**2-Sat :**

// 1- based.....

struct two\_sat{

int n,nn,m; vector<int>G[maxm],GT[maxm],DAG[maxm],C[maxm],topo; // G= graph.. GT= transpose graph.......

int col[maxm],comp,comp\_no[maxm],soln[maxm];

// comp= total number component;

// comp\_no[i]= component no of node i

// soln[i]= truth symbol of node i;

two\_sat(){}

void init(){

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

G[i].clear();

GT[i].clear();

DAG[i].clear();

C[i].clear();

}

topo.clear();

memset(col,0,sizeof(col));

memset(comp\_no,0,sizeof(comp\_no));

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

comp=0;

}

int inv(int no){

if(no<=n) return no+n;

return no-n;

}

void OR(int u,int v){

G[inv(v)].push\_back(u);

G[inv(u)].push\_back(v);

}

void AND(int u,int v){

G[u].push\_back(v);

G[v].push\_back(u);

}

void XOR(int u,int v){

G[inv(v)].push\_back(u);

G[u].push\_back(inv(v));

G[inv(u)].push\_back(v);

G[v].push\_back(inv(u));

}

void XNOR(int u,int v){

G[u].push\_back(v);

G[v].push\_back(u);

G[inv(u)].push\_back(inv(v));

G[inv(v)].push\_back(inv(u));

}

// problem Dependent.......

void build\_graph(){

int u,v,op;

nn=n+n;

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

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

if(op==1) XNOR(u,v);

else if(op==0) XOR(u,v);

}

}

void make\_reverse(){

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

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

GT[G[i][j]].push\_back(i);

}

}

}

int check\_solution(){

// Build scc..........

build\_scc();

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

if(comp\_no[i]==comp\_no[inv(i)]) return 0;

}

return 1;

}

void find\_solution(vector<int>&res){

int i,j,i\_p;

for(i=1;i<=comp;i++){

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

soln[i]=0;

i\_p=comp\_no[inv(C[i][0])];

soln[i\_p]=1;

for(j=0;j<C[i\_p].size();j++){

if(C[i\_p][j]<=n) res.push\_back(C[i\_p][j]);

}

}

}

}

void build\_dag(){

int i,j;

for(i=1;i<=nn;i++){

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

if(comp\_no[i]==comp\_no[G[i][j]]) continue;

DAG[comp\_no[i]].push\_back(comp\_no[G[i][j]]);

}

}

}

void build\_scc(){

make\_reverse();

int i;

for(i=1;i<=nn;i++){

if(!col[i]) dfs(i);

}

for(i=topo.size()-1;i>=0;i--){

if(!comp\_no[topo[i]]){

scc(topo[i],++comp);

}

}

}

void dfs(int s){

if(col[s]) return ;

col[s]=1;

for(int i=0;i<G[s].size();i++){

dfs(G[s][i]);

}

topo.push\_back(s);

}

void scc(int s,int comp){

if(comp\_no[s]) return ;

comp\_no[s]=comp;

C[comp].push\_back(s);

for(int i=0;i<GT[s].size();i++){

scc(GT[s][i],comp);

}

}

};

two\_sat T\_sat;

vector<int>res;

int main(){

int n,m;

while(scanf("%d",&n)==1){

scanf("%d",&m);

T\_sat.n=n; T\_sat.m=m;

T\_sat.init();

T\_sat.build\_graph();

int ans=T\_sat.check\_solution();

if(ans){

res.clear();

T\_sat.find\_solution(res);

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

for(i=0;i<res.size();i++){

if(i) printf(" ");

printf("%d",res[i]);

}

puts("");

}

else{

printf("Impossible\n");

}

}

return 0;

}

**Biconnected Component :**

stack<pii>st\_pii;

set<int>sets[maxm];

void bi\_comp(int u,int v){

while(!st\_pii.empty()){

pii now=st\_pii.top(); st\_pii.pop();

sets[tot].insert(now.uu);

sets[tot].insert(now.vv);

if(now.uu==u && now.vv==v) break;

if(now.uu==v && now.vv==u) break;

}

tot++;

}

void dfs(int s,int pre,int root){

if(vis[s]) return;

vis[s]=1;

low[s]=dep[s]=tim++;

// bi-connected with a single vertex

if(G[s].size()==0){

sets[tot++].insert(s);

return ;

}

int i,j,k,c=0;

for(i=0;i<G[s].size();i++){

int d=G[s][i];

if(d==pre) continue;

if(vis[d] && dep[d]<dep[s]){

st\_pii.push(mp(s,d));

low[s]=mini(low[s],dep[d]);

}

else if(!vis[d]){

st\_pii.push(mp(s,d));

dfs(d,s,root); c++;

if(low[d]>=dep[s]){

bi\_comp(s,d);

if(s!=root){

is\_cut[s]=1;

}

}

low[s]=mini(low[s],low[d]);

}

}

if(s==root && c>1){

is\_cut[s]=1;

}

}

**Bellman Ford :**

struct edge{

int u, v,cost;};

edge edges[maxe]; int d[maxm],flag[maxm];

void bellman(int s,int n,int e){

int i,j,k,l,u,v;

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

flag[i]=0;

d[i]=inf;

}

d[s]=0;

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

for(j=0;j<e;j++){

u=edges[j].u;

v=edges[j].v;

if(d[v]>d[u]+edges[j].cost){

d[v]=d[u]+edges[j].cost;

if(i>n){

// negative cycle .....

flag[v]=1; // node v is in negative cycle

}}}}

**Euler Curcuit Print :**

// A C++ program print Eulerian Trail in a given Eulerian or Semi-Eulerian Graph

// A class that represents an undirected graph

class Graph

{

int V; // No. of vertices

list<int> \*adj; // A dynamic array of adjacency lists

public:

// Constructor and destructor

Graph(int V) { this->V = V; adj = new list<int>[V]; }

~Graph() { delete [] adj; }

// functions to add and remove edge

void addEdge(int u, int v) { adj[u].push\_back(v); adj[v].push\_back(u); }

void rmvEdge(int u, int v);

// Methods to print Eulerian tour

void printEulerTour();

void printEulerUtil(int s);

// This function returns count of vertices reachable from v. It does DFS

int DFSCount(int v, bool visited[]);

// Utility function to check if edge u-v is a valid next edge in

// Eulerian trail or circuit

bool isValidNextEdge(int u, int v);

};

/\* The main function that print Eulerian Trail. It first finds an odd

degree vertex (if there is any) and then calls printEulerUtil()

to print the path \*/

void Graph::printEulerTour()

{

// Find a vertex with odd degree

int u = 0;

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

if (adj[i].size() & 1)

{ u = i; break; }

// Print tour starting from oddv

printEulerUtil(u);

cout << endl;

}

// Print Euler tour starting from vertex u

void Graph::printEulerUtil(int u)

{

// Recur for all the vertices adjacent to this vertex

list<int>::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i)

{

int v = \*i;

// If edge u-v is not removed and it's a a valid next edge

if (v != -1 && isValidNextEdge(u, v))

{

cout << u << "-" << v << " ";

rmvEdge(u, v);

printEulerUtil(v);

}

}

}

// The function to check if edge u-v can be considered as next edge in

// Euler Tout

bool Graph::isValidNextEdge(int u, int v)

{

// The edge u-v is valid in one of the following two cases:

// 1) If v is the only adjacent vertex of u

int count = 0; // To store count of adjacent vertices

list<int>::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i)

if (\*i != -1)

count++;

if (count == 1)

return true;

// 2) If there are multiple adjacents, then u-v is not a bridge

// Do following steps to check if u-v is a bridge

// 2.a) count of vertices reachable from u

bool visited[V];

memset(visited, false, V);

int count1 = DFSCount(u, visited);

// 2.b) Remove edge (u, v) and after removing the edge, count

// vertices reachable from u

rmvEdge(u, v);

memset(visited, false, V);

int count2 = DFSCount(u, visited);

// 2.c) Add the edge back to the graph

addEdge(u, v);

// 2.d) If count1 is greater, then edge (u, v) is a bridge

return (count1 > count2)? false: true;

}

// This function removes edge u-v from graph. It removes the edge by

// replacing adjcent vertex value with -1.

void Graph::rmvEdge(int u, int v)

{

// Find v in adjacency list of u and replace it with -1

list<int>::iterator iv = find(adj[u].begin(), adj[u].end(), v);

\*iv = -1;

// Find u in adjacency list of v and replace it with -1

list<int>::iterator iu = find(adj[v].begin(), adj[v].end(), u);

\*iu = -1;

}

// A DFS based function to count reachable vertices from v

int Graph::DFSCount(int v, bool visited[])

{

// Mark the current node as visited

visited[v] = true;

int count = 1;

// Recur for all vertices adjacent to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (\*i != -1 && !visited[\*i])

count += DFSCount(\*i, visited);

return count;

}

// Driver program to test above function

int main()

{

// Let us first create and test graphs shown in above figure

Graph g1(4);

g1.addEdge(0, 1);

g1.addEdge(0, 2);

g1.addEdge(1, 2);

g1.addEdge(2, 3);

g1.printEulerTour();

Graph g2(3);

g2.addEdge(0, 1);

g2.addEdge(1, 2);

g2.addEdge(2, 0);

g2.printEulerTour();

Graph g3(5);

g3.addEdge(1, 0);

g3.addEdge(0, 2);

g3.addEdge(2, 1);

g3.addEdge(0, 3);

g3.addEdge(3, 4);

g3.addEdge(3, 2);

g3.addEdge(3, 1);

g3.addEdge(2, 4);

g3.printEulerTour();

return 0;

}

### Hierholzer's algorithm ( Euler Cuircuit Print ):

[Hierholzer](http://en.wikipedia.org/wiki/Carl_Hierholzer)'s 1873 paper provides a different method for finding Euler cycles that is more efficient than Fleury's algorithm:

* Choose any starting vertex *v*, and follow a trail of edges from that vertex until returning to *v*. It is not possible to get stuck at any vertex other than *v*, because the even degree of all vertices ensures that, when the trail enters another vertex *w* there must be an unused edge leaving *w*. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.
* As long as there exists a vertex *v* that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from *v*, following unused edges until returning to *v*, and join the tour formed in this way to the previous tour.

By using a data structure such as a [doubly linked list](http://en.wikipedia.org/wiki/Doubly_linked_list) to maintain the set of unused edges incident to each vertex, to maintain the list of vertices on the current tour that have unused edges, and to maintain the tour itself, the individual operations of the algorithm (finding unused edges exiting each vertex, finding a new starting vertex for a tour, and connecting two tours that share a vertex) may be performed in constant time each, so the overall algorithm takes [linear time](http://en.wikipedia.org/wiki/Linear_time).

Fleury's algorithm is an elegant but inefficient algorithm which dates to 1883.[[7]](http://en.wikipedia.org/wiki/Eulerian_path#cite_note-7) Consider a graph known to have all edges in the same component and at most two vertices of odd degree. The algorithm starts at a vertex of odd degree, or, if the graph has none, it starts with an arbitrarily chosen vertex. At each step it chooses the next edge in the path to be one whose deletion would not disconnect the graph, unless there is no such edge, in which case it picks the remaining edge left at the current vertex. It then moves to the other endpoint of that vertex and deletes the chosen edge. At the end of the algorithm there are no edges left, and the sequence from which the edges were chosen forms an Eulerian cycle if the graph has no vertices of odd degree, or an Eulerian trail if there are exactly two vertices of odd degree.

While the *graph traversal* in Fleury's algorithm is linear in the number of edges, i.e. *O*(|*E*|), we also need to factor in the complexity of detecting [bridges](http://en.wikipedia.org/wiki/Bridge_%28graph_theory%29). If we are to re-run [Tarjan](http://en.wikipedia.org/wiki/Robert_Tarjan)'s linear time bridge-finding algorithm after the removal of every edge, Fleury's algorithm will have a time complexity of *O*(|*E*|2). A dynamic bridge-finding algorithm of [Thorup (2000](http://en.wikipedia.org/wiki/Eulerian_path#CITEREFThorup2000)) allows this to be improved to O(|E|\log^3|E|\log\log|E|)but this is still significantly slower than alternative algorithms.

**Convex Hull Trick 1:**

*/\**

*ID: brian\_bi21*

*PROG: acquire (Usaco Mar 08).*

*Algo: Convex Hull Trick.*

*\*/*

#include <iostream>

#include <vector>

#include <algorithm>

**using** **namespace** std;

int pointer; *//Keeps track of the best line from previous query*

vector<long long> M; *//Holds the slopes of the lines in the envelope*

vector<long long> B; *//Holds the y-intercepts of the lines in the envelope*

*//Returns true if either line l1 or line l3 is always better than line l2*

bool **bad**(int l1,int l2,int l3)

{

*/\**

*intersection(l1,l2) has x-coordinate (b1-b2)/(m2-m1)*

*intersection(l1,l3) has x-coordinate (b1-b3)/(m3-m1)*

*set the former greater than the latter, and cross-multiply to*

*eliminate division*

*\*/*

**return** (B[l3]-B[l1])\*(M[l1]-M[l2])<(B[l2]-B[l1])\*(M[l1]-M[l3]);

}

*//Adds a new line (with lowest slope) to the structure*

void **add**(long long m,long long b)

{

*//First, let's add it to the end*

M.**push\_back**(m);

B.**push\_back**(b);

*//If the penultimate is now made irrelevant between the antepenultimate*

*//and the ultimate, remove it. Repeat as many times as necessary*

**while** (M.**size**()>=3&&**bad**(M.**size**()-3,M.**size**()-2,M.**size**()-1))

{

M.**erase**(M.**end**()-2);

B.**erase**(B.**end**()-2);

}

}

*//Returns the minimum y-coordinate of any intersection between a given vertical*

*//line and the lower envelope*

long long **query**(long long x)

{

*//If we removed what was the best line for the previous query, then the*

*//newly inserted line is now the best for that query*

**if** (pointer>=M.**size**())

pointer=M.**size**()-1;

*//Any better line must be to the right, since query values are*

*//non-decreasing*

**while** (pointer<M.**size**()-1&&

M[pointer+1]\*x+B[pointer+1]<M[pointer]\*x+B[pointer])

pointer++;

**return** M[pointer]\*x+B[pointer];

}

int **main**()

{

int M,N,i;

pair<int,int> a[50000];

pair<int,int> rect[50000];

**freopen**("acquire.in","r",stdin);

**freopen**("acquire.out","w",stdout);

**scanf**("%d",&M);

**for** (i=0; i<M; i++)

**scanf**("%d %d",&a[i].first,&a[i].second);

*//Sort first by height and then by width (arbitrary labels)*

**sort**(a,a+M);

**for** (i=0,N=0; i<M; i++)

{

*/\**

*When we add a higher rectangle, any rectangles that are also*

*equally thin or thinner become irrelevant, as they are*

*completely contained within the higher one; remove as many*

*as necessary*

*\*/*

**while** (N>0&&rect[N-1].second<=a[i].second)

N--;

rect[N++]=a[i]; *//add the new rectangle*

}

long long cost;

