不在队内

vis[y]=1;//标记

cnt[y]++;//统计次数

q.push(y);

if(cnt[y]>n)//超过入队次数上限说明有负环

return 0;

}

}

}

}

return 1;

}

**K短路**

struct node{

int to,w,next;

};

struct node2{

int to;

int g,f;

bool operator< (const node2 &a) const{

if(a.f==f){

return a.g<g;

}

return a.f<f;

}

};

node edge[N],edge2[N];

int n,m,s,k,t,cnt,cnt2;

int dis[N],vis[N],head[N],head2[N];

void add(int i,int j,int w){

edge[cnt].to=j;

edge[cnt].w=w;

edge[cnt].next=head[i];

head[i]=cnt++;

}

void add2(int i,int j,int w){

edge2[cnt2].to=j;

edge2[cnt2].w=w;

edge2[cnt2].next=head2[i];

head2[i]=cnt2++;

}

bool SPFA(int s,int n,int head[],node edge[],int dis[]){

//vis标记是否在队列中

int inq[N];

mem(vis,0),mem(inq,0);

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

dis[i]=MAX\_INT;

}

dis[s]=0;

queue<int> q;

q.push(s);vis[s]=inq[s]=1;

while(!q.empty()){

//队头元素出队，并且消除标记

int x=q.front();q.pop();vis[x]=0;

for(int k=head[x];k!=-1;k=edge[k].next){

int y=edge[k].to;

if(dis[x]+edge[k].w<dis[y]){

dis[y]=dis[x]+edge[k].w;//松弛

if(!vis[y]){//点y不在队内

vis[y]=1;//标记

inq[y]++;//统计次数

q.push(y);

if(inq[y]>n)//超过入队次数上限说明有负环

return 0;

}

}

}

}

return 1;

}

int A\_star(int s,int t,int n,int k,int head[],node edge[],int dis[]){

node2 e,ne;

int cnt=0;

priority\_queue<node2> q;

if(s==t) k++;

if(dis[s]==MAX\_INT) return -1;

e.to=s,e.g=0,e.f=e.g+dis[e.to];

q.push(e);

while(!q.empty()){

e=q.top();q.pop();

if(e.to==t){//找到一条最短路

cnt++;

}

if(cnt==k){//找到k短路

return e.g;

}

for(int k=head[e.to];k!=-1;k=edge[k].next){

ne.to=edge[k].to;

ne.g=e.g+edge[k].w;

ne.f=ne.g+dis[ne.to];

q.push(ne);

}

}

return -1;

}

void init(){

mem(head,-1),mem(head2,-1);

cnt=cnt2=1;

}

int main(){

int u,v,w,ans;

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

init();

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

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

add(u,v,w);

add2(v,u,w);

}

scanf("%d%d%d",&s,&k,&t);

SPFA(t,n,head2,edge2,dis);

ans=A\_star(s,t,n,k,head,edge,dis);

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

}

return 0;

}

**欧拉回路**

int head[N],ans[N],num,len;

bool vis[2\*N];

struct EdgeNode{

int to,next;

}edge[2\*N];

void add(int i,int j){

edge[len].to=j;

edge[len].next=head[i];

head[i]=len++;

}

void dfs(int now){

for(int k=head[now];k!=-1;k=edge[k].next){

if(!vis[k]){

vis[k]=1;

vis[k^1]=1;//有向图不要这一句

dfs(edge[k].to);

ans[num++]=edge[k].to;

}

}

}

**网络流最大流**

int n,m;

int head[N],len;

int dep[N],cur[N];//cur就是记录当前点u循环到了哪一条边

struct EdgeNode{

int to,next,flow,cap;

}edge[N\*N];

void add(int i,int j,int flow){

edge[len].to=j;edge[len].cap=flow;edge[len].flow=0;edge[len].next=head[i];head[i]=len++;

edge[len].to=i;edge[len].cap=0;edge[len].flow=0;edge[len].next=head[j];head[j]=len++;

}

void init(){

mem(head,-1),len=0;

}

//分层

bool bfs(int s,int t){

queue<int> q;q.push(s);

mem(dep,0),dep[s]=1;

while(!q.empty()){

int u=q.front();q.pop();

if(u==t) return 1;

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

int v=edge[k].to;

if(edge[k].cap>edge[k].flow&&!dep[v]){

q.push(v);

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

}

}

}

return 0;

}

int DFS(int u,int a,int t){

if(u==t||a==0) return a;

int flow=0,f;

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

int v=edge[k].to;

if(dep[v]==dep[u]+1){

f=DFS(v,min(a,edge[k].cap-edge[k].flow),t);

if(f>0){

edge[k].flow+=f;

edge[k^1].flow-=f;

flow+=f;

a-=f;

if(a==0) break;

}

}

}

return flow;

}

int Dinic(int s,int t){

int ans=0;

while(bfs(s,t)){

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

cur[i]=head[i];

}

ans+=DFS(s,MAX\_INT,t);

}

return ans;

