**GRAPH CODE LIBRARY**

1.MINIMUM SPANNING TREE

//Minimum and Maximum spanning tree

//edgeArr[] is the array of edges

//parent[] is used to find the disjoint set union and save the parent connection

//MX = maximum edge number in this problem

//MX\_ONE = maximum nodes in this problem

struct info

{

int node1,node2,cost; // edge between node1,node2,cost;

} edgeArr[MX+5];

int parent[MX\_NODE+5]; // the parent declaration

//the disjoint find and disjoint set union

int findParent(int node) {

if(parent[node] == node) return node;

else parent[node] = findParent(parent[node]);

return parent[node];

}

//cmpSmall function is used to sort the edges according to min cost

bool cmpSmall(info a, info b)

{

if(a.cost<b.cost) return true;

else return false;

}

//Function to find the minimum spanning tree

int minimumSpanningTree(int edge, int node)

{

//edge says how many edges the problem has

//node says how much nodes are in this problem

//initializing the parent array

sort(edgeArr+1,edgeArr+1+edge,cmpSmall); // sorting on the basis of small edges cost

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

parent[i] = i; //everyone is his own parent

}

int sum = 0; // the min cost

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

int u = edgeArr[i].node1; // one node

int v = edgeArr[i].node2;

u = findParent(u); //taking the parent of u through dsu

v = findParent(v); //taking the parent of v through dsu

if(u != v) { // they are not connected

sum += edgeArr[i].cost; // adding the minimum cost

parent[v] = u; // making the parent of v as parent of u or u

}

}

return sum;

}

2. MAXIMUM SPANNING TREE

//Minimum and Maximum spanning tree

//edgeArr[] is the array of edges

//parent[] is used to find the disjoint set union and save the parent connection

//MX = maximum edge number in this problem

//MX\_ONE = maximum nodes in this problem

struct info

{

int node1,node2,cost; // edge between node1,node2,cost;

} edgeArr[MX+5];

int parent[MX\_NODE+5]; // the parent declaration

//the disjoint find and disjoint set union

int findParent(int node) {

if(parent[node] == node) return node;

else parent[node] = findParent(parent[node]);

return parent[node];

}

//cmpLarge function is used to sort the edges according to max cost

bool cmpLarge(info a, info b)

{

if(a.cost>b.cost) return true;

else return false;

}

//Function to find the maximum spanning tree

int maximumSpanningTree(int edge, int node)

{

//edge says how many edges the problem has

//node says how much nodes are in this problem

//initializing the parent array

sort(edgeArr+1,edgeArr+1+edge,cmpLarge); // sorting on the basis of large edges cost

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

parent[i] = i; //everyone is his own parent

}

int sum = 0; // the min cost

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

int u = edgeArr[i].node1; // one node

int v = edgeArr[i].node2;

u = findParent(u); //taking the parent of u through dsu

v = findParent(v); //taking the parent of v through dsu

if(u != v) { // they are not connected

sum += edgeArr[i].cost; // adding the minimum cost

parent[v] = u; // making the parent of v as parent of u or u

}

}

return sum;

}

3. Euler’s Circuit/Euler’s path(Hierholzer’s Algorithm)

vector<int>finalPath; //it's the final sequential path of mine

vector<int>edgeknown[MAX+5]; //identified this edge is traversed or not

vector<int>revEdge[MAX+5]; //the same edge which is present on which index of the opposite node

int edgeCounter; // the counted number of traversed edge

void hierholzersEuler(int source) //this is solely made to find euler circuit in a graph

{

for(int i=0; i<graph[source].size(); i++)

{

if(edgeknown[source][i] == 0)

{

//not identified edge

edgeknown[source][i] = 1;

//edgeknown[graph[source][i]][revEdge[source][i]]=1; // specially for undirected edge same edge don't need to be trqversed

edgeCounter++; //counting the edges

hierholzersEuler(graph[source][i]);

}

}

finalPath.push\_back(source);

return;