**add**(rect[0].second,0);

*//initially, the best line could be any of the lines in the envelope,*

*//that is, any line with index 0 or greater, so set pointer=0*

pointer=0;

**for** (i=0; i<N; i++) *//discussed in article*

{

cost=**query**(rect[i].first);

**if** (i<N)

**add**(rect[i+1].second,cost);

}

**printf**("%lld\n",cost);

**return** 0;

}

**Divide and Conquer Optimization :**

*/\**

*Author : rng\_58.*

*Sufficient Condition : pre[i][j]< pre[i][j+1] < pre[i][j+2].*

*Pre = Optimal path tracker .*

*\*/*

**REP**(i,N+1) dp[1][i] = **get\_cost**(0, i);

**for**(i=1;i<K;i++) **func**(i, 0, N+1, 0, N);

void **func**(int d, int l, int r, int sepl, int sepr){

int i;

**if**(r-l == 1) **return**;

int m = (l + r) / 2;

int sep = -1;

dp[d+1][m] = INF;

**for**(i=sepl;i<=sepr;i++) **if**(i <= m){

int tmp = dp[d][i] + **get\_cost**(i, m);

**if**(tmp < dp[d+1][m]){

dp[d+1][m] = tmp;

sep = i;

}

}

**func**(d, l, m, sepl, sep);

**func**(d, m, r, sep, sepr);

}

**Knuth Optimization :**

Let F[a][b] be the minimum cost to make all cuts from a to b inclusive.  
In the standart n^3 solution :  
  
F[a][b] = min(F[a][c-1] + F[c+1][b] + length(a, b)) - for every c from a to b;  
  
Let P[a][b] be the c for which F[a][b] is minimized. It can be shown that:  
  
F[a][b] = min(F[a][c-1] + F[c+1][b] + length(a, b)) - for every c from

P[a][b - 1] to P[a + 1][b];

**# convert spherical to cartesian co-ordinate......**

*// convert spherical to cartesian co-ordinate......*

void **sph\_to\_cartesian**(double R,double lat,double lng,point3D &p){

lat=**convdr**(lat);

lng=**convdr**(lng);

p.x=R\***sin**(lat)\***cos**(lng);

p.y=R\***sin**(lat)\***sin**(lng);

p.z=R\***cos**(lat);

}

**# convert longitude/latitude to cartesian co-ordinate......**

*// convert longitude/latitude to cartesian co-ordinate......*

void **earth\_to\_cartesian**(double R,double lat,double lng,point3D &p){

lat=**convdr**(lat);

lng=**convdr**(lng);

p.x=R\***cos**(lat)\***cos**(lng);

p.y=R\***cos**(lat)\***sin**(lng);

p.z=R\***sin**(lat);

}

**# convert cartesian co-ordinate to longitude/latitude**

lat = asin(z / R)

lon = atan2(y, x)

**Inside Triangle**

bool **inside\_tri**(tri t,point p){

point p1=t.p1,p2=t.p2,p3=t.p3;

*// check for boundary........*

**if**(**iseq**(**cross**(p,p1,p2),0) && **inside\_segment**(**segment**(p1,p2),p)) **return** 1;

**if**(**iseq**(**cross**(p,p2,p3),0) && **inside\_segment**(**segment**(p2,p3),p)) **return** 1;

**if**(**iseq**(**cross**(p,p1,p3),0) && **inside\_segment**(**segment**(p1,p3),p)) **return** 1;

*// .........*

**if**(**cross**(p,p1,p2)\***cross**(p3,p1,p2)<0) **return** 0;

**if**(**cross**(p,p2,p3)\***cross**(p1,p2,p3)<0) **return** 0;

**if**(**cross**(p,p1,p3)\***cross**(p2,p1,p3)<0) **return** 0;

**return** 1;

}

**Important Formulas**

# Area of a triangle :

Let *K* be the triangle's area and let *a*, *b* and *c*, be the lengths of its sides. By Heron’s Formula , the area of the triangle is

K = sqrt( s \* (s-a) \* (s-b) \* (s –c) ).

where **S** is the semiperimeter .

s= \tfrac{1}{2}(a+b+c) .

**length of median to side c** = sqrt(2\*(a\*a+b\*b)-c\*c)/2

**length of bisector of angle C** = sqrt(ab[(a+b)\*(a+b)-c\*c])/(a+b) .

**Radius of a In-cicle:**

The radius of the incircle (also known as the **inradius**, *r* ) is

r = \frac{2K}{P} = \sqrt{\frac{(s-a)(s-b)(s-c)}{s}}.

Thus, the area *K* of a triangle may be found by multiplying the inradius by the semiperimeter:

\displaystyle K=rs.

**Regular Polygon :**

a regular polygon is a Polygon that is [equiangular](http://en.wikipedia.org/wiki/Equiangular_polygon) (all angles are equal in measure) and [equilateral](http://en.wikipedia.org/wiki/Equilateral) (all sides have the same length).

**Angle :**

For a regular convex *n*-gon, each interior angle has a measure of:

**(n-2)\times \frac{180}{n}**  degrees .

**Apothem:** The **apothem** of a [regular polygon](http://en.wikipedia.org/wiki/Regular_polygon) is a line segment from the center to the midpoint of one of its sides. Equivalently, it is the line drawn from the center of the polygon that **is perpendicular to one of its sides.**

**Circumradius:**

The [**circumradius**](http://en.wikipedia.org/wiki/Circumradius) from the center of a regular polygon to one of the vertices is related to the side **length** ***s*** or to the [**apothem**](http://en.wikipedia.org/wiki/Apothem) ***a*** by

r=\frac{s}{2 \sin{ \frac{\pi}{n} }} = \frac{a}{\cos{ \frac{\pi}{n} }} .

**Area :**

The **area *A*** of a convex regular *n*-sided polygon having **Side *s***, **circumradius *r***, **apothem *a***, and **perimeter *p*** is given by

A= \tfrac{1}{2}nsa = \tfrac{1}{2}pa = \tfrac{1}{4}ns^2\cot{\tfrac{\pi}{n}} = na^2\tan{\tfrac{\pi}{n}} = \tfrac{1}{2}nr^2\sin{\tfrac{2\pi}{n}}

# Centroid of a 2D polygon:

As in the calculation of the area above, xN is assumed to be x0, in other words the polygon is closed.



**Searching**

**Ternary Search :**

double **ts**(){

double min=0;

double max=1;

int c=100; //for higher precision have to increase

double k,l,f,g;

**while**(c--){

f=min+(max-min)/(double)3.0;

g=min+(double)2.0\*((max-min)/(double)3.0);

k=**fun**(f); l=**fun**(g);

**if**(k<l){

max=g;

}

**else**{

min=f;

}

}

return (min+max)/2.0 ;

}

// problem dependent . . . .

double **fun**(double piv){  
}

**Green Hackenbush :**

*/\**

*Author : misof*

*Problem : ipsc 2003 G [hackenbush] (c) misof*

*Algo : Green Hackenbush .*

*\*/*

#define **min**(x,y) ((x)<(y))?(x):(y)

int Cases,N,M;

vector< list<int> > G,G2;

vector<int> GV;

vector<int> visited,from,time\_disc,time\_up;

int DFStime;

void **DFS\_Visit**(int v){

int edges\_to\_parent=0;

visited[v]=1; time\_disc[v]=time\_up[v]=++DFStime;

**for** (list<int>::iterator start=G[v].**begin**();start!=G[v].**end**();start++) {

**if** (!visited[\*start]) { from[\*start]=v; **DFS\_Visit**(\*start); time\_up[v]=**min**(time\_up[v],time\_up[\*start]); }

**else** {

**if** ((\*start)!=from[v]) { time\_up[v]=**min**(time\_up[v],time\_disc[\*start]); }

**else** {

**if** (edges\_to\_parent) { time\_up[v]=**min**(time\_up[v],time\_disc[\*start]); }

edges\_to\_parent++;

}

}

}

}

void **FindBridges**(void){

time\_disc.**clear**(); time\_up.**clear**(); visited.**clear**(); from.**clear**();

visited.**resize**(N+3,0); time\_disc.**resize**(N+3,0); time\_up.**resize**(N+3,0); from.**resize**(N+3,0);

from[1]=1; DFStime=0;

**DFS\_Visit**(1);

}

int **IsBridge**(int v\_lo, int v\_high) {

**if** (v\_high!=from[v\_lo]) **return** 0;

**return** ( time\_disc[v\_lo]==time\_up[v\_lo] );

}

void **ContractGraph**(void){

vector<int> **color**(N+3,0);

int colors=1;

color[1]=1;

list<int> Q;

Q.**clear**(); Q.**push\_back**(1);

**while** (!Q.**empty**()) {

int where=Q.**front**(); Q.**pop\_front**();

**for** (list<int>::iterator it=G[where].**begin**(); it!=G[where].**end**(); it++) **if** (!color[\*it]) {

**if** (**IsBridge**(\*it,where)) color[\*it]=++colors; **else** color[\*it]=color[where];

visited[\*it]=1; Q.**push\_back**(\*it);

}

}

G2.**clear**(); G2.**resize**(N+3);

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

**for** (list<int>::iterator it=G[i].**begin**(); it!=G[i].**end**(); it++)

G2[color[i]].**push\_back**(color[\*it]);

}

int **GrundyValue**(int v){

int loops=0,gv=0;

**if** (GV[v]!=-1) **return** GV[v]; GV[v]=1000000000;

**for** (list<int>::iterator start=G2[v].**begin**(); start!=G2[v].**end**(); start++) {

**if** ((\*start)==v) loops++; **else** **if** (GV[\*start]!=1000000000) gv^=(1+**GrundyValue**(\*start));

}

loops/=2; **if** (loops%2) gv^=1;

**return** GV[v]=gv;

}

int **main**(void){

int v1,v2;

*//freopen("g1.in","r",stdin);*

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

cin >> Cases;

**while** (Cases--) {

*// read graph dimensions*

cin >> N >> M;

*// read the graph*

G.**clear**(); G.**resize**(N+3);

**for** (int i=0;i<M;i++) { cin >> v1 >> v2; G[v1].**push\_back**(v2); G[v2].**push\_back**(v1); }

*// collapse all circuits in the graph*

**FindBridges**();

**ContractGraph**();

*// compute the SG value*

GV.**clear**(); **for** (int i=0;i<=N;i++) GV.**push\_back**(-1);

int result=**GrundyValue**(1);

**if** (result) cout << "Alice\n"; **else** cout << "Bob\n"; *//cout << result << "\n";*

}

**return** 0;

}

**Red\_blue Hacken Bush (stalk Only):**

*/\**

*problem: codechef (chef game) .*

*Algo : red-blue hackenbush.*

*\*/*

#define MAXN 55

**typedef** long long int64;

*/\**

*Problem can be reduced to red-black hackenbush*

<http://en.wikipedia.org/wiki/Hackenbush>

*Each pile represent a hackenbush stalk*

*Game value cooresponding to hackenbush stalk is easy to find.*

*Please refer here :* <http://www.geometer.org/mathcircles/hackenbush.pdf.>

*For hackebush games value of two disjoint game is equal to sum of individual game value.*

*(*<http://www-math.mit.edu/~rstan/transparencies/games.pdf>*)*

*\*/*

int t,n,tcase;

int arr[MAXN];

int64 **calculate**(){

int64 res = 0; int64 value = 1LL<<48;

res = (arr[0]%2==0)?value:-value;

bool is\_changed = **false**;

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

**assert**(arr[i]!=arr[i-1]);

**if**(arr[i]%2 != arr[i-1]%2){

is\_changed = **true**;

}

**if**(is\_changed) value /= 2;

res += (arr[i]%2==0)?value:-value;

}

**return** res;

}

int **main**(){

**for**(**scanf**("%d",&tcase); tcase; tcase-=1){

**scanf**("%d",&t);

int64 res = 0;

**for**(int i=0; i<t; ++i){

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

**for**(int j=0; j<n; ++j) **scanf**("%d",&arr[j]);

**sort**(arr,arr+n);

res += **calculate**();

}

**if**(res > 0 ) **printf**("FIRST\n");

**else** **if**(res < 0 ) **printf**("SECOND\n");

**else** **printf**("DON'T PLAY\n");

}

**return** 0;

}

**Matrix**

**Gaussian Elimination :**

*Problem : LOJ 1151 - Snakes and Ladders*

#define maxm 110

int n,m;

int pos[maxm];

*///Gaussian Elimination...............................................*

#define **eps** (1e-9)

#define **iseq**(a,b) (**fabs**(a-b)<eps)

*// a1x1+a2x2+.....=b1 .....*

double a[maxm][maxm],b[maxm],x[maxm];

void **gauss**(int r,int c){

int i,j,k,l;

double val;

i=j=0;

**while**(i<r&&j<c){

**for**(k=i;k<r;k++){

**if**(**iseq**(a[k][j],0.0)) **continue**;

**for**(l=j;l<c;l++){

**swap**(a[i][l],a[k][l]);

}

**swap**(b[i],b[k]);

**break**;

}

**if**(k==r){

j++; **continue**;

}

*// Making jth col of every row from (i+1)th to rth row into zero........*

**for**(k=i+1;k<r;k++){

val=a[k][j]/a[i][j];

**for**(l=j;l<c;l++){

a[k][l]-=(a[i][l]\*val);

}

b[k]-=(b[i]\*val);

}

i++; j++;

}

*/// Additional information.............*

*/\**

*rep(k,i,r)*

*{*

*rep(j,0,c) if(!(fabs(a[k][j])<eps)) goto stop ;*

*if(!(fabs(b[k])<eps)) return -1 ; // no solution*

*stop : ;*

*}*

*if(i>c) return -1 ; // no solution*

*if(i==c) return 0 ; // unique solution*

*if(i<c) return 1 ; // multiple solution*

*\*/*

*/// ................*

**for**(i=c-1;i>=0;i--){

x[i]=b[i];

**for**(k=i+1;k<c;k++){

x[i]-=(a[i][k]\*x[k]);

}

**if**(!**iseq**(a[i][i],0.0)) x[i]/=a[i][i];

}

}

*/// Gaussian Elimination Finish.....................*

int **main**(){

int i,j,k,l,test,t=1;

**scanf**("%d",&test);

**while**(test--){

**for**(i=0;i<=100;i++){

pos[i]=i;

}

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

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

**scanf**("%d %d",&k,&l);

k--; l--;

pos[k]=l;

