L24 : Advanced Graphs

1-Tut : **Permutation Swaps**

Send Feedback

Kevin has a permutation P of N integers 1, 2, ..., N, but he doesn't like it. Kevin wants to get a permutation Q.

He also believes that there are M good pairs of integers (ai, bi). Kevin can perform following operation with his permutation:

Swap Px and Py only if (x, y) is a good pair.
Help him and tell if Kevin can obtain permutation Q using such operations.

Input format:
The first line of input will contain an integer T, denoting the number of test cases.
Each test case starts with two space-separated integers N and M. The next line contains N space-separated integers Pi. The next line contains N space-separated integers Qi. Each of the next M lines contains two space-separated integers ai and bi.
Output format:
For every test case output "YES" (without quotes) if Kevin can obtain permutation Q and "NO" otherwise.
Constraints:
1 ≤ T ≤ 10
2 ≤ N ≤ 100000
1 ≤ M ≤ 100000
1 ≤ Pi, Qi ≤ N. Pi and Qi are all distinct.
1 ≤ ai < bi ≤ N
Time Limit: 1 second
Sample Input 1:

```
2
4 1
1 3 2 4
1 4 2 3
3 4
4 1
1 3 2 4
1 4 2 3
2 4
```

Sample Output 1:

```
NO

YES
```

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <iterator>
```

```cpp
using namespace std;
void dfs(vector<int> *edges, int start, bool *visited, unordered_set<int> *component)
{
    if (visited[start])
    {
        return;
    }
    visited[start] = true;
    component->insert(start);
    for (vector<int>::iterator it = edges[start].begin(); it != edges[start].end(); it++)
    {
        if (!visited[*it])
        {
            dfs(edges, *it, visited, component);
        }
    }
}
unordered_set<unordered_set<int> *> *getComponents(vector<int> *edges, int n)
{
    bool *visited = new bool[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
    }
    unordered_set<unordered_set<int> *> *output = new unordered_set<unordered_set<int> *>();
    for (int i = 0; i < n; i++)
    {
        if (!visited[i])
        {
            unordered_set<int> *component = new unordered_set<int>();
            dfs(edges, i, visited, component);
            output->insert(component);
        }
    }
    return output;
}
int main()
{
    int t; // test cases
    cin >> t;
    while (t--)
    {
        int n, m;
        cin >> n >> m;
        int *p = new int[n];
        int *q = new int[n];
        for (int i = 0; i < n; i++)
        {
            cin >> p[i];
        }
        for (int i = 0; i < n; i++)
        {
            cin >> q[i];
        }

        // graph initiated
        vector<int> *edges = new vector<int>[n];
        for (int i = 0; i < m; i++)
        {
            int a, b;
            cin >> a >> b;
            edges[a - 1].push_back(b - 1);
            edges[b - 1].push_back(a - 1);
        }
        // adjacency list completed.
        unordered_set<unordered_set<int> *> *components = getComponents(edges, n);

        unordered_set<unordered_set<int> *>::iterator it1 = components->begin();
        bool flag = true;
        while (it1 != components->end())
        {
            unordered_set<int> *component = *it1;
            unordered_set<int>::iterator it2 = component->begin();
            unordered_set<int> p_index_set;
            unordered_set<int> q_index_set;
            while (it2 != component->end())
            {
                p_index_set.insert(p[*it2]);
                q_index_set.insert(q[*it2]);
                it2++;
            }
            if (p_index_set != q_index_set)
            {
                flag = false;
            }
            it1++;
        }
```

```
        if (flag)
        {
            cout << "YES" << endl;
        }
        else
        {
            cout << "NO" << endl;
        }
    }
    return 0;
}
```

2-Tut : Connected Horses

Send Feedback

You all must be familiar with the chess-board having 8 x 8 squares of alternate black and white cells. Well, here we have for you a similar N x M size board with similar arrangement of black and white cells.

A few of these cells have Horses placed over them. Each horse is unique. Now these horses are not the usual horses which could jump to any of the 8 positions they usually jump in. They can move only if there is another horse on one of the 8-positions that it can go to usually and then both the horses will swap their positions. This swapping can happen infinitely times.

A photographer was assigned to take a picture of all the different ways that the horses occupy the board! Given the state of the board, calculate answer. Since this answer may be quite large, calculate in modulo 10^9+7.

Input Format:

First line contains T which is the number of test cases.

T test cases follow first line of each containing three integers N, M and Q where N,M is the size of the board and Q is the number of horses on it.

Q lines follow each containing the 2 integers, X and Y which are the coordinates of the Horses.

Output format:

For each test case, output the number of photographs taken by a photographer in new line.

Constraints:

1<=T<=10

1<=N,M<=1000

1<=Q<=N*M

Sample Input:

```
2
4 4 4
1 1
1 2
3 1
3 2
4 4 4
1 1
```

```
1 2
3 1
4 4
```

Sample Output:

```
4

2
```

```
#include <iostream>
#include <vector>

using namespace std;
#define pii pair<ll, ll>
#define MOD 1000000007
#define ll long long int

ll dfs(ll i, ll j, vector<pii> **graph, ll **visited)
{
    visited[i][j] = 1;
    ll answer = 1;

    for (ll k = 0; k < graph[i][j].size(); k++)
    {
        ll x = graph[i][j][k].first;
        ll y = graph[i][j][k].second;
        if (!visited[x][y])
        {
            answer = (answer + dfs(x, y, graph, visited)) % MOD;
        }
    }

    return answer;
}

void fill_vector(vector<pii> **graph, ll **board, ll m, ll n)
{
    for (ll i = 0; i < n; ++i)
    {
        for (ll j = 0; j < m; ++j)
        {
            if (board[i][j] == 1)
            {
                graph[i][j].push_back(make_pair(i, j));

                if (i + 2 < n && j + 1 < m && board[i + 2][j + 1] == 1)
                {
                    graph[i][j].push_back(make_pair(i + 2, j + 1));
                }

                if (i + 2 < n && j - 1 >= 0 && board[i + 2][j - 1] == 1)
                {
                    graph[i][j].push_back(make_pair(i + 2, j - 1));
                }

                if (i - 2 >= 0 && j + 1 < m && board[i - 2][j + 1] == 1)
                {
                    graph[i][j].push_back(make_pair(i - 2, j + 1));
                }

                if (i - 2 >= 0 && j - 1 >= 0 && board[i - 2][j - 1] == 1)
                {
                    graph[i][j].push_back(make_pair(i - 2, j - 1));
                }

                if (i + 1 < n && j + 2 < m && board[i + 1][j + 2] == 1)
                {
                    graph[i][j].push_back(make_pair(i + 1, j + 2));
                }

                if (i + 1 < n && j - 2 >= 0 && board[i + 1][j - 2] == 1)
                {
                    graph[i][j].push_back(make_pair(i + 1, j - 2));
                }

                if (i - 1 >= 0 && j + 2 < m && board[i - 1][j + 2] == 1)
                {
                    graph[i][j].push_back(make_pair(i - 1, j + 2));
                }

                if (i - 1 >= 0 && j - 2 >= 0 && board[i - 1][j - 2] == 1)
                {
                    graph[i][j].push_back(make_pair(i - 1, j - 2));
                }
            }
        }
    }

    return;
}
```

```cpp

int main()
{
    // To get the factorial
    ll *factorial = new ll[1000000];
    factorial[1] = 1;
    for (ll i = 2; i < 1000000; i++)
    {
        factorial[i] = (factorial[i - 1] * i) % MOD;
    }

    // Main part of the input
    ll t;
    cin >> t;
    while (t--)
    {
        ll n, m;
        ll q;
        cin >> n >> m >> q;

        // Create a 2d array for board
        ll **board = new ll *[n];
        for (ll i = 0; i < n; ++i)
        {
            board[i] = new ll[m];
            for (ll j = 0; j < m; ++j)
            {
                board[i][j] = 0;
            }
        }

        // Fill the board
        while (q--)
        {
            ll x, y;
            cin >> x >> y;
            board[x - 1][y - 1] = 1;
        }

        // Creating a graph 2d array
        vector<pii> **graph = new vector<pii> *[n];
        for (ll i = 0; i < n; ++i)
        {
            graph[i] = new vector<pii>[m];
        }