}

4 . Bridge Tree

//Formation of Bridge Tree

vector<int>corres[MAX+5]; //corresponding edges

vector<int>articulationBridge[MAX+5]; //says this edge is articulation bridge or not

int color[MAX+5];

int low[MAX+5];

int disc[MAX+5];

int Time;

int parent[MAX+5];

//searching of articulation bridges

void findArticulationBridge(int node)

{

color[node] = 1; //1 Means Grey, still processing

Time++;

disc[node] = Time; //disc is discovery time

low[node] = disc[node];

for(int i = 0; i<graph[node].size() ;i++)

{

int v = graph[node][i];

if(color[v] == 0)

{

parent[v] = node;

findArticulationBridge(v);

low[node] = min(low[node],low[v]); //looking if parent node can go down and and go to upper

//nodes with its child nodes

}

else if(color[v] !=0 && v != parent[node]) // it's upper node than nodes parent node so it can go to up

{

low[node] = min(low[node],disc[v]);

}

if(disc[node]<low[v])

{

//this means this child has no previous upper shorter linkage this edge can be considered as a bridge

//so this edge can be considered as an articulation bridge

articulationBridge[node][i] = 1; // articulation bridge indicator

int u = corres[node][i];

articulationBridge[v][u] = 1; //same edge from opposite node

}

}

color[node] = 2;

return;

}

//this function makes the bridge tree

int numCompo; //represents nth bridge

vector<int>tree[MAX+5]; //the bridge nodes

vector<int>treeEdge[MAX+5];//the tree edge of the graph

int visit[MAX+5];

void makeBridgeTree(int node)

{

int presentNum = numCompo; //local working component

visit[node] = 1;

tree[numCompo].push\_back(node); //tree[numCompo] consists of this node

queue<int>Q;

Q.push(node);

while(Q.empty() != true)

{

int u = Q.front();

Q.pop();

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

{

int v = graph[u][i];

if(visit[v] == 1) continue;

else

{

if(articulationBridge[u][i] == 1) //this edge is an articulation bridge and so now I am going to

// a different connected component or bridge node

{

numCompo++;

treeEdge[presentNum].push\_back(numCompo); // an edge from present connected component to where I am going

treeEdge[numCompo].push\_back(presentNum); //bi directional tree

makeBridgeTree(v);

}

else

{

//this node will increase it's component

tree[presentNum].push\_back(v);

visit[v] = 1;

Q.push(v);

}

}

}

}

return;

}

5. Dijkstra

struct Graph{

int node;

int cost;

}temp,matha;

bool operator < (const Graph &a, const Graph &b)

{

if(a.cost>b.cost) return true;

else return false;

}

priority\_queue<Graph>PQ;

void dijkstra(int n,int sr, int des)

{

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

{

dist[i] = INF;

way[i]=0;

visit[i] = 0;

}

temp.node=sr;

temp.cost=0;

dist[sr]=0;

PQ.push(temp);

while(PQ.empty() != true)

{

assert(PQ.size()<=30000000);

matha = PQ.top();

PQ.pop();

//cout<<matha.node<<" "<<matha.cost<<" "<<dist[matha.node]<<endl;

if(visit[matha.node] == 0)

{

//counter[matha.node]++;

visit[matha.node] = 1;

way[matha.node] = 1;

}

else

{

if(matha.cost==dist[matha.node]) way[matha.node]++;

else continue;

}

if(sp[matha.node] == 1 && way[matha.node]>1) return;

if(way[matha.node]>2) continue;

for(int i=0;i<graph[matha.node].size();i++)

{

int v=graph[matha.node][i];

if(dist[v] > dist[matha.node]+weight[matha.node][i])

{

dist[v] = dist[matha.node]+weight[matha.node][i];

temp.node=v;

temp.cost=dist[v];

PQ.push(temp);

assert(PQ.size()<=30000000);

}

else if(dist[v] == dist[matha.node]+weight[matha.node][i])