}

n=100;

**memset**(a,0.0,**sizeof**(a));

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

a[i][i]=1.0;

**if**(pos[i]!=i){

a[i][pos[i]]=-1.0;

b[i]=0.0;

**continue**;

}

**if**(i==n-1){ b[i]=0; **continue**; }

**else** b[i]=1;

*// prob= probabilty.........*

double prob=(double) 1.0/ (double) 6.0;

**for**(j=1;j<=6;j++){

k=(i+j);

**if**(k>99) k=i;

k=pos[k];

a[i][k]-=(prob);

}

}

**gauss**(n,n);

**printf**("Case %d: %.8lf\n",t++,x[0]);

}

**return** 0;

}

**Memorization Technique in Matrix Expo :**

**memcpy**(mem[0],base,**sizeof**(base));

**for**(i=1;i<=62;i++){

**cal**(mem[i-1],mem[i-1],mem[i]);

}

void **exp**(ii r[4][4],ii n){

ii b[4][4];

**memcpy**(r,unit,**sizeof**(unit));

**memcpy**(b,base,**sizeof**(base));

int j=0;

**while**(n>0){

**if**(n%2==1) **cal**(r,mem[j],r);

n/=2; j++;

*//cal(b,b);*

}

}

**Primality Test:**

*/\* this function calculates (a\*b)%c taking into account that a\*b might overflow \*/*

ii **mulmod**(ii a,ii b,ii c){

ii x = 0,y=a%c;

**while**(b > 0){

**if**(b%2 == 1){

x = (x+y)%c;

}

y = (y\*2)%c;

b /= 2;

}

**return** x%c;}

*/\* Miller-Rabin primality test, iteration signifies the accuracy of the test \*/*

bool **Miller**(long long p,int iteration){

**if**(p<2){

**return** **false**;

}

**if**(p!=2 && p%2==0){

**return** **false**;

}

long long s=p-1;

**while**(s%2==0){

s/=2;

}

**for**(int i=0;i<iteration;i++){

long long a=**rand**()%(p-1)+1,temp=s;

long long mod=**big\_mod**(a,temp,p);

**while**(temp!=p-1 && mod!=1 && mod!=p-1){

mod=**mulmod**(mod,mod,p);

temp \*= 2;

}

**if**(mod!=p-1 && temp%2==0){

**return** **false**;

}

}

**return** **true**;

}

**Extended Euclid :**

*/\**

*Problem : LOJ 1306 (Solutions to an Equation ).*

*Algo : Extended Euclid ( Number of solution of a Linear Diaphontine equation in a given range ).*

*\*/*

*// Extended Euclid .....*

**struct** node{

ii x,y,g;

**node**(){};

**node**(ii xx,ii yy,ii gg){ x=xx; y=yy; g=gg;};

};

*// ax+by=g where g=gcd(a,b)....*

node **euclid**(ii a,ii b){

**if**(!b) **return** **node**(1,0,a);

node r=**euclid**(b,a%b);

**return** **node**(r.y,r.x-(a/b)\*r.y,r.g);

}

*//............//*

ii A,B,C,xl,xh,yl,yh;

ii **find\_lo**(ii x0,ii y0,ii ag,ii bg);

int **valid\_lo**(ii x0,ii t,ii bg,ii lo,ii hi);

ii **find\_hi**(ii x0,ii y0,ii ag,ii bg);

int **valid\_hi**(ii x0,ii t,ii bg,ii lo,ii hi);

ii **common**(ii a,ii b,ii c,ii d){

**if**(b<c || d<a) **return** 0;

**if**(a>=c && b<=d) **return** (b-a+1);

**if**(c>=a && d<=b) **return** (d-c+1);

**if**(b>=c && a<=c) **return** (b-c+1);

**if**(a<=d && b>d) **return** (d-a+1);

**return** 0;

}

ii **find\_ans**(){

node piv=**euclid**(A,B);

**if**(!piv.g){

**if**(C) **return** 0;

**return** (xh-xl+1)\*(yh-yl+1);

}

**if**(C%piv.g) **return** 0;

ii x0=piv.x,y0=piv.y;

x0\*=(C/piv.g); y0\*=(C/piv.g);

ii ag=A/piv.g,bg=B/piv.g;

*// x= x0 - t\*bg , y= y0 + t\*ag;*

ii lo1=**find\_lo**(x0,bg,xl,xh);

ii lo2=**find\_lo**(y0,-ag,yl,yh);

ii hi1=**find\_hi**(x0,bg,xl,xh);

ii hi2=**find\_hi**(y0,-ag,yl,yh);

**return** **common**(lo1,hi1,lo2,hi2);

}

**scanf**("%lld %lld %lld %lld %lld %lld %lld",&A,&B,&C,&xl,&xh,&yl,&yh);

C=-C;

**printf**("Case %d: %lld\n",t++,**find\_ans**());

ii **find\_lo**(ii x0,ii bg,ii lox,ii hix){

*// x= x0 - t\*bg , y= y0 + t\*ag;*

ii lo=-inf,hi=inf;

ii mid;

**while**(lo<hi){

mid=lo+hi; mid/=2;

**if**(**valid\_lo**(x0,mid,bg,lox,hix)){

**if**(hi==mid){

**if**(**valid\_lo**(x0,mid-1,bg,lox,hix)) **return** mid-1;

**return** mid;

}

hi=mid;

}

**else**{

lo=mid+1;

}

}

**return** hi;

}

ii **find\_hi**(ii x0,ii bg,ii lox,ii hix){

*// x= x0 - t\*bg , y= y0 + t\*ag;*

ii lo=-inf,hi=inf;

ii mid;

**while**(lo<hi){

mid=lo+hi; mid/=2;

**if**(**valid\_hi**(x0,mid,bg,lox,hix)){

**if**(lo==mid){

**if**(**valid\_hi**(x0,mid+1,bg,lox,hix)) **return** mid+1;

**return** mid;

}

lo=mid;

}

**else**{

hi=mid-1;

}

}

**return** lo;

}

int **valid\_lo**(ii x0,ii t,ii bg,ii lo,ii hi){

*// check increasing .....*

**if**(bg<0){

**if**(x0-(t\*bg)<lo) **return** 0;

**return** 1;

}

**else**{

**if**(x0-(t\*bg)>hi) **return** 0;

**return** 1;

}

}

int **valid\_hi**(ii x0,ii t,ii bg,ii lo,ii hi){

*// check increasing .....*

**if**(bg<0){

**if**(x0-(t\*bg)>hi) **return** 0;

**return** 1;

}

**else**{

**if**(x0-(t\*bg)<lo) **return** 0;

**return** 1;

}

}

**Heavy-Light Decomposition :**

/\*

Problem : Spoj – Query on a tree

\*/

#define maxm 200100

#define lg\_maxm 20

struct tree{

int mx\_cost;

};

tree seg\_T[4\*maxm];

int n,m;

vector<int>G[maxm],W[maxm],edge\_ind[maxm];

int L[maxm],T[maxm],P[maxm][lg\_maxm],subtree\_size[maxm];

int edge[maxm],ptr;

int chain\_head[maxm],chain\_ind[maxm],chain\_no;

int pos\_in\_base[maxm],base\_arr[maxm];

// fixing parent,size and level

void dfs(int s,int pre,int lev){

T[s]=pre;

L[s]=lev;

subtree\_size[s]=1;

for(int i=0;i<G[s].size();i++){

if(G[s][i]==pre) continue;

edge[edge\_ind[s][i]]=G[s][i];

dfs(G[s][i],s,lev+1);

subtree\_size[s]+=subtree\_size[G[s][i]];

}

}

// Updating sparse table for lca . . .

void init\_sparse(){

int i,j;

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

for(j=0;(1<<j)<n;j++){

P[i][j]=-1;

}

}

// the first ancestor ..

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

P[i][0]=T[i];

}

// sparse table ..

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

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

if(P[i][j-1]!=-1){

P[i][j]=P[P[i][j-1]][j-1];

}

}

}

}

/\*

\* Actual HL-Decomposition part

\* Initially all entries of chainHead[] are set to -1.

\* So when ever a new chain is started, chain head is correctly assigned.

\* As we add a new node to chain, we will note its position in the baseArray.

\* In the first for loop we find the child node which has maximum sub-tree size.

\* The following if condition is failed for leaf nodes.

\* When the if condition passes, we expand the chain to special child.

\* In the second for loop we recursively call the function on all normal nodes.

\* chainNo++ ensures that we are creating a new chain for each normal child.

\*/

void heavy\_light(int s,int pre,int curr\_cost){

if(chain\_head[chain\_no]==-1){

chain\_head[chain\_no]=s; // Assign chain head

}

chain\_ind[s]=chain\_no;

pos\_in\_base[s]=++ptr; // Position of this node in baseArray which we will use in Segtree

base\_arr[ptr]=curr\_cost;

int heavy\_child=-1,heavy\_cost=0,heavy\_size=0,i;

// Loop to find heavy child

for(i=0;i<G[s].size();i++){

if(G[s][i]==pre) continue;

if(subtree\_size[G[s][i]]>heavy\_size){

heavy\_size=subtree\_size[G[s][i]];

heavy\_child=G[s][i];

heavy\_cost=W[s][i];

}

}

if(heavy\_child!=-1){

// Expand the chain

heavy\_light(heavy\_child,s,heavy\_cost);

}

for(i=0;i<G[s].size();i++){

if(G[s][i]==pre || G[s][i]==heavy\_child) continue;

// light node . . . .

chain\_no++;

heavy\_light(G[s][i],s,W[s][i]);

}

}

void init\_segtree(int node,int b,int e){

if(b>e) return ;

if(b==e){

seg\_T[node].mx\_cost=base\_arr[b];

return ;

}

int left=node<<1,right=left+1,mid=b+e;

mid/=2;

init\_segtree(left,b,mid);

init\_segtree(right,mid+1,e);

seg\_T[node].mx\_cost=maxi(seg\_T[left].mx\_cost,seg\_T[right].mx\_cost);

}

int seg\_query(int node,int b,int e,int k,int l){

if(b>e) return 0;

if(b==k && e==l) return seg\_T[node].mx\_cost;

int left=node<<1,right=left+1,mid=b+e;

mid/=2;

if(l<=mid) return seg\_query(left,b,mid,k,l);

else if(k>mid) return seg\_query(right,mid+1,e,k,l);

else{

return maxi(seg\_query(left,b,mid,k,mid),seg\_query(right,mid+1,e,mid+1,l));

}

}

void seg\_update(int node,int b,int e,int ind,int v){

if(b>e) return ;

if(b==e){

seg\_T[node].mx\_cost=v;

return ;

}

int left=node<<1,right=left+1,mid=b+e;

mid/=2;

if(ind<=mid) seg\_update(left,b,mid,ind,v);

else seg\_update(right,mid+1,e,ind,v);

seg\_T[node].mx\_cost=maxi(seg\_T[left].mx\_cost,seg\_T[right].mx\_cost);

}

int lca(int p,int q){

int log, i;

//if p is situated on a higher level than q then we swap them

if (L[p] < L[q])

swap(p,q);

//we compute the value of [log(L[p)]

for (log = 1; 1 << log <= L[p]; log++);

log--;

//we find the ancestor of node p situated on the same level

//with q using the values in P

for (i = log; i >= 0; i--)

if (L[p] - (1 << i) >= L[q])

p = P[p][i];

if (p == q)

return p;

//we compute LCA(p, q) using the values in P

for (i = log; i >= 0; i--)

if (P[p][i] != -1 && P[p][i] != P[q][i])

p = P[p][i], q = P[q][i];

return T[p];

}

/\*

\* query\_up:

\* It takes two nodes u and lc, condition is that lc is an ancestor of u

\* We query the chain in which u is present till chain head, then move to next chain up

\* We do that way till u and lc are in the same chain, we query for that part of chain and break

\*/

int query\_up(int u,int lc){

if(u==lc) return 0;

int u\_chain,lc\_chain,ret=-1;

lc\_chain=chain\_ind[lc];

while(1){

if(u==lc) break;

u\_chain=chain\_ind[u];

if(u\_chain==lc\_chain){

// Both u and lc are in the same chain, so we need to query from u to lc, update ret and break.

// We break because we came from u up till v, we are done

ret=maxi(ret,seg\_query(1,1,ptr,pos\_in\_base[lc]+1,pos\_in\_base[u]));

return ret;

}

ret=maxi(ret,seg\_query(1,1,ptr,pos\_in\_base[chain\_head[u\_chain]],pos\_in\_base[u]));

// Above is call to segment tree query function. We do from chainHead of u till u. That is the whole chain from

// start till head. We then update the answer

u=chain\_head[u\_chain]; // move u to u's chainHead

u=T[u]; //Then move to its parent, that means we changed chains

}

return ret;

}

int query(int u,int v){

int lc=lca(u,v);

int ret=maxi(query\_up(u,lc),query\_up(v,lc));

return ret;

}

void update(int u,int v){

seg\_update(1,1,ptr,pos\_in\_base[u],v);

}

int main(){

int i,j,k,l,test,t=1;

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

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

scanf("%d",&test);

while(test--){

scanf("%d",&n);

// init\_all

ptr=0,chain\_no=1;

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

G[i].clear();

W[i].clear();

edge\_ind[i].clear();

chain\_head[i]=-1;

}

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

scanf("%d %d %d",&k,&l,&j);

G[k].push\_back(l);

W[k].push\_back(j);

edge\_ind[k].push\_back(i);

G[l].push\_back(k);

W[l].push\_back(j);

edge\_ind[l].push\_back(i);

}

// init . . .

dfs(1,1,1);

init\_sparse();

heavy\_light(1,1,0);

// seg-tree

init\_segtree(1,1,ptr);

char qs[10];

int u,v;

while(1){

scanf("%s",qs);

if(qs[0]=='D') break;

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

if(qs[0]=='Q'){

printf("%d\n",query(u,v));

}

else{

u=edge[u];

update(u,v);

}

}

}

return 0;}

**Aho-corasick :**

*/\**

*Problem : Beaver (Codeforces Round 71 - problem C ).*

*Algo : Aho corasick , DP , Trie*

*\*/*

#define maxm 100100

#define **inf** (1<<29)

int **maxi**(int a,int b){

**if**(a>b) **return** a;

**return** b;

}

int **mini**(int a,int b){

**if**(a<b) **return** a;

**return** b;

}

*/////\*\*\*\* Trie + Aho Corasick \*\*\*\*\*\*\*\*///////*

*// maxc= query...*

#define maxc 15

*// maxl = length of query string...*

#define maxl 20

*// maxn =required trie node ....*

#define **maxn** ((maxc\*maxl)+10)