        // Fill the graph
        fill_vector(graph, board, m, n);

        ll **visited = new ll *[n];
        for (ll i = 0; i < n; ++i)
        {
            visited[i] = new ll[m];
            ;
            for (ll j = 0; j < m; ++j)
            {
                visited[i][j] = 0;
            }
        }

        ll ans = 1;
        for (ll i = 0; i < n; ++i)
        {
            for (ll j = 0; j < m; ++j)
            {
                if (visited[i][j] == 0 && board[i][j] == 1)
                {
                    ans = (ans * factorial[dfs(i, j, graph, visited)]) % MOD;
                }
            }
        }

        cout << ans << endl;
    }

    return 0;
}
```

3-Tut : Dominos

Send Feedback

Dominos are lots of fun. Children like to stand the tiles on their side in long lines. When one domino falls, it knocks down the next one, which knocks down the one after that, all the way down the line.

However, sometimes a domino fails to knock the next one down. In that case, we have to knock it down by hand to get the dominos falling again.

Your task is to determine, given the layout of some domino tiles, the minimum number of dominos that must be knocked down by hand in order for all of the dominos to fall.

Input Format:
The first line of input contains one integer T, specifying the number of test cases to follow.
Each test case begins with a line containing two integers
The first integer n is the number of domino tiles and the second integer m is the number of lines to follow in the test case.
The domino tiles are numbered from 1 to n.
Each of the following lines contains two integers x and y indicating that if domino number x falls, it will cause domino number y to fall as well.

Constraints:
1 <= T <= 50
1 <= N, M <= 10^5

Output Format:
For each test case, output a line containing one integer, the minimum number of dominos that must be knocked over by hand in order for all the dominos to fall.

Sample Input 1:
1
3 2
1 2
2 3

Sample Output 2:

1

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

stack<int> st;
vector<int> adjList[100010];
bool visited[100010];

void dfs2(int index)
{
    visited[index] = true;
    for (unsigned int j = 0; j < adjList[index].size(); j++)
    {
        if (!visited[adjList[index][j]])
        {
            dfs2(adjList[index][j]);
        }
    }
}

void dfs(int index)
{
    visited[index] = true;
    for (unsigned int j = 0; j < adjList[index].size(); j++)
    {
        if (!visited[adjList[index][j]])
        {
            dfs(adjList[index][j]);
        }
    }
    st.push(index);
}

int main()
{
    int tc;
    scanf("%d", &tc);
    while (tc--)
    {
        memset(visited, false, sizeof(visited));

        int n, m;
        scanf("%d %d", &n, &m);
        for (int i = 0; i < m; i++)
        {
            int a, b;
            scanf("%d %d", &a, &b);
            adjList[a].push_back(b);
        }

        for (int i = 1; i <= n; i++)
        {
            if (!visited[i])
            {
                dfs(i);
            }
        }
        memset(visited, false, sizeof(visited));
        int count = 0;
        while (!st.empty())
        {
            int index = st.top();
            st.pop();
            if (!visited[index])
            {
                count++;
                dfs2(index);
            }
        }
        printf("%d\n", count);
        for (int i = 1; i <= n; i++)
        {
            adjList[i].clear();
        }
    }
    return 0;
}
```

4-Tut : BOTTOM

Send Feedback

We will use the following (standard) definitions from graph theory. Let V be a non-empty and finite set, its elements being called vertices (or nodes). Let E be a subset of the Cartesian product V×V, its elements being called edges. Then G=(V,E) is called a directed graph.

Let n be a positive integer, and let p=(e1,…,en) be a sequence of length n of edges ei∈E such that ei=(vi,vi+1)for a sequence of vertices (v1,…,vn+1). Then p is called a path from vertex v1 to vertex vn+1 in G and we say that vn+1 is reachable from v1, writing (v1→vn+1).

Here are some new definitions. A node v in a graph G=(V,E) is called a sink, if for every node w in G that is reachable from v, v is also reachable from w. The bottom of a graph is the subset of all nodes that are sinks, i.e., bottom(G)={v∈V∣∀w∈V:(v→w)⇒(w→v)}. You have to calculate the bottom of certain graphs.

Input Format:

First line of input will contain T(number of test case), each test case follows as.
First line will contain two space-separated integers N and M denoting the number of vertex and edges respectively.
Next M line will contain two space separated integers a, b denoting an edge from a to b.

Output Format:

For each test case output the bottom of the specified graph on a single line.