}

**费用流**

int head[N],len;

int pre[N],dis[N];//记录前驱点

int ans;

bool vis[N];

struct node{

int from,to,w,c,next;

}edge[M\*4];

void init(){

mem(head,-1),ans=len=0;

}

void add(int u,int v,int w,int c){

edge[len].from=u,edge[len].to=v,edge[len].w=w,edge[len].c=c;

edge[len].next=head[u],head[u]=len++;

edge[len].from=v,edge[len].to=u,edge[len].w=-w,edge[len].c=0;

edge[len].next=head[v],head[v]=len++;

}

void update(int s,int t){

int f=INF;

for(int i=t;i!=s;i=edge[pre[i]].from){

f=min(f,edge[pre[i]].c);

}

for(int i=t;i!=s;i=edge[pre[i]].from){

edge[pre[i]].c-=f;

edge[pre[i]^1].c+=f;

ans+=f\*edge[pre[i]].w;

}

}

bool spfa(int s,int t){

int u,v,w;

mem(pre,-1),mem(dis,0x3f);

queue<int> q;

q.push(s);dis[s]=0,vis[s]=1;

while(!q.empty()){

u=q.front();q.pop();

vis[u]=0;

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

v=edge[i].to,w=edge[i].w;

if(edge[i].c&&dis[v]>dis[u]+w){

dis[v]=dis[u]+w;

pre[v]=i;

if(!vis[v]){

vis[v]=1;

q.push(v);

}

}

}

}

return pre[t]!=-1;

}

void solve(int s,int t){

ans=0;

while(spfa(s,t)){

update(s,t);

}

}

int main(){

int n,m,u,v,w,c;

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

init();

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

scanf("%d%d%d%d",&u,&v,&w,&c);

add(u,v,w,c);

}

solve(1,n);

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

}

return 0;

}

**次小生成树**

int F[N],ans;

int Find(int x){

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

}

struct node{

int a,b;

int w;

bool flag;

}edge[N\*N/2];

bool cmp(node a,node b){

if(a.w!=b.w) return a.w<b.w;

if(a.a!=b.a) return a.a<b.a;

return a.b<b.b;

}

struct Node{

int to,next;

}link[N];

int head[N];//邻接表的头节点位置

int end[N];//邻接表的尾节点位置

int len[N][N];//每两点在最小生成树上路径中的最长边长

void kruskal(node \*edge,int n,int m){

int x,y,cnt=0;

//初始化邻接表，对于每个节点添加一条指向其自身的边，表示以i为代表元的集合只有点i

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

link[i].to=i+1;

link[i].next=head[i+1];

end[i+1]=i;

head[i+1]=i;

}

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

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

if(cnt==n-1) break;

if(edge[i].w<0) continue;

x=Find(edge[i].a);

y=Find(edge[i].b);

if(x!=y){

//遍历两个节点所在集合

for(int w=head[x];w!=-1;w=link[w].next){

for(int v=head[y];v!=-1;v=link[v].next){

//每次合并两个等价类的时候，分别属于两个等价类的两个点间的最长边一定是当前加入的边

len[link[w].to][link[v].to]=len[link[v].to][link[w].to]=edge[i].w;debug;

}

}

//合并两个邻接表

link[end[y]].next=head[x];

end[y]=end[x];

F[x]=y;

cnt++;

edge[i].flag=1;

}

}

}

void Build(node \*edge,int n,int m){

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

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

}

}

void init(){

mem(edge,0),mem(link,0),mem(head,-1),mem(end,0),mem(len,0);

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

F[i]=i;

}

}

int main(){

int t,n,m,minn,ans;

scanf("%d",&t);

while(t--){

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

init(),minn=0,ans=INT\_MAX;

Build(edge,n,m);

kruskal(edge,n,m);

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

if(edge[i].flag){

minn+=edge[i].w;

}

}

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

if(!edge[i].flag){

ans=min(ans,minn+edge[i].w-len[edge[i].a][edge[i].b]);

}

}

if(ans==minn){

puts("Not Unique!");

}

else{

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

}

}

return 0;

}

**最小树形图**

int id[N],vis[N],pre[N],pos;

LL dis[N],INF=1e17;

struct node{

int u,v,cost;

}edge[N\*N];

LL zhuliu(int root,int V,int E){

LL sum=0;

while(true){

for(int i=0;i<V;i++){

dis[i]=INF;

}

//找最小入边

for(int i=0;i<E;i++){

int u=edge[i].u,v=edge[i].v;

if(u!=v&&dis[v]>edge[i].cost){

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

pre[v]=u;

}

}

//某点不存在入边，算法结束

for(int i=0;i<V;i++){

if(dis[i]==INF&&i!=root){

return -1;

}

}

int cnt=0;

mem(id,-1),mem(vis,-1);

dis[root]=0;

//找环

for(int i=0;i<V;i++){

int v=i;

sum+=dis[i];

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

vis[v]=i;

v=pre[v];

}

//找到环的时候缩点编号

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

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

id[u]=cnt;

}

id[v]=cnt++;