#define cn 64

**struct** trie{

*//vector<int>v; // for keeping track of patterns ends here ...*

int edges[cn+3],ind;

int pat\_no; *// highest lenght pattern ends at this node...*

int pat\_len; *// minimum lenght pattern ends at this node...*

};

trie Tri[maxn],root;

int len[maxc]; *// len[i]= length of pattern i ...*

int tot,f[maxn],n,pos[maxc]; *// f=failure , pos[i]=position of ith pattern in trie node .*

int c[maxn]; *// c[i] = (number of occurence) count of ith node of trie in string s .*

int fin[maxm]; *// fin[i] = minimum lenght of pattern finish at pos i of string s*

int **getid**(char ch){

**if**(ch>='a' && ch<='z') **return** ch-'a';

**else** **if**(ch>='A' && ch<='Z') **return** ch-'A'+26;

**else** **if**(ch>='0' && ch<='9') **return** ch-'0'+52;

**if**(ch=='\_') **return** cn-1;

**return** ch-'a';

}

void **init**(trie \*a,int ind){

a->ind=ind;

a->pat\_no=0;

a->pat\_len=inf;

*//a->v.clear();*

**memset**(a->edges,-1,**sizeof**(a->edges));

}

void **add**(trie \*a,char \*s,int ind){

int i,l,id;

l=**strlen**(s);

**for**(i=0;i<=l;i++){

**if**(i==l){

pos[ind]=a->ind;

a->pat\_no=ind;

a->pat\_len=**mini**(a->pat\_len,len[ind]);

*//a->v.push\_back(ind);*

**continue**;

}

id=**getid**(s[i]);

**if**(a->edges[id]==-1){

a->edges[id]=tot;

**init**(&Tri[a->edges[id]],tot++);

}

a=&Tri[a->edges[id]];

}

}

void **build**(){

int i,j,piv;

trie \*a=&Tri[0];

**for**(i=0;i<=cn;i++){

**if**(a->edges[i]==-1) a->edges[i]=0;

}

*// Failure Function .........*

queue<int>q;

**for**(i=0;i<=cn;i++){

**if**(a->edges[i]){

f[a->edges[i]]=0;

q.**push**(a->edges[i]);

}

}

**while**(!q.**empty**()){

int state=q.**front**(); q.**pop**();

a=&Tri[state];

*//sort(a->v.begin(),a->v.end());*

*//unique(a->v.begin(),a->v.end());*

**for**(i=0;i<=cn;i++){

**if**(a->edges[i]==-1) **continue**;

int failure=f[state];

**while**(Tri[failure].edges[i]==-1){

failure=f[failure];

}

failure=Tri[failure].edges[i];

piv=Tri[state].edges[i];

f[piv]=failure;

Tri[piv].pat\_len=**mini**(Tri[piv].pat\_len,Tri[failure].pat\_len);

*/\**

*trie \*a1=&Tri[failure];*

*trie \*a2=&Tri[piv];*

*for (int ind=0; ind<a1->v.size(); ind++){*

*a2->v.push\_back(a1->v[ind]);*

*}*

*\*/*

q.**push**(piv);

}

}

}

void **match**(char \*s);

*//////\*\*\*\*\*\*\*\*\*\*\*\*\*\* END \*\*\*\*\*\*\*\*\*\*\*\*\*/////*

char s[maxm];

char pat[maxl];

*// Problem depenedent.....*

int can[maxm]; *// can[i] = minimum index of string s from pos i for which it is valid ...*

int dp[maxm]; *// dp[i] = store minimum index for which string S obtaining from s[dp[i]] to s[i] is valid .*

int **main**(){

int i,j,k,l,test,t=1;

**scanf**("%s",s);

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

*//Init.......*

c[0]=0;

root=Tri[0];

**init**(&Tri[0],tot++);

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

**scanf**("%s",pat);

len[i]=**strlen**(pat);

*//printf("%s\n",pat);*

**add**(&Tri[0],pat,i);

}

**build**(); // building automata

int m;

**match**(s); // match string

queue<int>q;

vector<int>v;

q.**push**( 0 );

**while**( !q.**empty**()){

int now = q.**front**();

q.**pop**();

v.**push\_back**(now);

**for**( i=0;i<=cn;i++ ){

**if**( Tri[now].edges[i]!=-1 && Tri[now].edges[i]!=0 ) q.**push**(Tri[now].edges[i]);

}

}

**for**( i=v.**size**()-1;i>=0;i-- ){

c[f[v[i]]] += c[v[i]];

}

*/\**

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

*printf("%d\n",c[pos[i]]);*

*}*

*for(i=0;s[i];i++){*

*//if(fin[i]==inf) fin[i]=-1;*

*printf("%2d ",i);*

*}*

*puts("");*

*\*/*

int ans=0,mark=0,ans1;

**for**(i=0;s[i];i++){

*//if(fin[i]==inf) fin[i]=-1;*

can[i]=**maxi**(i-fin[i]+2,0);

dp[i]=can[i];

**if**(i) dp[i]=**maxi**(dp[i],dp[i-1]);

ans1=i-dp[i]+1;

**if**(ans1>ans){

ans=ans1;

mark=dp[i];

}

*//printf("%2d ",can[i]);*

}

*//puts("");*

*/\**

*for(i=0;s[i];i++){*

*//if(fin[i]==inf) fin[i]=-1;*

*printf("%2d ",dp[i]);*

*}*

*puts("");*

*for(i=0;s[i];i++){*

*//if(fin[i]==inf) fin[i]=-1;*

*printf("%2c ",s[i]);*

*}*

*puts("");*

*\*/*

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

**return** 0;

}

int **find\_next**(int curr,char ch){

int id=**getid**(ch);

*//printf("curr =%d %c-%d\n",curr,ch,id);*

**while**(Tri[curr].edges[id]==-1) curr=f[curr];

*//printf("curr =%d %c-%d %d\n",curr,ch,id,Tri[curr].edges[id]);*

**return** Tri[curr].edges[id];

}

void **match**(char \*s){

int i,j,l;

l=**strlen**(s);

int curr=0;

**for**(i=0;i<l;i++){

curr=**find\_next**(curr,s[i]);

c[curr]++;

fin[i]=Tri[curr].pat\_len;

*/\**

*trie \*a=&Tri[curr];*

*for (it=a->v.begin(); it!=a->v.end(); ++it){*

*c[\*it]++;*

*}*

*\*/*

}

}

**Manachar’s algorithm:**

*/\**

*Manacher algorithm implementation.*

*Application, largest palindromic substring, largest palindromic suffix*

*\*/*

int lengths[MAX<<1];

int **manacher**(char \*buff, int len) {

int i, k, pallen, found, d, j, s, e;

k = pallen = 0;

**for**(i = 0; i < len; ) {

**if**(i > pallen && buff[i-pallen-1] == buff[i]) {

pallen += 2, i++;

**continue**;

}

lengths[k++] = pallen;

s = k - 2, e = s - pallen, found = 0;

**for**(j = s; j > e; j--) {

d = j - e - 1;

**if**(lengths[j] == d) {

pallen = d;

found = 1;

**break**;

}

lengths[k++] = (d < lengths[j]? d : lengths[j]);

}

**if**(!found) { pallen = 1; i++; }

}

lengths[k++] = pallen;

**return** lengths[k-1];

}

**Shank’s Algorithm:**

This algorithm finds **x** ( 0 <= x <= p - 2 ) **for** the equation

b = ax mod p where b, a, p are known

Using the fact that x can be expressed as jm + i, where 0 <= i <= m – 1, 0 <= j < p/m, and m = **ceil**(**sqrt**( p - 1 ))

So, the equation can be written as

b = amj+i mod p

b = amj ai mod p

ba-i = amj mod p

If two lists of ordered **pairs** (i, ba-i) **and** (j, amj), ordered by their second components are built, then it is possible to find one pair from each list that have equal second components. Then x = mj + i, where i and j are the first elements of the matching pairs.

**Code:**

*/\**

*Shanks baby step giant step - discrete logarithm algorithm*

*for the equation: b = a^x % p where a, b, p known, finds x*

*works only when p is an odd prime*

*\*/*

int **shank**(int a, int b, int p) {

int i, j, m;

long long c, aj, ami;

map< long long, int > M;

map< long long, int > :: iterator it;

m = (int)**ceil**(**sqrt**((double)(p)));

M.**insert**(**make\_pair**(1, 0));

**for**(j = 1, aj = 1; j < m; j++) {

aj = (aj \* a) % p;

M.**insert**(**make\_pair**(aj, j));

}

ami = **modexp**(**modinv**(a, p), m, p);

**for**(c = b, i = 0; i < m; i++) {

it = M.**find**(c);

**if**(it != M.**end**()) **return** i \* m + it->second;

c = (c \* ami) % p;

}

**return** 0;

}

**Negative Base:**

string **negaBase**(int n,int b){

int i,tmp;

string a;

**for**(i=0;n;i++){

tmp=n%b; n=n/b;

**if**(tmp<0) { tmp+= (-b), n++; }

a+='0'+tmp;

}

**for**(n=0;n<(i/2);n++) **swap**(a[n],a[i -n - 1]);

**if**(i) **return** a;

**return** "0";

}

**Joseph:**

int **joseph**(int n,int k){

**if**(n==1) **return** 0;

**return** ((**joseph**(n-1,k)+k)%n);

}

**Meet in the middle + Ternary Mask:**

struct ternary{

ii pow3[maxm];

void init(){

init(maxm-1);

}

void init(int n){

pow3[0]=1;

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

pow3[i]=pow3[i-1]\*3;

}

}

int get\_bit(int mask,int k){

ii tmp=mask; tmp/=pow3[k];

return (tmp%3);

}

int set\_bit(int mask,int k,int v){

ii tmp=mask;

tmp/=pow3[k];

tmp%=3;

mask-=(tmp\*pow3[k]);

mask+=(v\*pow3[k]);

return mask;

}

};

ternary t\_mask;

int n,req;

int a[maxm],b[maxm];

set<int>can\_set;

int build(int mask,int a[],int n,int flag){

int ret=0;

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

int bit\_val=t\_mask.get\_bit(mask,i);

for(int j=1;j<=bit\_val;j++){

ret+=a[i];

if(ret>req) return -1;

}

}

if(flag) can\_set.insert(ret);

return ret;

}

int main(){

int i,j,k,l,test,t=1;

t\_mask.init();

scanf("%d",&test);

while(test--){

can\_set.clear();

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

int n1=n/2,n2=n-n1;

for(i=0;i<n1;i++){

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

}

for(i=n1,j=0;i<n;i++,j++){

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

}

int tot=0;

for(i=0;i<t\_mask.pow3[n1];i++){

build(i,a,n1,1);

}

bool soln\_found=false;

for(i=0;i<t\_mask.pow3[n2];i++){

int now=build(i,b,n2,0);

if(now==-1) continue;

now=req-now;

if(can\_set.find(now)!=can\_set.end()){

soln\_found=true;

break;

}

}

printf("Case %d: ",t++);

if(soln\_found==true){

printf("Yes\n");

}

else{

puts("No");

}

}

return 0;

}

**Treap :**

/\*

Algo : Treap (Balanced BST).

Problem : Spoj - Yet another range difference query.

\*/

// TREAP >>>>>>>>>>>>>>>>>>>>>>>>>

typedef int treap\_type;

struct node{

treap\_type value,min\_val,max\_val,diff;

ii priority;

int cnt;

node \*left,\*right;

node(){}

node(treap\_type \_value){

cnt=1;

value=min\_val=max\_val=\_value;

diff=inf;

priority=rand();

left=right=NULL;

}

};

// 1-based . . . . . . . . . .

struct treap{

node \*root;

void fix(node \* &t){

if(t==NULL) return ;

t->cnt=get\_count(t->left)+get\_count(t->right)+1;

t->diff=inf;

if(t->left){

t->min\_val=t->left->min\_val;

t->diff=mini(t->left->diff,t->value - t->left->max\_val);

}

else t->min\_val=t->value;

if(t->right){

t->max\_val=t->right->max\_val;

t->diff=mini( t->diff,mini( t->right->diff , t->right->min\_val- t->value) );

}

else t->max\_val=t->value;

}

inline int get\_count(node\* t){

return t ? t->cnt : 0;

}

inline void left\_rotate(node\* &t){

node\* tmp = t->left;

t->left = tmp->right;

tmp->right = t;

t = tmp;

}

inline void right\_rotate(node\* &t){

node\* tmp = t->right;

t->right = tmp->left;

tmp->left = t;

t = tmp;

}

bool insert(node \* &t,treap\_type value){

if(t==NULL){

t=new node(value);

fix(t);

return true;

}

if(t->value==value) return false;

bool ret;

if(value < t->value) ret=insert(t->left,value);

else ret=insert(t->right,value);

if(t->left && t->left->priority > t->priority){

left\_rotate(t);

}

else if(t->right && t->right->priority > t->priority){

right\_rotate(t);

}

if(t->left) fix(t->left);

if(t->right) fix(t->right);

fix(t);

return ret;

}

bool insert(treap\_type value){

return insert(root,value);

}

inline ii get\_priority(node\* t){

return t ? t->priority : -1;

}

bool erase(node\* &t, treap\_type val){

if(!t) return false;

bool ret;

if(t->value != val){

ret=erase(val < t->value ? t->left : t->right, val);

}

else{

if(!t->left && !t->right){

delete t;

t = NULL;

}else{

if(get\_priority(t->left) < get\_priority(t->right))

right\_rotate(t);

else

left\_rotate(t);

ret=erase(t, val);

}

}

if(t){

if(t->left) fix(t->left);

if(t->right) fix(t->right);

fix(t);

}

return ret;

}

bool erase(treap\_type value){

return erase(root,value);

}

node\* find(node \* t,int value){

if(t==NULL) return NULL;

node \*ret;

if(value > t->value){

ret=find(t->right,value);

if(ret==NULL) return t;

return ret;

}

else if(value==t->value){

return t;

}

else{

return find(t->left,value);

}

}

node\* find(int value){

return find(root,value);

}

treap\_type find\_max(int beg,int end){

if(beg>end) swap(beg,end);

return 0;

}

treap\_type find\_min(int beg,int end){

if(beg>end) swap(beg,end);

return 0;

}

};

// END >>>>>>>>>>>>>>>>>>

treap tree;

ii find\_min(node \*t,int left,int beg,int end){

if(t==NULL) return inf;

ii r\_left=left;

ii r\_right=left+t->cnt-1;

ii sz=tree.get\_count(t->left)+1;

ii curr\_ind=left+tree.get\_count(t->left);

if(beg<=r\_left && end>=r\_right){

return t->diff;

}

ii ret=inf;

if(beg>curr\_ind) ret=mini(ret,find\_min(t->right,curr\_ind+1,beg,end));

else if(end<curr\_ind) ret=mini(ret,find\_min(t->left,left,beg,end));

else ret=mini(mini(find\_min(t->left,left,beg,end)

,find\_min(t->right,curr\_ind+1,beg,end)),ret);

if(curr\_ind>beg && curr\_ind<=end && t->left){

ret=mini(ret,t->value - t->left->max\_val);

}

if(curr\_ind<end && curr\_ind>=beg && t->right){

ret=mini(ret,t->right->min\_val - t->value);

}

return ret;

}

ii find\_min(int k,int l){

if(l<=k) return -1;

return find\_min(tree.root,1,k,l);

}

ii find\_kth(node \*t,int kth){

if(tree.get\_count(t)<kth){

return -1;

}

int sz=tree.get\_count(t->left)+1;

if(sz==kth){

return t->value;

}

if(kth<sz){

return find\_kth(t->left,kth);

}

else{

return find\_kth(t->right,kth-sz);

}

}

ii find\_max(int k,int l){

if(l<=k) return -1;

return find\_kth(tree.root,l)-find\_kth(tree.root,k);

}

int find\_count(node \*t,int value){

if(t==NULL) return 0;

if(value==t->value) return tree.get\_count(t->left);

else if(value > t->value) return tree.get\_count(t->left)

+find\_count(t->right,value)+1;

else return find\_count(t->left,value);

}

int n,m;

char type[5];

int main(){

int i,j,k,l,test,t=1,val;

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

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

scanf("%d",&test);

while(test--){

scanf("%s",type);

//printf("%s\n",type);

if(type[0]=='I'){

scanf("%d",&val);

tree.insert(val);

}

if(type[0]=='D'){

scanf("%d",&val);

tree.erase(val);

}

if(type[0]=='N'){

scanf("%d %d",&k,&l);

printf("%d\n",find\_min(k+1,l+1));

}

if(type[0]=='X'){

scanf("%d %d",&k,&l);

printf("%d\n",find\_max(k+1,l+1));

}

}

return 0;

}

**Suffix Automation :**

/\*

Algo : Suffix-Automation.  
Problem : Codeforces 235c- Cyclical Quest.