Constraints:

1 <= T <= 50
1 <= N, M <= 10^5

Sample Input:

```
1
3 6
3 1
2 3
3 2
1 2
1 3
2 1
```

Sample Output:

```
1 2 3
```

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

void dfs(vector<int> *edges, int start, unordered_set<int> &visited, stack<int> &finished_vertices_stack)
{
    visited.insert(start);
    for (int i = 0; i < edges[start].size(); i++)
    {
        int adjacent = edges[start][i];
        if (visited.count(adjacent) == 0)
        {
            dfs(edges, adjacent, visited, finished_vertices_stack);
        }
    }
    finished_vertices_stack.push(start);
}
void dfs2(vector<int> *edgesT, int start, unordered_set<int> &visited, vector<int> &component)
{
    visited.insert(start);
    component.push_back(start);
    for (int i = 0; i < edgesT[start].size(); i++)
    {
        int adjacent = edgesT[start][i];
        if (visited.count(adjacent) == 0)
        {
            dfs2(edgesT, adjacent, visited, component);
        }
    }
}
vector<vector<int>> getSCC(vector<int> *edges, vector<int> *edgesT, int n)
{
    unordered_set<int> visited;
    stack<int> finished_vertices_stack;
    for (int i = 0; i < n; i++)
    {
        if (visited.count(i) == 0)
        {
            dfs(edges, i, visited, finished_vertices_stack);
        }
    }
    visited.clear();
    vector<vector<int>> output;
    while (!finished_vertices_stack.empty())
    {
        int element = finished_vertices_stack.top();
        finished_vertices_stack.pop();
        if (visited.count(element) == 0)
        {
            vector<int> component;
            dfs2(edgesT, element, visited, component);
            output.push_back(component);
        }
    }
    return output;
}
int main()
{
    int t;
    cin >> t;
    while (t--)
    {
        int v, e;
        cin >> v >> e;

        vector<int> *edges = new vector<int>[v];
        vector<int> *edgesT = new vector<int>[v];
        for (int i = 0; i < e; i++)
        {
            int j, k;
            cin >> j >> k;
            edges[j - 1].push_back(k - 1);
            edgesT[k - 1].push_back(j - 1);
        }
        vector<vector<int>> components = getSCC(edges, edgesT, v);

        auto it = components.begin();
        vector<int> ans;
        while (it != components.end())
        {
            int flag = 0;
            auto it2 = (*it).begin();
            while (it2 != (*it).end())
            {
                for (int i = 0; i < edges[*it2].size(); ++i)
                {

```

```
                int cc = count((*it).begin(), (*it).end(), edges[*it2][i]);
                if (cc == 0)
                {
                    flag = 1;
                    break;
                }
            }
            if (flag == 1)
            {
                break;
            }
            it2++;
        }
            if (flag == 0)
            {
                it2 = (*it).begin();
                while (it2 != (*it).end())
                {
                    ans.push_back(*it2 + 1);
                    it2++;
                }
            }

            it++;
        }

        sort(ans.begin(), ans.end());
        for (int i = 0; i < ans.size(); ++i)
        {
            cout << ans[i] << " ";
        }
        cout << endl;
    }
}

```

5-Tut : Fill the Matrix

Send Feedback

A matrix B (consisting of integers) of dimension N × N is said to be good if there exists an array A (consisting of integers) such that B[i][j] = |A[i] - A[j]|, where |x| denotes absolute value of integer x.

You are given a partially filled matrix B of dimension N × N. Q of the entries of this matrix are filled by either 0 or 1. You have to identify whether it is possible to fill the remaining entries of matrix B (the entries can be filled by any integer, not necessarily by 0 or 1) such that the resulting fully filled matrix B is good.

Input Format:

The first line of the input contains an integer T denoting the number of test cases.
The first line of each test case contains two space separated integers N, Q.
Each of the next Q lines contain three space separated integers i, j, val, which means that B[i][j] is filled with value val.

Constraints:

1 ≤ T ≤ 20
2 ≤ N ≤ 10^5
1 ≤ Q ≤ 10^5
1 ≤ i, j ≤ N
0 ≤ val ≤ 1

Output Format:

For each test case, output "yes" or "no" (without quotes) in a single line corresponding to the answer of the problem.

Sample Input 1:

```
1
5 4
1 2 0
2 2 0
5 2 1
2 1 1
```

Sample Output 1:

```
no
```

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+5;
bool error; int col[N];
vector<pair<int,int> > adj[N];
void dfs(int start){
        for(auto edge: adj[start]){
                int next = edge.first;
                if(col[next] == -1){
                        col[next] = col[start]^edge.second;
                        dfs(next);
                }
                else if(col[next] != col[start]^edge.second){
                        error = true;
                }
        }
}
int main(){
        ios_base::sync_with_stdio(false);
        cin.tie(0); cout.tie(0);
        int t; cin>>t;
```

```
            while(t--){
                        int n,m;
                        cin>>n>>m;
                        error = false;
                        for(int i=1;i<=n;i++){
                                    adj[i].clear();
                                    col[i] = -1;
                        }
                        while(m--){
                                    int a,b,c;
                                    cin>>a>>b>>c;
                                    adj[a].push_back({b,c});
                                    adj[b].push_back({a,c});
                        }
                        for(int i=1;i<=n;i++){
                                    if(col[i] == -1){
                                                col[i] = 0;
                                                dfs(i);
                                    }
         }
                        if(error) cout<<"no"<<endl;
                        else cout<<"yes"<<endl;
            }
}
```

6-Ass : Monk and the Islands

Send Feedback

Monk visits the land of Islands. There are a total of N islands numbered from 1 to N. Some pairs of islands are connected to each other by Bidirectional bridges running over water.

Monk hates to cross these bridges as they require a lot of efforts. He is standing at Island #1 and wants to reach the Island #N. Find the minimum the number of bridges that he shall have to cross, if he takes the optimal route.

Input format:

First line contains T. T testcases follow.
First line of each test case contains two space-separated integers N, M.
Each of the next M lines contains two space-separated integers X and Y , denoting that there is a bridge between Island X and Island Y.

Output format:

Print the answer to each test case in a new line.

Constraints:

1 ≤ T ≤ 10
1 ≤ N ≤ 10000
1 ≤ M ≤ 100000

1 ≤ X, Y ≤ N

Sample Input

```
2
3 2
1 2
2 3
4 4
1 2
2 3
3 4
4 2
```

Sample Output

```
2

2
```

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

inline int bfs(vector<vector<int>> graph, int n)
{
    bool *visited = new bool[n]; // considering the time complexity requirement of this question, an unordered set cant be used
    // for storing visited vertices. because visited.count() would cross the time limit

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

    int *level = new int[n + 1];
    for (int i = 0; i <= n; i++)
        level[i] = 0;

    queue<int> q;
    q.push(0);

    visited[0] = true;

    while (!q.empty())
    {
        int v = q.front();
        q.pop();
        for (int i = 0; i < graph[v].size(); i++)
        {
            int next = graph[v][i];
            if (!visited[next])
            {
                q.push(next);
                visited[next] = true;
                level[next] = level[v] + 1;
            }
        }
    }
    return level[n - 1];
}

int main()
{

    int t;
    cin >> t;
    while (t--)
    {
        int n, m;
        cin >> n >> m;
        vector<vector<int>> graph(n);

        for (int i = 0; i < m; i++)
        {
            int x, y;
            cin >> x >> y;
            graph[x - 1].push_back(y - 1);
            graph[y - 1].push_back(x - 1);
        }
        cout << bfs(graph, n) << endl;
    }
    return 0;
}
```

7-Ass : Kingdom Of Monkeys

Send Feedback

This is the story in Zimbo, the kingdom officially made for monkeys. Our Code Monk visited Zimbo and declared open a challenge in the kingdom, thus spoke to all the monkeys :

You all have to make teams and go on a hunt for Bananas. The team that returns with the highest number of Bananas will be rewarded with as many gold coins as the number of Bananas with them. May the force be with you!

Given there are N monkeys in the kingdom. Each monkey who wants to team up with another monkey has to perform a ritual. Given total M rituals are performed. Each ritual teams up two monkeys. If Monkeys A and B teamed up and Monkeys B and C teamed up, then Monkeys A and C are also in the same team.

You are given an array A where Ai is the number of bananas i'th monkey gathers.

Find out the number of gold coins that our Monk should set aside for the prize.

Input Format:
First line contains an integer T. T test cases follow.
First line of each test case contains two space-separated N and M. M lines follow.
Each of the M lines contains two integers Xi and Yi, the indexes of monkeys that perform the i'th ritual.
Last line of the testcase contains N space-separated integer constituting the array A.

Output Format:
Print the answer to each test case in a new line.

Constraints:
1 ≤ T ≤ 10
1 ≤ N ≤ 10^5
0 ≤ M ≤ 10^5
0 ≤ Ai ≤ 10^9

Sample Input:
```
1
4 3
1 2
2 3
3 1
1 2 3 5
```

Sample Output:
```
6
```

Explanation:
Monkeys 1,2 ,3 are in the same team. They gather 1+2+3=6 bananas.
Monkey 4 is a team. It gathers 5 bananas.

Therefore, number of gold coins (highest number of bananas by a team) = 6.