}

}

//如果没有环，则以找到最小树形图，算法结束

if(!cnt){

break;

}

//把余下的不在环里的点编号

for(int i=0;i<V;i++){

if(id[i]==-1){

id[i]=cnt++;

}

}

//更新距离

for(int i=0;i<E;i++){

int u=edge[i].u,v=edge[i].v;

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

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

if(id[u]!=id[v]){

edge[i].cost-=dis[v];

}

}

V=cnt;

root=id[root];

}

return sum;

}

**最小权点基**

int head[N],len;

int dfn[N],low[N],dfs\_num;//dfn表示遍历深度，low(u)为u或u的子树能够追溯到的最早的栈中节点的次序号

int color[N],col\_num;//染色

int stack[N],vis[N],top;//栈和栈指针

int val[N],cost[N],ind[N];

int ans1,ans2;

struct EdgeNode{

int from,to,next;

}edge[N];

void add(int i,int j){

edge[len].from=i;

edge[len].to=j;

edge[len].next=head[i];

head[i]=len++;

}

void init(int n){

mem(ind,0),mem(vis,0),mem(dfn,0),mem(low,0),mem(head,-1),len=ans1=ans2=top=col\_num=dfs\_num=0;

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

cost[i]=MAX\_INT;

}

}

void tarjan(int x){

dfn[x]=++dfs\_num;

low[x]=dfs\_num;

vis[x]=1;//是否在栈中

stack[++top]=x;

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

int temp=edge[i].to;

if(!dfn[temp]){

tarjan(temp);

low[x]=min(low[x],low[temp]);

}

else if(vis[temp]){

low[x]=min(dfn[temp],low[x]);

}

}

if(dfn[x]==low[x]){//构成强连通分量

vis[x]=0;

//染色

color[x]=++col\_num;

while(stack[top]!=x){//退栈

color[stack[top]]=col\_num;

vis[stack[top]]=0;

top--;

}

top--;

}

}

void solve(int n){

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

if(!dfn[i]){

tarjan(i);

}

}

//计算入度并缩点

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

if(color[edge[i].from]!=color[edge[i].to]){

ind[color[edge[i].to]]++;

}

}

//计算每个强连通分量的最小点权值

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

if(cost[color[i]]>val[i]){

cost[color[i]]=val[i];

}

}

for(int i=1;i<=col\_num;i++){

//若入度为0，则加入最小权值

if(ind[i]==0){

ans1++;

ans2+=cost[i];

}

}

}

**桥**

int head[N],len;

int dfn[N],low[N],dfs\_num;//dfn表示遍历深度，low(u)为u或u的子树能够追溯到的最早的栈中节点的次序号

int stack[N],top;//栈和栈指针

int ansx[N],ansy[N],col\_num;

int cnt;

struct EdgeNode{

int to,next;

}edge[N\*N];

void add(int i,int j){

edge[len].to=j;

edge[len].next=head[i];

head[i]=len++;

}

void init(){

mem(dfn,0),mem(low,0),mem(head,-1),cnt=len=top=col\_num=dfs\_num=0;

}

void addAns(int x,int y){

if(x>y){

swap(x,y);

}

ansx[++cnt]=x,ansy[cnt]=y;

}

void tarjan(int x,int from){

dfn[x]=low[x]=++dfs\_num;

stack[++top]=x;

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

int temp=edge[i].to;

if(temp==from) continue;

if(!dfn[temp]){

tarjan(temp,x);

low[x]=min(low[x],low[temp]);

if(dfn[x]<low[temp]){

addAns(x,temp);

}

}

else{

low[x]=min(low[x],dfn[temp]);

}

}

}

void solve(int n){

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

if(!dfn[i]){

tarjan(i,-1);

}

}

}

**割点**

int head[N],len;

int dfn[N],low[N],dfs\_num;//dfn表示遍历深度，low(u)为u或u的子树能够追溯到的最早的栈中节点的次序号

int ans[N];

struct EdgeNode{

int to,next;

}edge[N];

void add(int i,int j){

edge[len].to=j;

edge[len].next=head[i];

head[i]=len++;

}

void init(){

mem(ans,0),mem(dfn,0),mem(low,0),mem(head,-1),len=dfs\_num=0;

}

void tarjan(int x,int from){

int son=0;

dfn[x]=low[x]=++dfs\_num;

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

int temp=edge[i].to;

if(temp==from) continue;

if(!dfn[temp]){

son++;

tarjan(temp,x);

low[x]=min(low[x],low[temp]);

if(dfn[x]<=low[temp]){

ans[x]=1;

}

}

else{

low[x]=min(dfn[temp],low[x]);

}

}

if(from<0&&son==1){

ans[x]=0;

}

}

void solve(int n){

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

if(!dfn[i]){

tarjan(i,-1);

}

}

}

int main(){

int n,m,u,v;

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

init();

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

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

add(u,v);

add(v,u);

}

solve(n);

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

if(ans[i]){

cout<<i<<endl;

}

}

}

return 0;

}