\*/

int n;

char s[maxm];

// Suffix-Automation

struct state {

int len, link;

bool suffix;

ii count;

map<char,int> next;

};

const int MAXLEN = maxm+2;

state st[MAXLEN\*2];

int sz, last;

pair<int, int> sorter[MAXLEN \* 2 + 10];

inline void sa\_init() {

sz = last = 0;

st[0].len = 0;

st[0].link = -1;

st[0].count = 0;

st[0].suffix=0;

++sz;

}

inline void sa\_extend (char c) {

int cur = sz++;

st[cur].len = st[last].len + 1;

st[cur].suffix=0;

st[cur].count=1;

int p;

for (p=last; p!=-1 && !st[p].next.count(c); p=st[p].link)

st[p].next[c] = cur;

if (p == -1)

st[cur].link = 0;

else {

int q = st[p].next[c];

if (st[p].len + 1 == st[q].len)

st[cur].link = q;

else {

int clone = sz++;

st[clone].len = st[p].len + 1;

st[clone].next = st[q].next;

st[clone].link = st[q].link;

for (; p!=-1 && st[p].next[c]==q; p=st[p].link)

st[p].next[c] = clone;

st[q].link = st[cur].link = clone;

}

}

last = cur;

}

// Suffix-Automation End. ..

void post\_process(){

int i;

for(i=0;i<sz;i++){

sorter[i]=mp(st[i].len,i);

}

sort(sorter,sorter+sz);

for(i=sz-1;i>=0;i--){

int ind=sorter[i].vv;

st[st[ind].link].count+=st[ind].count;

}

}

vector<pii>ans;

int pre[maxm];

void failure(char \*p){

int i,j,k,l;

l=strlen(p);

pre[1]=0;

k=0;

for(i=2;i<=l;i++){

while(k>0 && p[k]!=p[i-1]) k=pre[k];

if(p[k]==p[i-1]) k++;

pre[i]=k;

}

}

ii cal(char \*s,int lim){

int i;

int curr\_st=0,len=0;

ii ret=0;

failure(s);

/\*for(i=0;s[i];i++){

printf("%d ",pre[i+1]);

}

puts("");

\*/

for(i=0;s[i];i++){

while (curr\_st && !st[curr\_st].next.count(s[i])) {

curr\_st = st[curr\_st].link;

len=st[curr\_st].len;

}

if (st[curr\_st].next.count(s[i])) {

curr\_st = st[curr\_st].next[s[i]];

len++;

}

while(st[st[curr\_st].link].len>=lim){

curr\_st=st[curr\_st].link;

}

if(len>=lim && pre[i+1]<lim){

ret+=st[curr\_st].count;

}

}

return ret;

}

char tmp[maxm];

int main(){

int i,j,k,l;

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

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

scanf("%s",s);

sa\_init();

for(i=0;s[i];i++){

sa\_extend(s[i]);

}

post\_process();

int q;

int len;

scanf("%d",&q);

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

scanf("%s",tmp);

len=strlen(tmp);

strcpy(s,tmp);

for(j=len,k=0;k<len-1;k++,j++){

s[j]=tmp[k];

}

s[j]=0;

//puts(s);

printf("%I64d\n",cal(s,len));

}

return 0;

}

**IDA\*:**

int ida\_star(int puzzle[maxm][maxm]){

int i,j;

node root;

for(i=1;i<=4;i++){

for(j=1;j<=4;j++){

root.puzzle[i][j]=puzzle[i][j];

}

}

curr\_node=root;

int bound=mini(50,heuristic());

solution="";

while(true){

curr\_node=root;

pii zero=find\_pos(root.puzzle,0);

int next\_bound=ida\_search(zero,bound,0);

if(next\_bound<=bound) return 1;

next\_bound=mini(55,next\_bound);

//if(next\_bound<=bound) break;

bound=next\_bound;

}

return 1;

}

int ida\_search(pii pos\_zero,int bound,int d){

int f=heuristic();

if(f+d>bound) return f+d;

if(!f){

if(solution.size()==0 || solution.size()>d+1){

soln[d]=0;

solution=soln;

}

return f;

}

int ret=-1,x,y;

for(int i=0;i<4;i++){

if(d && soln[d-1]==move[3-i]){

continue;

}

x=dirx[i],y=diry[i];

pii pos=pos\_zero;

int ret1=ida\_search(new\_pos,bound,d+1);

if(!ret1) return ret1;

// move

if(ret==-1) ret=ret1;

ret=mini(ret,ret1);

// reverse move

}

return ret;

}

int heuristic(){

int i,j,ret=0;

return ret;

}

**Default Template ( Rashed ):**

#include<stdio.h>

#include<string.h>

#include<math.h>

#include<stdlib.h>

#include<ctype.h>

#include<iostream>

#include<algorithm>

#include<vector>

#include<string>

#include<queue>

#include<stack>

#include<map>

#include<set>

using namespace std;

#define maxm 2010

#define inf (1<<29)

#define ii int

#define pi acos(-1.0)

#define eps 1e-9

#define iseq(a,b) (fabs(a-b)<eps)

#define pii pair<int,int>

#define mp make\_pair

#define uu first

#define vv second

ii on(ii n,ii k){ return (n|(1<<k)); }

ii off(ii n,ii k){ return (n-(n&(1<<k))); }

bool chck(ii n,ii k){ return (n&(1<<k)); }

ii mini(ii a,ii b){ if(a<b) return a; return b; }

ii maxi(ii a,ii b){ if(a>b) return a; return b; }

int n,m;

int main(){

int i,j,k,l,test,t=1;

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

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

scanf("%d",&test);

while(test--){

}

return 0;

}

**HISTOGRAM**

/\*

Finds largest rectangular area in a histogram in O(n)

\*/

i64 calc(int \*ht, int n) {

i64 ret = 0;

int top = 1, st[MAX], i;

ht[0] = st[0] = ht[++n] = 0;

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

while(top > 1 && ht[st[top-1]] >= ht[i]) {

ret = \_max(ret, (i64)ht[st[top-1]]\*(i64)(i - st[top-2]-1));

top--;

}

st[top++] = i;

}

return ret;

}

**KNIGHT DISTANCE ON INF CHESSBOARD**

/\*

NK is the size of grid you want to precalculate

NK/2,NK/2 will be considered origin

Calculates minimum knight distance from 0,0 to x,y

\*/

const int KN = 101;

i64 dk[KN][KN];

int dx[] = {-1, -1, 1, 1, -2, -2, 2, 2};

int dy[] = {-2, 2, -2, 2, -1, 1, -1, 1};

void precalc() {

int x, y, x1, y1, i;

queue< int > Q;

memset(dk, 0x3f, sizeof dk);

x = y = (KN >> 1);

dk[x][y] = 0;

Q.push(x); Q.push(y);

while(!Q.empty()) {

x = Q.front(); Q.pop();

y = Q.front(); Q.pop();

for(i = 0; i < 8; i++) {

x1 = x + dx[i], y1 = y + dy[i];

if(0 <= x1 && x1 < KN && 0 <= y1 && y1 < KN) {

if(dk[x1][y1] > dk[x][y] + 1) {

dk[x1][y1] = dk[x][y] + 1;

Q.push(x1); Q.push(y1);

}

}

}

}

}

i64 knight(i64 x, i64 y) {

i64 step, res = 0;

if(x < y) swap(x, y);

while((x<<1) > KN) {

step = x / 2 / 2; res += step;

x -= step \* 2; y -= step;

if(y < 0) y = ((y % 2) + 2) % 2;

if(x < y) swap(x, y);

}

res += dk[x+(KN>>1)][y+(KN>>1)];

return res;

}

**MINIMUM EXPRESSION ( MINMOVE )**

/\*

Finds alphabetically first representation of a cyclic string in O(length)

\*/

inline int minimumExpression(char \*s) {

int i, j, k, n, len, p, q;

len = n = strlen(s), n <<= 1, i = 0, j = 1, k = 0;

while(i + k < n && j + k < n) {

p = i+k >= len ? s[i+k-len] : s[i+k];

q = j+k >= len ? s[j+k-len] : s[j+k];

if(p == q) k++;

else if(p > q) { i = i+k+1; if(i <= j) i = j+1; k = 0; }

else if(p < q) { j = j+k+1; if(j <= i) j = i+1; k = 0; }

}

return i < j ? i : j;

}

**SHANK’S ALGORITHM**

/\*

Shanks baby step giant step - discrete logarithm algorithm

for the equation: b = a^x % p where a, b, p known, finds x

works only when p is an odd prime

\*/

int shank(int a, int b, int p) {

int i, j, m;

long long c, aj, ami;

map< long long, int > M;

map< long long, int > :: iterator it;

m = (int)ceil(sqrt((double)(p)));

M.insert(make\_pair(1, 0));

for(j = 1, aj = 1; j < m; j++) {

aj = (aj \* a) % p;

M.insert(make\_pair(aj, j));

}

ami = modexp(modinv(a, p), m, p);

for(c = b, i = 0; i < m; i++) {

it = M.find(c);

if(it != M.end()) return i \* m + it->second;

c = (c \* ami) % p;

}

return 0;

}

**EXTENDED EUCLID**

/\*

Implementation of extended euclid algorithm.

Modular inverse requires gcd(a, n) = 1.

\*/

class Euclid {

public:

i64 x, y, d;

Euclid() {}

Euclid(i64 \_x, i64 \_y, i64 \_d) : x(\_x), y(\_y), d(\_d) {}

};

Euclid egcd(i64 a, i64 b) {

if(!b) return Euclid(1, 0, a);

Euclid r = egcd(b, a % b);

return Euclid(r.y, r.x - a / b \* r.y, r.d);

}

i64 modinv(i64 a, i64 n) {

Euclid t = egcd(a, n);

if(t.d > 1) return 0;

i64 r = t.x % n;

return (r < 0 ? r + n : r);

}

**POINT IN CONVEX POLYGON**

/\*

C[] array of points of convex polygon in ccw order,

nc number of points in C, p target points.

returns true if p is inside C (including edge) or false otherwise.

complexity O(lg n)

\*/

inline bool inConvexPoly(point \*C, int nc, const point &p) {

int st = 1, en = nc - 1, mid;

while(en - st > 1) {

mid = (st + en)>>1;

if(triArea2(C[0], C[mid], p) < 0) en = mid;

else st = mid;

}

if(triArea2(C[0], C[st], p) < 0) return false;

if(triArea2(C[st], C[en], p) < 0) return false;

if(triArea2(C[en], C[0], p) < 0) return false;

return true;

}

**CLOSEST PAIR ALGORITHM**

/\*

closestPair(Point \*X, Point \*Y, int n);

X contains the points sorted by x co-ordinate,

Y contains the points sorted by y co-ordinate,

One additional item in Point structure is needed, the original index.