```
#include <bits/stdc++.h>
using namespace std;
typedef long long int ll;
ll dfs(vector<ll> *edges, ll *arr, ll n, bool *visited, ll start)
{
    ll sum = arr[start];
    visited[start] = true;
    for (ll i = 0; i < edges[start].size(); i++)
    {
        ll adjacent = edges[start][i];
        if (!visited[adjacent])
        {
            sum += dfs(edges, arr, n, visited, adjacent);
        }
    }
    return sum;
}
int main()
{
    ll t;
    cin >> t;
    while (t--)
    {
        ll n, m;
        cin >> n >> m;
        vector<ll> *edges = new vector<ll>[n];
        for (ll i = 0; i < m; i++)
        {
            ll x, y;
            cin >> x >> y;
            edges[x - 1].push_back(y - 1);
            edges[y - 1].push_back(x - 1);
        }
        ll *arr = new ll[n];
        for (ll i = 0; i < n; i++)
        {
            cin >> arr[i];
        }
        bool *visited = new bool[n];
        for (ll i = 0; i < n; i++)
        {
            visited[i] = false;
        }
        ll maximum = INT_MIN;
        for (ll i = 0; i < n; i++)
        {
            if (!visited[i])
            {
                ll current_component_total_bananas = dfs(edges, arr, n, visited, i);
                if (current_component_total_bananas > maximum)
                {
                    maximum = current_component_total_bananas;
                }
            }
        }
        cout << maximum << endl;
    }
}
```

8-Ass : New Year Transportation

Send Feedback

New Year Transportation

New Year is coming in Line World! In this world, there are n cells numbered by integers from 1 to n, as a 1 × n board. People live in cells. However, it was hard to move between distinct cells, because of the difficulty of escaping the cell. People wanted to meet people who live in other cells.

So, user tncks0121 has made a transportation system to move between these cells, to celebrate the New Year. First, he thought of n - 1 positive integers a1, a2, ..., an - 1. For every integer i where 1 ≤ i ≤ n - 1 the condition 1 ≤ ai ≤ n - i holds. Next, he made n - 1 portals, numbered by integers from 1 to n - 1. The i-th (1 ≤ i ≤ n - 1) portal connects cell i and cell (i + ai), and one can travel from cell i to cell (i + ai) using the i-th portal. Unfortunately, one cannot use the portal backwards, which means one cannot move from cell (i + ai) to cell i using the i-th portal. It is easy to see that because of condition 1 ≤ ai ≤ n - i one can't leave the Line World using portals.

Currently, I am standing at cell 1, and I want to go to cell d. However, I don't know whether it is possible to go there. Please determine whether I can go to cell d by only using the construted transportation system.

Input Format:

First line will contain T(number of test case), each test case follows as.
Line1: will contain N (number of cells)
Line2: contains n - 1 space-separated integers a1, a2, ..., an - 1 (1 ≤ ai ≤ n - i). It is guaranteed, that using the given transportation system, one cannot leave the Line World.
Line3: contain an integer Q (number of queries)
Next Q line will contain an integer d the cell where i want to go for that query

Output Format:

If I can go to cell d using the transportation system, print "YES". Otherwise, print "NO" for each test case and query in newline.

Sample Input 1:

```
1
8
1 2 1 2 1 2 1
1
4
```

Sample Output 1:

```
YES
```

Sample Input 2:

```
1
8
1 2 1 2 1 1 1
1
5
```

Sample Output 2:

```
NO
```

Note:

In the first sample, the visited cells are: 1, 2, 4; so we can successfully visit the cell 4.

In the second sample, the possible cells to visit are: 1, 2, 4, 6, 7, 8; so we can't visit the cell 5, which we want to visit.

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

void dfs(vector<int> *edges, int n, int start, bool *visited)
{
    visited[start] = true;
    for (int i = 0; i < edges[start].size(); i++)
    {
        int adjacent = edges[start][i];
        if (!visited[adjacent])
        {
            dfs(edges, n, adjacent, visited);
        }
    }
}
int main()
{
    int t;
    cin >> t;
    while (t--)
    {
        int n;
        cin >> n;
        vector<int> *edges = new vector<int>[n];
        for (int i = 1; i <= n - 1; i++)
        {
            int a;
            cin >> a;
            edges[i - 1].push_back(i + a - 1); // i-th (1 ≤ i ≤ n - 1) portal connects cell i and cell (i + ai)
        }
        bool *visited = new bool[n];
        for (int i = 0; i < n; i++)
        {
            visited[i] = false;
        }
        dfs(edges, n, 0, visited);
        int q;
        cin >> q;
        while (q--)
        {
            int d;
            cin >> d;
            if (visited[d - 1])
            {
                cout << "YES" << endl;
            }
            else
            {
                cout << "NO" << endl;
            }
        }
    }
    return 0;
}
```