{

temp.node=v;

temp.cost=dist[v];

if(visit[v] == 0)

{

PQ.push(temp);

assert(PQ.size()<=30000000);

}

else

{

if(way[v]<=1) PQ.push(temp);

else way[v]++;

}

}

assert(PQ.size()<=30000000);

}

}

}

6. Minimum Vertex Cover (Graph + DP)

int dp[MAX+1][3];

int parent[MAX+1];

//minimum vertex cover = maximum matching

int minimumVertexCover(int node, int isGuard){

if(graph[node].size() == 0){

return isGuard; // o means no need to place , 1 means need to place

}

if(dp[node][isGuard] != -1) return dp[node][isGuard];

int ans=0;

for(int i=0;i<graph[node].size();i++) {

int v=graph[node][i];

if(v != parent[node]) { //not calculating for parent

parent[v]=node;

if(isGuard) { //if the guard is placed then there is no mandatory to place guards in child so both try

ans +=min(minimumVertexCover(v,1),minimumVertexCover(v,0));

}

else if(!isGuard){

ans += minimumVertexCover(v,1); //the guard is not present each child needs to have guard

}

}

}

if(isGuard) ans += 1; //need to place here the guard

dp[node][isGuard]=ans;

return dp[node][isGuard];

}

// CALL WILL BE min(minimumVertexCover(1,0), miniumVertexCover(1,1))

// Implementation of Bellman Ford Algorithm, Tested for Lightoj 1074 Extended Traffic

#include <bits/stdc++.h>

using namespace std;

#define MAX 205

#define INF 1000000000

typedef long long int ll;

vector<int>graph[MAX+1];

vector<ll> weight[MAX+1];

ll dist[MAX+1];

int new\_update[MAX+1];

int visit[MAX+1];

void init(){

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

graph[i].clear();

weight[i].clear();

dist[i] = INF;

new\_update[i]=0;

visit[i] = 0;

}

}

void bellman\_ford(int n){

dist[1] = 0;

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

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

for(int k=0;k<graph[j].size();k++) {

if(dist[graph[j][k]]>(dist[j]+weight[j][k])) {

dist[graph[j][k]]=weight[j][k]+dist[j];

}

}

}

}

for(int i=1;i<=50;i++) {

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

for(int k=0;k<graph[j].size();k++) {

if(dist[graph[j][k]]>(dist[j]+weight[j][k])) {

dist[graph[j][k]]=weight[j][k]+dist[j];

new\_update[graph[j][k]] = 1;

}

}

}

}

}

void dfs(int n){

for(int i=0;i<graph[n].size();i++){

if(visit[graph[n][i]]==0) {

visit[graph[n][i]]=1;

dfs(graph[n][i]);

}

}

return;

}

int main(void) {

//freopen("in.txt","r",stdin);

//freopen("out.txt","w",stdout);

int T,t;

scanf("%d",&T);

for(t=1;t<=T;t++){

init();

ll cost[MAX+1];

int n;

scanf("%d",&n);

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

scanf("%lld",&cost[i]);

}

int m;

scanf("%d",&m);

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

int a,b;

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

graph[a].push\_back(b);

ll value = cost[b] - cost[a];

value = value \* value \* value;

weight[a].push\_back(value);

}

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

for(int j=0;j<graph[i].size();j++) {

cout<<"( " << i << " "<< graph[i][j]<<", " << weight[i][j]<<" ) " << endl;

}

}\*/

dfs(1);

bellman\_ford(n);

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

cout<<dist[i]<<" "<<new\_update[i]<<endl;

}\*/

int q;

scanf("%d",&q);

printf("Case %d:\n",t);

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

int a;

scanf("%d",&a);

if(new\_update[a] == 1) {

printf("?\n");

}

else {

if(dist[a]<3 || dist[a] == INF || visit[a] == 0) {

printf("?\n");

}

else {

printf("%lld\n",dist[a]);

}

}

}

}

return 0;

}