\*/

typedef long long i64;

typedef struct { int x, y, i; } Point;

int flag[MAX];

inline i64 sq(const i64 &x) {

return x\*x;

}

inline i64 sqdist(const Point &a, const Point &b) {

return sq(a.x-b.x) + sq(a.y-b.y);

}

inline i64 closestPair(Point \*X, Point \*Y, int n) {

if(n == 1) return INF;

if(n == 2) return sqdist(X[0], X[1]);

int i, j, k, n1, n2, ns, m = n >> 1;

Point Xm = X[m-1], \*XL, \*XR, \*YL, \*YR, \*YS;

i64 lt, rt, dd, tmp;

XL = new Point[m], YL = new Point[m];

XR = new Point[m+1], YR = new Point[m+1];

YS = new Point[n];

for(i = 0; i < m; i++) XL[i] = X[i], flag[X[i].i] = 0;

for(; i < n; i++) XR[i - m] = X[i], flag[X[i].i] = 1;

for(i = n2 = n1 = 0; i < n; i++) {

if(!flag[Y[i].i]) YL[n1++] = Y[i];

else YR[n2++] = Y[i];

}

lt = closestPair(XL, YL, n1);

rt = closestPair(XR, YR, n2);

dd = min(lt, rt);

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

if(sq(Y[i].x - Xm.x) < dd)

YS[ns++] = Y[i];

for(j = 0; j < ns; j++)

for(k = j + 1; k < ns && sq(YS[k].y - YS[j].y) < dd; k++)

dd = min(dd, sqdist(YS[j], YS[k]));

delete[] XL; delete[] XR;

delete[] YL; delete[] YR;

delete[] YS;

return dd;

}

**TETRAHEDRON ( PYRAMID )**

/\*

Some tetrahedron formulas

\*/

inline double volume(double u, double v, double w, double U, double V, double W) {

double u1,v1,w1;

u1 = v \* v + w \* w - U \* U;

v1 = w \* w + u \* u - V \* V;

w1 = u \* u + v \* v - W \* W;

return sqrt(4.0\*u\*u\*v\*v\*w\*w - u\*u\*u1\*u1 - v\*v\*v1\*v1 - w\*w\*w1\*w1 + u1\*v1\*w1) / 12.0;

}

inline double surface(double a, double b, double c) {

return sqrt((a + b + c) \* (-a + b + c) \* (a - b + c) \* (a + b - c)) / 4.0;

}

inline double insphere(double WX, double WY, double WZ, double XY, double XZ, double YZ) {

double sur, rad;

sur = surface(WX, WY, XY) + surface(WX, XZ, WZ) + surface(WY, YZ, WZ) + surface(XY, XZ, YZ);

rad = volume(WX, WY, WZ, YZ, XZ, XY) \* 3.0 / sur;

return rad;

}

**ARTICULATION POINT**

/\*

G[][]: undirected graph

cut[v] is true if node v is an articulation point / cut-vertex

\*/

vector< int > G[MAX];

int low[MAX], vis[MAX], used[MAX], cut[MAX], dfstime;

void dfs(int u, int par = -1) {

int i, v, child = 0;

used[u] = 1;

vis[u] = low[u] = ++dfstime;

for(i = 0; i < G[u].size(); i++) {

v = G[u][i];

if(v == par) continue;

if(used[v]) low[u] = min(low[u], vis[v]);

else {

child++;

dfs(v, u);

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

if(low[v] >= vis[u]) cut[u] = 1;

}

}

if(par == -1) cut[u] = (child > 1);

}

**BRIDGE**

/\*

G[][]: undirected graph

finds all the bridges in a connected graph and

adds those edges to the Bridges[] vector

\*/

vector< int > G[MAX];

vector< pair< int, int > > Bridges;

int low[MAX], vis[MAX], used[MAX], dfstime;

void dfs(int u, int par) {

int i, v;

used[u] = 1;

vis[u] = low[u] = ++dfstime;

for(i = 0; i < G[u].size(); i++) {

v = G[u][i];

if(v == par) continue;

if(used[v]) low[u] = min(low[u], vis[v]);

else {

dfs(v, u);

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

if(low[v] > vis[u]) Bridges.push\_back(make\_pair(u, v));

}

}

}

**BI CONNECTED COMPONENTS**

/\*

G[][]: undirected graph

Separates bi-connected component by edges.

\*/

vector< int > G[MAX];

stack< pii > S;

int dfstime;

int low[MAX], vis[MAX], used[MAX];

void dfs(int u, int par) {

int v, i, sz = G[u].size();

pii e, curr;

used[u] = 1;

vis[u] = low[u] = ++dfstime;

for(i = 0; i < sz; i++) {

v = G[u][i];

if(v == par) continue;

if(!used[v]) {

S.push(pii(u, v));

dfs(v, u);

if(low[v] >= vis[u]) {

// new component

curr = pii(u, v);

do {

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

// e is an edge in current bcc

} while(e != curr);

}

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

}

else if(vis[v] < vis[u]) {

S.push(pii(u, v));

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

}

}

}

**DINIC**

/\*

max flow (dinitz algorithm)

works on undirected graph

can have loops, multiple edges, cycles

\*/

int src, snk, nNode, nEdge;

int Q[MAXN], fin[MAXN], pro[MAXN], dist[MAXN];

int flow[MAXE], cap[MAXE], next[MAXE], to[MAXE];

inline void init(int \_src, int \_snk, int \_n) {

src = \_src, snk = \_snk, nNode = \_n, nEdge = 0;

SET(fin);

}

inline void add(int u, int v, int \_cap) {

to[nEdge] = v, cap[nEdge] = \_cap, flow[nEdge] = 0;

next[nEdge] = fin[u], fin[u] = nEdge++;

to[nEdge] = u, cap[nEdge] = 0, flow[nEdge] = 0;

next[nEdge] = fin[v], fin[v] = nEdge++;

}

bool bfs() {

int st, en, i, u, v;

SET(dist);

dist[src] = st = en = 0;

Q[en++] = src;

while(st < en) {

u = Q[st++];

for(i=fin[u]; i>=0; i=next[i]) {

v = to[i];

if(flow[i] < cap[i] && dist[v]==-1) {

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

Q[en++] = v;

}

}

}

return dist[snk]!=-1;

}

int dfs(int u, int fl) {

if(u==snk) return fl;

for(int &e=pro[u], v, df; e>=0; e=next[e]) {

v = to[e];

if(flow[e] < cap[e] && dist[v]==dist[u]+1) {

df = dfs(v, min(cap[e]-flow[e], fl));

if(df>0) {

flow[e] += df;

flow[e^1] -= df;

return df;

}

}

}

return 0;

}

i64 dinitz() {

i64 ret = 0;

int df;

while(bfs()) {

for(int i=1; i<=nNode; i++) pro[i] = fin[i];

while(true) {

df = dfs(src, INF);

if(df) ret += (i64)df;

else break;

}

}

return ret;

}

**MIN COST MAX FLOW**

/\*

min cost flow (bellman ford)

works only on directed graphs

handles multiple edges, cycles, loops

\*/

int src, snk, nNode, nEdge;

int fin[MAXN], pre[MAXN], dist[MAXN];

int cap[MAXE], cost[MAXE], next[MAXE], to[MAXE], from[MAXE];

inline void init(int \_src, int \_snk, int nodes) {

SET(fin);

nNode = nodes, nEdge = 0;

src = \_src, snk = \_snk;

}

inline void addEdge(int u, int v, int \_cap, int \_cost) {

from[nEdge] = u, to[nEdge] = v, cap[nEdge] = \_cap, cost[nEdge] = \_cost;

next[nEdge] = fin[u], fin[u] = nEdge++;

from[nEdge] = v, to[nEdge] = u, cap[nEdge] = 0, cost[nEdge] = -(\_cost);

next[nEdge] = fin[v], fin[v] = nEdge++;

}

bool bellman() {

int iter, u, v, i;

bool flag = true;

MEM(dist, 0x7f);

SET(pre);

dist[src] = 0;

for(iter = 1; iter < nNode && flag; iter++) {

flag = false;

for(u = 0; u < nNode; u++) {

for(i = fin[u]; i >= 0; i = next[i]) {

v = to[i];

if(cap[i] && dist[v] > dist[u] + cost[i]) {

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

pre[v] = i;

flag = true;

}

}

}

}

return (dist[snk] < INF);

}

int mcmf(int &fcost) {

int netflow, i, bot, u;

netflow = fcost = 0;

while(bellman()) {

bot = INF;

for(u = pre[snk]; u >= 0; u = pre[from[u]]) bot = min(bot, cap[u]);

for(u = pre[snk]; u >= 0; u = pre[from[u]]) {

cap[u] -= bot;

cap[u^1] += bot;

fcost += bot \* cost[u];

}

netflow += bot;

}

return netflow;

}

**MATCHING N^3**

/\*

G[] is the left-side graph, must be bipartite

match(n): n is the number of nodes in left-side set

and returns the maximum possible matching.

Left[] anf Right[] ar assigned with corresponding matches

\*/

vector < int > G[MAX];

bool visited[MAX];

int Left[MAX], Right[MAX];

bool dfs(int u) {

if(visited[u]) return false;

visited[u] = true;

int len = G[u].size(), i, v;

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

v = G[u][i];

if(Right[v]==-1) {

Right[v] = u, Left[u] = v;

return true;

}

}

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

v = G[u][i];

if(dfs(Right[v])) {

Right[v] = u, Left[u] = v;

return true;

}

}

return false;

}

int match(int n) {

int i, ret = 0;

bool done;

SET(Left); SET(Right);

do {

done = true; CLR(visited);

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

if(Left[i]==-1 && dfs(i)) {

done = false;

}

}

} while(!done);

for(i=0; i<n; i++) ret += (Left[i]!=-1);

return ret;

}

**HopKroft**

/\*

\* Algo :

\* First start a bfs using all the left nodes still not match.

\* Bfs it by checking the right adj nodes of each left node in queue untill in a bfs level 1 or more unmatched right nodes found.

\* Then for each unmatched left node try to find a augment path by dfs. If found then break it to get a bigger matching

\* total number of times dfs will return true is the matching.

\*/

//Hopcroft karp maximum matching algo. Tested in spoj MATCHING

#define mx 100000

#define INF 100000000

//for any node lf is the left match node of that right side node, rt is right match node of left side nodes

//d is the distance array. d[0] is the distance of left ndoes, d[1] for rights

//ls and rs are the left and right side node number respectedly. nodes are 1<=ls,rs type. 0 is a special node

//adj list is the nodes of left graph

int lf[mx],rt[mx],d[mx],ls,rs;

vector < int > g[mx];

bool bfs() {

int i,j,u;

queue < int > q;

for(i=1;i<=ls;i++) {

if(!rt[i]) {

d[i]=0;

q.push(i);

}

else d[i]=INF;

}

d[0]=INF;

while(!q.empty()) {

u=q.front();

q.pop();

j=g[u].size();

for(i=0;i<j;i++) if(d[lf[g[u][i]]]==INF) {

d[lf[g[u][i]]]=d[u]+1;

q.push(lf[g[u][i]]);

}

}

return d[0]!=INF;

}

bool dfs(int u) {

if(!u) return true;

int i,j=g[u].size();

for(i=0;i<j;i++) if(d[lf[g[u][i]]]==(d[u]+1) && dfs(lf[g[u][i]])) {

rt[u]=g[u][i];

lf[g[u][i]]=u;

return true;

}

d[u]=INF;

return false;

}

int HopCropKarp() {

CLR(lf);

CLR(rt);

int r=0,i;

while(bfs()) {

//cout<<"bfs\n";

for(i=1;i<=ls;i++) if(!rt[i] && dfs(i)) r++;

}

return r;

}

int main()

{

int i,j,u,v,m;

cin>>ls>>rs>>m;

while(m--) {

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

g[u].push\_back(v);

}

cout<<HopCropKarp()<<'\n';

return 0;

}

**Heavy light decomposition**

/\*

\* Heavy light decomposition. Divide the edge's of a tree in to heavy or light catagory by this law

\* This code assign weights to nodes and ask distance bitween two nodes dynamically.

\* Its do it by first decomp() it with root 0 to divide edges to heavy and light.

\* Then init() make a bit for each chain and also initialize other veriable.

\* lca() compute LCA, query() return a nodes total sum weight from its chain head; a bit's function.

\* dist() compute distance of a node from root(hardest part to think)

\* update() updates the bit of a node.

\*/

#define mx 30009

vector < int > bit[mx],g[mx];

int ds[mx],skip[mx],parent[mx],wt[mx]; //ds has distance, skip has the node to skip + the bit to handle; -1 means a light parent node

int decomp(int u,int pr,int d) {

ds[u]=d;

parent[u]=pr;

int i,j=g[u].size(),t,q=0,r=1,k;

for(i=0;i<j;i++) if(g[u][i]!=pr) { //from all child, take the one with biggest subtree.

t=decomp(g[u][i],u,d+1);

r+=t;

if(t>q) {

q=t;

k=g[u][i];

}

}

if(r!=1 && (r>>1)<q) { //if subtree of k is bigger then the half of size of subtree at u, its heavy

skip[k]=u;

}

return r;

}

void init(int u,int r,int pr) {

bit[r].PB(0); //for this node insert a entry at bit

skip[u]=r; //r is the head of this chain

int i,j=g[u].size();

for(i=0;i<j;i++) if(g[u][i]!=pr) { //for each child, if that's skip is -1 then its a light edge and child's chain's parent is himself. Otherwist head is r

if(skip[g[u][i]]!=-1) init(g[u][i],r,u);

else {

bit[g[u][i]].clear();

bit[g[u][i]].PB(0); //for 0 first for 1 based bit

init(g[u][i],g[u][i],u);

}

}

}

int lca(int u,int v,int root) {

while(skip[u]!=skip[v]) { //while u & v are not in same chain

if(ds[skip[u]]<ds[skip[v]]) v=parent[skip[v]];

else u=parent[skip[u]];

}

if(ds[u]<ds[v]) return u;

else return v;

}

void update(int u,int v) {

int r=skip[u],w=v-wt[u]; //w is what have to add

wt[u]=v;

int x=ds[u]-ds[r]+1; //x is the position of u in it's bit chain

while(x<(int)bit[r].size()) {

bit[r][x]+=w;

x+=x&(-x);

}

}

int query(int u) {

int r=skip[u],x=ds[u]-ds[r]+1,q=0;

while(x) {

q+=bit[r][x];

x-=x&(-x);

}

return q;

}

int dist(int u,int root) {

int r=0;

while(u!=root) {

if(skip[u]==u) { //if light parent node, add weight and skip

r+=wt[u];

u=parent[u];

}

else { //heavy parent node.

r+=query(u);

if(skip[u]==root) return r; //if chain contains root, take it and leave

u=parent[skip[u]];

}

}

r+=wt[root]; //normally root not added.

return r;

}

int tm[mx];

int main() {

int I,T,n,i,j,k,l,q,u,v;

cin>>T;

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

cin>>n;

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

g[i].clear();

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

skip[i]=-1;

}

CLR(wt);

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

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

g[u].PB(v);

g[v].PB(u);

}

decomp(0,0,0);

bit[0].clear();

bit[0].PB(0);

init(0,0,0);

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

update(i,tm[i]);

}

cin>>q;

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

while(q--) {

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

if(i) {

update(u,v);

}

else {

k=lca(u,v,0);

i=dist(u,0);

j=dist(v,0);

l=dist(k,0);

printf("%d\n",i+j-(l<<1)+wt[k]);

}

}

}

return 0;

}

**Offline LCA algorithm**

/\*

Tarjan's offline LCA algorithm. For each pair of node's in P {u, v, qid},

it finds the LCA of the nodes in the rooted tree G (no edge to back to the parent.

The array ans holds the result for queries in orders defined by qid.

\*/

void lca(int u) {

int v, i, sz;

make\_set(u);

ancestor[find\_set(u)] = u;

sz = G[u].size();

for(i = 0; i < sz; i++) {

v = G[u][i];

lca(v);

union\_set(u, v);

ancestor[find\_set(u)] = u;

}

color[u] = 1;

sz = P[u].size();

for(i = 0; i < sz; i++) {

v = P[u][i].first;

if(color[v]) ans[P[u][i].second] = ancestor[find\_set(v)];

}

}

**Linear LCP Array Construction ( Kasai Algo )**

/\*

\* Kasai algo for computing LCP array in linear time.