9-Ass : AIRPORTS

Send Feedback

The government of a certain developing nation wants to improve transportation in one of its most inaccessible areas, in an attempt to attract investment. The region consists of several important locations that must have access to an airport.

Of course, one option is to build an airport in each of these places, but it may turn out to be cheaper to build fewer airports and have roads link them to all of the other locations. Since these are long distance roads connecting major locations in the country (e.g. cities, large villages, industrial areas), all roads are two-way. Also, there may be more than one direct road possible between two areas. This is because there may be several ways to link two areas (e.g. one road tunnels through a mountain while the other goes around it etc.) with possibly differing costs.

A location is considered to have access to an airport either if it contains an airport or if it is possible to travel by road to another location from there that has an airport. You are given the cost of building an airport and a list of possible roads between pairs of locations and their corresponding costs. The government now needs your help to decide on the cheapest way of ensuring that every location has access to an airport. The aim is to make airport access as easy as possible, and with minimum cost.

Input Format:

The first line of input contains the integer T (the number of test cases), each test case follow as.

Line1: Three space-separated integers N, M and cost number of locations, number of possible roads and cost of airport respectively

The following M lines each contain three integers X, Y and C, separated by white space. X and Y are two locations, and C is the cost of building a road between X and Y .

Output Format:

For each test case print the cost in a newline.

Constraints:

1 <= T <= 20

```
1 <= N, M <= 10^5
1 <= cost <= 10^9
1 <= weight(of each road) <= 10^9
```

Sample Input

```
2
4 4 100
1 2 10
4 3 12
4 1 41
2 3 23
5 3 1000
1 2 20
4 5 40
3 2 30
```

Sample Output

```
145

2090
```

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

#define ll long long int

class edge
{
public:
    ll ei, ej, cost;
};

ll find_parent(ll num, ll *parents, ll *rank)
{
    if (num == parents[num])
    {
        return num;
    }
    ll p = find_parent(parents[num], parents, rank);
    if (rank[p] <= rank[num])
    {
        parents[p] = num;
        rank[num]++;
    }
    else
    {
        parents[num] = p;
        rank[p]++;
    }
    return p;
}

bool compare(edge e1, edge e2)
{
    return e1.cost < e2.cost;
}

int main()
{
    ll t;
    cin >> t;
    for (ll iter = 1; iter <= t; iter++)
    {
        ll n, m, a;
        cin >> n >> m >> a;
        edge *edges = new edge[m + 1];
        for (ll i = 0; i < m; i++)
        {
            cin >> edges[i].ei >> edges[i].ej >> edges[i].cost;
        }
        sort(edges, edges + m, compare);
        // initially all will be parents of itself
        ll *parents = new ll[n + 1];
        for (ll i = 0; i <= n; i++)
        {
            parents[i] = i;
        }
        // set rank for each vertex
        ll *rank = new ll[n + 1]{};
        // connect those vertices with roads which have less cost
        // than building an airport
        ll cost = 0;
        ll j = 0;
        for (ll i = 0; i < m && j < n - 1; i++)
        {
            edge e = edges[i];
            // build a road if cost of building a road is less than airport
            // as we have already sorted we dont need to do this
            if (e.cost >= a)
            {
                break;
            }
            // find parents of both vertices
            ll p1 = find_parent(e.ei, parents, rank);
            ll p2 = find_parent(e.ej, parents, rank);
            if (p1 != p2)
            {
                cost += e.cost;
                if (rank[p1] <= rank[p2])
                {
                    parents[p1] = p2;
                    rank[p2]++;
                }
                else
                {
                    parents[p2] = p1;
                    rank[p1]++;
                }
                j++;
            }
        }
        ll numair = 0;
        for (ll i = 1; i <= n; i++)
        {
            if (i == parents[i])
            {
                cost += a;
                numair += 1;
            }
        }
         cout << cost << "\n";
     }
     return 0;
 }
```

10-Ass : Edges in MST

Send Feedback

You are given a connected weighted undirected graph without any loops and multiple edges.

Let us remind you that a graph's spanning tree is defined as an acyclic connected subgraph of the given graph that includes all of the graph's vertexes. The weight of a tree is defined as the sum of weights of the edges that the given tree contains. The minimum spanning tree (MST) of a graph is defined as the graph's spanning tree having the minimum possible weight. For any connected graph obviously exists the minimum spanning tree, but in the general case, a graph's minimum spanning tree is not unique.

Your task is to determine the following for each edge of the given graph: whether it is either included in any MST, or included at least in one MST, or not included in any MST.

Input Format:
The first line contains two space-separated integers n and m — the number of the graph's vertexes and edges, correspondingly.
Then follow m lines, each of them contains three integers — the description of the graph's edges as "ai bi wi", where ai and bi are the numbers of vertexes connected by the i-th edge, wi is the edge's weight.
Output Format:
Print m lines — the answers for all edges. If the i-th edge is included in any MST, print "any"; if the i-th edge is included at least in one MST, print "at least one"; if the i-th edge isn't included in any MST, print "none". Print the answers for the edges in the order in which the edges are specified in the input.
Constraints:
1 <= N, M <= 10^5
1 <= a, b <= N
1 <= w[i] <= 10^6
Graph is connected and does not contain self loops and multiple edges.
Sample Input:
```
4 5
1 2 101
1 3 100
2 3 2
2 4 2
3 4 1
```
Sample Output:
```
none
any
at least one
at least one