\* LCP array is LCP of succeded pair of strings in suffix array

\* its based on a observation that for any suffix from ln[i] and its previous

\* suffix's lcp is h, then next suffix ln[i+1] has atleast h-1 lcp with thats precedor

\*/

int x,h=0;

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

x=dp[j][i];

if(!x) {

h=0;

lcp[0]=0;

continue;

}

a=i+h;

b=dt[x-1]+h;

while(ln[a++]==ln[b++]) h++;

lcp[x]=h;

if(h) h--;

}

r=0;

t=p-1;

i=0;if(min(l-dt[i],q)>max(t,p-1)) r+=min(l-dt[i],q)-max(t,p-1);

for(i=1;i<l;i++) {

t=lcp[i];

if(min(l-dt[i],q)>max(t,p-1)) r+=min(l-dt[i],q)-max(t,p-1);

}

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

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Three Dimensional convex hull**

/\*

\* Three Dimensional convex hull

\* \*/

struct Point {

double x, y, z;

Point() {}

Point(double xx, double yy, double zz) : x(xx), y(yy), z(zz) {}

//The difference between the two vectors

Point operator -(const Point p1) {

return Point(x - p1.x, y - p1.y, z - p1.z);

}

//The sum of two vectors

Point operator +(const Point p1) {

return Point(x + p1.x, y + p1.y, z + p1.z);

}

//take a fork

Point operator \*(const Point p) {

return Point(y \* p.z - z \* p.y, z \* p.x - x \* p.z, x \* p.y - y \* p.x);

}

Point operator \*(double d) {

return Point(x \* d, y \* d, z \* d);

}

Point operator /(double d) {

return Point(x / d, y / d, z / d);

}

//Dot

double operator ^(Point p) {

return (x \* p.x + y \* p.y + z \* p.z);

}

};

#define MAXN 400

struct CH3D {

struct face {

//Said convex surface of a three-point packet number

int a, b, c;

//Indicates that the plane belongs to the final surface on the convex hull

bool ok;

};

//The initial vertices

int n;

//Initial vertex

Point P[MAXN];

//The number of convex hull surface triangle

int num;

//Triangular convex hull surface

face F[8 \* MAXN];

//Triangular convex hull surface

int g[MAXN][MAXN];

//Vector length

double vlen(Point a) {

return sqrt(a.x \* a.x + a.y \* a.y + a.z \* a.z);

}

//Take a fork

Point cross(const Point &a, const Point &b, const Point &c) {

return Point((b.y - a.y) \* (c.z - a.z) - (b.z - a.z) \* (c.y - a.y),

(b.z - a.z) \* (c.x - a.x) - (b.x - a.x) \* (c.z - a.z),

(b.x - a.x) \* (c.y - a.y) - (b.y - a.y) \* (c.x - a.x));

}

//Triangle Area \* 2

double area(Point a, Point b, Point c) {

return vlen((b - a) \* (c - a));

}

//Directed tetrahedron volume \* 6

double volume(Point a, Point b, Point c, Point d) {

return (b - a) \* (c - a) ^ (d - a);

}

//Positive: point in the same direction as the surface

double dblcmp(Point &p, face &f) {

Point m = P[f.b] - P[f.a];

Point n = P[f.c] - P[f.a];

Point t = p - P[f.a];

return (m \* n) ^ t;

}

void deal(int p, int a, int b) {

int f = g[a][b]; //Another plane searching the adjacent side

face add;

if (F[f].ok) {

if (dblcmp(P[p], F[f]) > eps)

dfs(p, f);

else {

add.a = b;

add.b = a;

add.c = p; //Note here that the order to a right-handed

add.ok = true;

g[p][b] = g[a][p] = g[b][a] = num;

F[num++] = add;

}

}

}

void dfs(int p, int now) //Recursive search of all should be removed from

//the inner surface of the convex hull

{

F[now].ok = 0;

deal(p, F[now].b, F[now].a);

deal(p, F[now].c, F[now].b);

deal(p, F[now].a, F[now].c);

}

bool same(int s, int t) {

Point &a = P[F[s].a];

Point &b = P[F[s].b];

Point &c = P[F[s].c];

return fabs(volume(a, b, c, P[F[t].a])) < eps

&& fabs(volume(a, b, c, P[F[t].b])) < eps

&& fabs(volume(a, b, c, P[F[t].c])) < eps;

}

//Construction of three-dimensional convex hull

void create() {

int i, j, tmp;

face add;

num = 0;

if (n < 4)

return;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//This section is to ensure that the first four non-coplanar points

bool flag = true;

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

if (vlen(P[0] - P[i]) > eps) {

swap(P[1], P[i]);

flag = false;

break;

}

}

if (flag)

return;

flag = true;

//So that the first three points are not collinear

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

if (vlen((P[0] - P[1]) \* (P[1] - P[i])) > eps) {

swap(P[2], P[i]);

flag = false;

break;

}

}

if (flag)

return;

flag = true;

//Not four points of the front face

for (int i = 3; i < n; i++) {

if (fabs((P[0] - P[1]) \* (P[1] - P[2]) ^ (P[0] - P[i])) > eps) {

swap(P[3], P[i]);

flag = false;

break;

}

}

if (flag)

return;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

for (i = 0; i < 4; i++) {

add.a = (i + 1) % 4;

add.b = (i + 2) % 4;

add.c = (i + 3) % 4;

add.ok = true;

if (dblcmp(P[i], add) > 0)

swap(add.b, add.c);

g[add.a][add.b] = g[add.b][add.c] = g[add.c][add.a] = num;

F[num++] = add;

}

for (i = 4; i < n; i++) {

for (j = 0; j < num; j++) {

if (F[j].ok && dblcmp(P[i], F[j]) > eps) {

dfs(i, j);

break;

}

}

}

tmp = num;

for (i = num = 0; i < tmp; i++)

if (F[i].ok)

F[num++] = F[i];

}

//Surface

double area() {

double res = 0;

if (n == 3) {

Point p = cross(P[0], P[1], P[2]);

res = vlen(p) / 2.0;

return res;

}

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

res += area(P[F[i].a], P[F[i].b], P[F[i].c]);

return res / 2.0;

}

double volume() {

double res = 0;

Point tmp(0, 0, 0);

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

res += volume(tmp, P[F[i].a], P[F[i].b], P[F[i].c]);

return fabs(res / 6.0);

}

//The number of surface triangles

int triangle() {

return num;

}

//The number of polygons surface

int polygon() {

int i, j, res, flag;

for (i = res = 0; i < num; i++) {

flag = 1;

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

if (same(i, j)) {

flag = 0;

break;

}

res += flag;

}

return res;

}

//Three-dimensional convex hull focus

Point barycenter() {

Point ans(0, 0, 0), o(0, 0, 0);

double all = 0;

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

double vol = volume(o, P[F[i].a], P[F[i].b], P[F[i].c]);

ans = ans + (o + P[F[i].a] + P[F[i].b] + P[F[i].c]) / 4.0 \* vol;

all += vol;

}

ans = ans / all;

return ans;

}

//Point to the plane distance

double ptoface(Point p, int i) {

return fabs(

volume(P[F[i].a], P[F[i].b], P[F[i].c], p)

/ vlen(

(P[F[i].b] - P[F[i].a])

\* (P[F[i].c] - P[F[i].a])));

}

};

**CHINESE REMAINDER THEOREM**

/\*

\* Author : Bidhan Roy

\* Required Headers : <vector>

\* If you find any bug report at : mail2bidhan@gmail.com

\* Solves equations of the format x % mods[i] = r[i], ( 0<=i<n, where n is the number of equations )

\*/

long long CRT(const vector< long long > &r,const vector< long long > &mods){

long long M=1;

for(int i=0; i<int(mods.size()); i++) M\*=mods[i];

vector< long long > m, s;

for(int i=0; i<int(mods.size()); i++){

m.push\_back(M/mods[i]);

long long temp=m[i]%mods[i];

long long k=0;

/\* if there is a possibility of k being very big, then prime factorize m[i],

\* find modular inverse of 'temp' of each of the factors

\* 'k' equals to the multiplication ( modular mods[i] ) of modular inverses

\*/

while(true){

if((k\*temp)%mods[i]==1) break;

k++;

}

s.push\_back(k);

}

long long ret=0;

for(int i=0; i<int(s.size()); i++) {

ret+=( (m[i]\*s[i])%M \*r[i] )%M;

if(ret>=M) ret-=M;

}

return ret;

}

**ONLINE BRIDGE COUNT**

const int MAXN = 50010 ;

int n, bridges, par [ MAXN ] , bl [ MAXN ] , comp [ MAXN ] , size [ MAXN ] ;

void init ( ) {

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

bl [ i ] = comp [ i ] = i ;

size [ i ] = 1 ;

par [ i ] = - 1 ;

}

bridges = 0 ;

}

int get ( int v ) {

if ( v == - 1 ) return - 1 ;

return bl [ v ] == v ? v : bl [ v ] = get ( bl [ v ] ) ;

}

int get\_comp ( int v ) {

v = get ( v ) ;

return comp [ v ] == v ? v : comp [ v ] = get\_comp ( comp [ v ] ) ;

}

void make\_root ( int v ) {

v = get ( v ) ;

int root = v,

child = - 1 ;

while (

v

!= - 1 ) {

int p = get ( par [ v ] ) ;

par [ v ] = child ;

comp [ v ] = root ;

child = v ; v = p ;

}

size [ root ] = size [ child ] ;

}

int cu, u [ MAXN ] ;

void merge\_path ( int a, int b ) {

++ cu ;

vector < int > va, vb ;

int lca = - 1 ;

for ( ;; ) {

if ( a != - 1 ) {

a = get ( a ) ;

va. pb ( a ) ;

if ( u [ a ] == cu ) {

lca = a ;

break ;

}

u [ a ] = cu ;

a = par [ a ] ;

}

if ( b != - 1 ) {

b = get ( b ) ;

vb. pb ( b ) ;

if ( u [ b ] == cu ) {

lca = b ;

break ;

}

u [ b ] = cu ;

b = par [ b ] ;

}

}

for ( size\_t i = 0 ; i < va. size ( ) ; ++ i ) {

bl [ va [ i ] ] = lca ;

if ( va [ i ] == lca ) break ;

-- bridges ;

}

for ( size\_t i = 0 ; i < vb. size ( ) ; ++ i ) {

bl [ vb [ i ] ] = lca ;

if ( vb [ i ] == lca ) break ;

-- bridges ;

}

}

void add\_edge ( int a, int b ) {

a = get ( a ) ; b = get ( b ) ;

if ( a == b ) return ;

int ca = get\_comp ( a ) ,

cb = get\_comp ( b ) ;

if ( ca != cb ) {

++ bridges ;

if ( size [ ca ] > size [ cb ] ) {

swap ( a, b ) ;

swap ( ca, cb ) ;

}

make\_root ( a ) ;

par [ a ] = comp [ a ] = b ;

size [ cb ] += size [ a ] ;

}

else

merge\_path ( a, b ) ;

}

template<typename T> T SS(){

T res=0;

char c;

while(1) {

c=getchar\_unlocked();

if(c==EOF) return T(-1);

if(c==' '||c=='\n') continue;

else break;

}

res=c-'0';

while(1) {

c=getchar\_unlocked();

if(c>='0' && c<='9') res=10\*res+c-'0';

else break;

}

return res;

}

#define Q SS<int>()

#define QQ SS<i64>()

int main(){

int test=Q;

while( test-- ){

n=Q;

int m=Q;

init();

while(m--){

add\_edge(Q,Q);

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

}

}

return 0;

}

**REGULAR EXPRESSION EDIT DISTANCE**

#define id(ch) (ch=='a'?1:2)

#define mx 300

vi reg[mx][3];

int Next;

char str[60];

void addEdge(int s,int e,char trans){

int num=0;

if(trans=='a') num=1;

else if(trans=='b') num=2;

reg[s][num].pb(e);

}

void build(int &idx,int &s,int &t){

if( str[idx]!='(' ){

s=Next++, t=Next++;

addEdge(s,t,str[idx]);

idx++;

}

else{

idx++;

int s1,t1;

build(idx,s1,t1);

if(str[idx]=='\*'){

addEdge(t1,s1,' ');

s=t=s1;

idx++;

}

else if(str[idx]=='|'){

int s2,t2;

idx++;

s=Next++, t=Next++;

build(idx,s2,t2);

addEdge(s,s1,' ');

addEdge(s,s2,' ');

addEdge(t1,t,' ');

addEdge(t2,t,' ');

}

else{

int s2,t2;

build(idx,s2,t2);

addEdge(t1,s2,' ');

s=s1;

t=t2;

}

idx++;

}

}

#define d(x) dp[x.xx][x.yy]

int dp[mx][mx];

int dijkstra(pii s,pii e){

rep(i,Next) memset(dp[i],63,sizeof(int)\*Next);

d(s)=0;

priority\_queue< pair< int , pii > > pq;

pq.push( mp(d(s),s) );

while(!pq.empty()){

pii top=pq.top().yy;

if(top==e) return d(top);

pq.pop();

rep(t1,3) rep(t2,3) {

foreach(i,reg[top.xx][t1]) foreach(j,reg[top.yy][t2]) {

int add=(t1!=t2);

if(dp[\*i][\*j]>d(top)+add){

dp[\*i][\*j]=d(top)+add;

pq.push(mp(-dp[\*i][\*j],pii(\*i,\*j)));

}

}

}

rep(t,3){

foreach(i,reg[top.xx][t]){

int add=(t>0);

if(dp[\*i][top.yy]>d(top)+add){

dp[\*i][top.yy]=d(top)+add;

pq.push(mp(-dp[\*i][top.yy],pii(\*i,top.yy)));

}

}

foreach(i,reg[top.yy][t]){

int add=(t>0);

if(dp[top.xx][\*i]>d(top)+add){

dp[top.xx][\*i]=d(top)+add;

pq.push(mp(-dp[top.xx][\*i],pii(top.xx,\*i)));

}

}

}

}

if(d(e)>1000000) return -1;

return d(e);

}

int main(){

//read("rin");

int test;

scanf("%d",&test);

while(test--){

rep(i,Next) {

reg[i][0].clr;

reg[i][1].clr;

reg[i][2].clr;

}

Next=0;

scanf("%s",str);

int idx=0,s1,t1,s2,t2;

build(idx,s1,t1);

scanf("%s",str);

idx=0;

build(idx,s2,t2);

pii start,end;

start=pii(s1,s2);

end=pii(t1,t2);

printf("%d\n",dijkstra(start,end));

}

return 0;

}

**Minimum Subsequent Value O(N)**

struct MyQ{

deque<int> D,Min;

void push(int val){

D.push\_back(val);

while(!Min.empty() && Min.back()>val) Min.pop\_back();

Min.push\_back(val);

}

void pop(){

if(Min.front()==D.front() )

Min.pop\_front();

D.pop\_front();

}

int get(){

return Min.front();

}

};

**FAST FOURIER TRANSFORM ( FFT )**

struct pdd{

long double x,y;

pdd():x(0),y(0) {}

pdd(double \_x,double \_y):x(\_x),y(\_y) {}

inline pdd operator +(const pdd &b){

return pdd(x+b.x,y+b.y);

}

inline pdd operator -(const pdd &b){

return pdd(x-b.x,y-b.y);

}

inline pdd operator \*(const pdd &b){

return pdd(x\*b.x-y\*b.y,x\*b.y+y\*b.x);

}

inline pdd operator /(const double &b){

return pdd(x/b,y/b);

}

inline pdd conj(){

return pdd(x,-y);

}

};

const double PI=M\_PI;

const int maxn = 1<<20;

struct FFT {

int n,SN,rv[1<<20];

pdd w[1<<20];

void fft(pdd \*a, bool inv){

int cc=0;

rep(i,30) if (n&1<<i) cc=i;

if (cc!=SN){

SN=cc;

rv[0]=0; rv[1]=1;

for(int st = 1; st <= SN-1; st++){

int k=1<<st;

rep(i,k){

rv[i+(1<<st)]=rv[i]\*2+1;

rv[i]\*=2;

}

}

rep(i,1<<SN) w[i]=pdd(cos(2.0\*PI\*i/n),sin(2.0\*PI\*i/n));

}

rep(i,n) if (rv[i]<=i) swap(a[i],a[rv[i]]);

for (int st=2;st<=n;st\*=2){

int d=n/st,o=st/2;

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

for (int j=0;j<o;++j){

pdd u=a[i+j],v=a[i+j+o]\*(inv?w[j\*d].conj():w[j\*d]);

a[i+j]=u+v;

a[i+j+o]=u-v;

}

}

}

if (inv) rep(i,n) a[i]=a[i]/n;

}

void Multi(long long \*A, int NA, long long\* B, int NB, long long\* C, pdd\* tA, pdd\* tB) {

SN = -1;

n = 1;

while(n < NA + NB) n \*= 2;

n \*= 2;

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

tA[i] = (i < NA ? pdd(A[i], 0) : pdd(0, 0));

tB[i] = (i < NB ? pdd(B[i], 0) : pdd(0, 0));

}

fft(tA, 0);

fft(tB, 0);

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

tA[i] = tA[i] \* tB[i];

fft(tA, 1);

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

C[i] = (long long)(tA[i].x + 0.5);

}

};

FFT fft;

i64 A[maxn],B[maxn],C[maxn];

pdd tA[maxn],tB[maxn];

#define mx 20000

char l1[mx],l2[mx];

int len1,len2;

i64 mul[]={1,10,100,1000,10000,100000};

int main(){

int test;

scanf("%d",&test);

while(test--){

scanf("%s%s",l1,l2);

len1=strlen(l1), len2=strlen(l2);

int idx1=0, idx2=0;

A[0]=0, B[0]=0;

int k1=len1-1, k2=len2-1;

int aa=0, bb=0;

while(k1>=0 || k2>=0){

if(k1>=0){

A[idx1]=i64(l1[k1]-'0')\*mul[aa]+A[idx1];

aa++;

if(aa==5){

aa=0;

idx1++;

A[idx1]=0;

}

k1--;

}

if(k2>=0){

B[idx2]=i64(l2[k2]-'0')\*mul[bb]+B[idx2];

bb++;

if(bb==5){

bb=0;

idx2++;

B[idx2]=0;

}

k2--;

}

}

if(aa) idx1++;

if(bb) idx2++;

fft.Multi(A,idx1,B,idx2,C,tA,tB);

int idx=idx1+idx2-1;

i64 tmp=0;

rep(i,idx){

tmp+=C[i];

if(tmp>100000) C[i]=tmp%100000, tmp/=100000;

else C[i]=tmp, tmp=0;

}

while(tmp){

C[idx++]=tmp%100000, tmp/=100000;

}

for(int i=idx-1; i>=0; i--) if(i==idx-1) printf("%lld",C[i]);

else printf("%05lld",C[i]); puts("");

}

return 0;

}

**SIMPLEX**

// Simon Lo's

// Simplex algorithm on augmented matrix a of dimension (m+1)x(n+1)

// returns 1 if feasible, 0 if not feasible, -1 if unbounded

// returns solution in b[] in original var order, max(f) in ret

// form: maximize sum\_j(a\_mj\*x\_j)-a\_mn s.t. sum\_j(a\_ij\*x\_j)<=a\_in

// in standard form.

// To convert into standard form:

// 1. if exists equality constraint, then replace by both >= and <=

// 2. if variable x doesn't have nonnegativity constraint, then replace by

// difference of 2 variables like x1-x2, where x1>=0, x2>=0

// 3. for a>=b constraints, convert to -a<=-b

// note: watch out for -0.0 in the solution, algorithm may cycle

// eps = 1e-7 may give wrong answer, 1e-10 is better

#define maxm 30

#define maxn 30

double inf = 1e100;

double eps = 1e-10;

void pivot(int m, int n, double a[maxm][maxn], int B[maxm], int N[maxn], int r, int c) {

int i, j;

swap(N[c], B[r]);

a[r][c]=1/a[r][c];

for (j=0; j<=n; j++)if (j!=c) a[r][j]\*=a[r][c];

for (i=0; i<=m; i++)if (i!=r) {

for (j=0; j<=n; j++)if (j!=c)

a[i][j]-=a[i][c]\*a[r][j];

a[i][c] = -a[i][c]\*a[r][c];

}

}

int feasible(int m, int n, double a[maxm][maxn], int B[maxm], int N[maxn]) {

int r, c, i; double p, v;

while (1) {

for (p=inf, i=0; i<m; i++) if (a[i][n]<p) p=a[r=i][n];

if (p>-eps) return 1;

for (p=0, i=0; i<n; i++) if (a[r][i]<p) p=a[r][c=i];

if (p>-eps) return 0;

p = a[r][n]/a[r][c];

for (i=r+1; i<m; i++) if (a[i][c]>eps) {

v = a[i][n]/a[i][c];

if (v<p) r=i, p=v;

}

pivot(m, n, a, B, N, r, c);

}

}

int simplex(int m, int n, double a[maxm][maxn], double b[maxn], double& ret) {

int B[maxm], N[maxn], r, c, i; double p, v;

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

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

if (!feasible(m, n, a, B, N)) return 0;

while (1) {

for (p=0, i=0; i<n; i++) if (a[m][i]>p)

p=a[m][c=i];

if (p<eps) {

for (i=0; i<n; i++) if (N[i]<n)

b[N[i]]=0;

for (i=0; i<m; i++) if (B[i]<n)

b[B[i]]=a[i][n];

ret = -a[m][n];

return 1;

}

for (p=inf, i=0; i<m; i++) if (a[i][c]>eps) {

v = a[i][n]/a[i][c];

if (v<p) p=v, r=i;

}

if (p==inf) return -1;

pivot(m, n, a, B, N, r, c);

}

}

**CONVEX HULL**

/\* Author : Bidhan Roy

\* Complexity : O (Nlog(N))

\* Handles Collinear points and duplicate points

\* Report at `mail2bidhan@gmail.com` if you find any buf

\*/

point p0;

bool comp(point a,point b){

i64 d=area(p0,a,b);

if(d<0) return false;

if(d) return true;

return sqDist(p0,a)<sqDist(p0,b);

}

void convex(vector<point> &points,vector<point> &ans){

int pos=0;

rep(i,sz(points))

if(points[pos].y>points[i].y || (points[pos].y==points[i].y && points[pos].x>points[i].x)) pos=i;

p0=points[pos];

sort(all(points),comp);

int i=0;

while(p0==points[i]) {

i++;

if(i==sz(points)) return ;

}

int start=i;

ans.pb(points[i-1]);

while(!area(ans[0],points[start],points[i])) {

i++;

if(i==sz(points)) return ;

}

i--;

int sec=i;

ans.pb(points[i]); bool one=1;

while(!area(ans[0],ans[1],points[i])){

i++;

if(i==sz(points)) { if(one) return ; else break; }

one=0;

}

if(i-1>sec) i--;

if(i==sz(points)) return ;

ans.pb(points[i]);

i++;

for(; i<sz(points); i++){

while(area(ans[sz(ans)-2],ans[sz(ans)-1],points[i])<=0) ans.erase(ans.begin()+sz(ans)-1);

ans.pb(points[i]);

}

}

**SUFFIX ARRAY ( MANBER MAYER)**

namespace SA{

// Begins Suffix Arrays implementation

// O(n log n) - Manber and Myers algorithm

// Refer to "Suffix arrays: A new method for on-line string searches",

// by Udi Manber and Gene Myers

//Usage:

// Fill str with the characters of the string.

// Call SuffixSort(n), where n is the length of the string stored in str.

// That's it!

//Output:

// pos = The suffix array. Contains the n suffixes of str sorted in lexicographical order.

// Each suffix is represented as a single integer (the position of str where it starts).

// Rank = The inverse of the suffix array. Rank[i] = the index of the suffix str[i..n)

// in the pos array. (In other words, pos[i] = k <==> Rank[k] = i)

// With this array, you can compare two suffixes in O(1): Suffix str[i..n) is smaller

// than str[j..n) if and only if Rank[i] < Rank[j]

#define N 0

int str[N]; //input

int Rank[N], pos[N]; //output

int cnt[N], Next[N]; //internal

bool bh[N], b2h[N];

// Compares two suffixes according to their first characters

bool smaller\_first\_char(int a, int b){

return str[a] < str[b];

}

void suffixSort(int n){

//sort suffixes according to their first characters

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

pos[i] = i;

}

sort(pos, pos + n, smaller\_first\_char);

//{pos contains the list of suffixes sorted by their first character}

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

bh[i] = i == 0 || str[pos[i]] != str[pos[i-1]];

b2h[i] = false;

}

for (int h = 1; h < n; h <<= 1){

//{bh[i] == false if the first h characters of pos[i-1] == the first h characters of pos[i]}

int buckets = 0;

for (int i=0, j; i < n; i = j){

j = i + 1;

while (j < n && !bh[j]) j++;

Next[i] = j;

buckets++;

}

if (buckets == n) break; // We are done! Lucky bastards!

//{suffixes are separted in buckets containing strings starting with the same h characters}

for (int i = 0; i < n; i = Next[i]){

cnt[i] = 0;

for (int j = i; j < Next[i]; ++j){

Rank[pos[j]] = i;

}

}

cnt[Rank[n - h]]++;

b2h[Rank[n - h]] = true;

for (int i = 0; i < n; i = Next[i]){

for (int j = i; j < Next[i]; ++j){

int s = pos[j] - h;

if (s >= 0){

int head = Rank[s];

Rank[s] = head + cnt[head]++;

b2h[Rank[s]] = true;

}

}

for (int j = i; j < Next[i]; ++j){

int s = pos[j] - h;

if (s >= 0 && b2h[Rank[s]]){

for (int k = Rank[s]+1; !bh[k] && b2h[k]; k++) b2h[k] = false;

}

}

}

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

pos[Rank[i]] = i;

bh[i] |= b2h[i];

}

}

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

Rank[pos[i]] = i;

}

}

// End of suffix array algorithm

// Begin of the O(n) longest common prefix algorithm

// Refer to "Linear-Time Longest-Common-Prefix Computation in Suffix

// Arrays and Its Applications" by Toru Kasai, Gunho Lee, Hiroki

// Arimura, Setsuo Arikawa, and Kunsoo Park.

int LCP[N];

// LCP[i] = length of the longest common prefix of suffix pos[i] and suffix pos[i-1]

// LCP[0] = 0

void getHeight(int n){

rep(i,n) Rank[pos[i]]=i;

LCP[0]=0;

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

if(Rank[i]>0){

int j=pos[Rank[i]-1];

while(i+h<n && j+h<n && str[i+h]==str[j+h]) h++;

LCP[Rank[i]]=h;

if(h>0) h--;

}

}

}

// End of longest common prefixes algorithm

#undef N

}

**MATRIX EXPONENTIATION**

namespace matrix{

#define size 105 ///Maximum size of the matrix

#define wint int ///datatype to use (int,long long etc)

wint mat[size][size],tmp[size][size],res[size][size];

wint MOD;

int n;

void init(int \_n,wint \_MOD){ ///initialization, \_n=size of the square matrix , \_MOD=mod value

n=\_n;

MOD=\_MOD;

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

memset(tmp,0,sizeof (tmp));

memset(res,0,sizeof (res));

}

void mul(wint r[][size],wint a[][size],wint b[][size]) {

int i,j,t;

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

long long val=0;

for(t=0; t<n; t++) val+=1LL\*a[i][t]\*b[t][j];

tmp[i][j]=val%MOD;

}

memcpy(r,tmp,n\*size\*sizeof(tmp[0][0]));

}

void mPow(wint r[][size],wint a[][size],wint p) {

int i;

if (p<0) return;

memset(r,0,sizeof(r));

for(i=0; i<n; i++) r[i][i]=1;

while(p>0) {

if(p&1) mul(r,r,a);

mul(a,a,a); p>>=1;

}

}

void pow(wint p){

mPow(res,mat,p);

memcpy(mat,res,n\*size\*sizeof(tmp[0][0]));

}

void print(wint pp[][size]){

int i,j;

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

for(j=0; j<n; j++) printf(" %09d",pp[i][j]);

printf("\n");

}

}

}

**MANACHER ALGORITHM**

/\*

\* Author : Bidhan Roy

\* Required Headers : <vector>,<algorithm>;

\* Complexity : O (|N|)

\* If you find any bug report at : mail2bidhan@gmail.com

\*/

int call(char \*inp,char \*str,int \*F,vector< pair<int,int> > &vec){

//inp is the actual string

//str is the modified string with double size of inp

//F[i] contains the length of the palindrome centered at index i

//Every element of vec cointains starting and ending positions of palindromes

int len=0;

str[len++]='\*';

for(int i=0; inp[i]; i++){

str[len++]=inp[i];

str[len++]='\*';

}

str[len]='\0';

int c=0,r=0,ans=0;

for(int i=1; i&lt;len-1; i++){

int \_i=c-(i-c);

if(r&gt;i) F[i]=min(F[\_i],r-i);

else F[i]=0;

while(i-1-F[i]&gt;=0 &amp;&amp; str[i-1-F[i]]==str[i+1+F[i]]) {

F[i]++;

}

if(i+F[i]&gt;r) r=i+F[i],c=i;

ans=max(ans,F[i]);

vec.push\_back(make\_pair(i-F[i],i+F[i]));

}

return ans;

}