any
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#define N 100001
using namespace std;
int n, m, x[N], y[N], z[N], p[N];
int ans[N], f[N], h[N], pe[N], d[N];
vector<pair<int, int>> v[N];
bool cmp(const int &x, const int &y)
{
    return z[x] < z[y];
}
int par(int x)
{
    while (pe[x])
        x = pe[x];
    return x;
}
void uni(int x, int y)
{
    x = par(x);
    y = par(y);
    v[x].clear();
    v[y].clear();
    f[x] = 0;
    f[y] = 0;
    if (x == y)
        return;
    if (h[x] > h[y])
        pe[y] = x;
    else
    {
        pe[x] = y;
        if (h[x] == h[y])
            h[y]++;
    }
}
void add_edge(int x, int y, int i)
{
    if (x == y)
        return;
    ans[i] = 1;
    v[x].push_back({y, i});
    v[y].push_back({x, i});
}
void kruskal(int c, int g, int h)
{
    f[c] = true;
    d[c] = h;
    for (pair<int, int> i : v[c])
        if (!f[i.first])
        {
            kruskal(i.first, i.second, h + 1);
            d[c] = min(d[c], d[i.first]);
        }
        else if (i.second != g)
            d[c] = min(d[c], d[i.first]);
    if (d[c] == h)
```

```
        ans[g] = 2;
}
int main()
{
    cin >> n >> m;
    for (int i = 1; i <= m; i++)
    {
        cin >> x[i] >> y[i] >> z[i];
        p[i] = i;
    }
    sort(p + 1, p + m + 1, cmp);
    for (int i = 1; i <= m;)
    {
        int j;
        for (j = i; z[p[j]] == z[p[i]]; j++)
            add_edge(par(x[p[j]]), par(y[p[j]]), p[j]);
        for (j = i; z[p[j]] == z[p[i]]; j++)
        {
            int k = par(x[p[j]]);
            if (!f[k])
                kruskal(k, 0, 0);
        }
        for (j = i; z[p[j]] == z[p[i]]; j++)
            uni(x[p[j]], y[p[j]]);
        i = j;
    }
    for (int i = 1; i <= m; i++)
        if (!ans[i])
            cout << "none" << endl;
        else if (ans[i] == 1)
            cout << "at least one" << endl;
        else
            cout << "any" << endl;
    return 0;
}
```

11-Bonus : Strongly Connected Components

```
#include <iostream>
#include <vector>
#include <stack>
#include <iterator>
#include <unordered_set>
using namespace std;
```

```
void dfs(vector<int> *edges, unordered_set<int> &visited, stack<int> &finished_vertices, int start)
{
    visited.insert(start);
    for (int i = 0; i < edges[start].size(); i++)
    {
        int adjacent_element = edges[start][i];
        if (visited.count(adjacent_element) == 0)
        {
            dfs(edges, visited, finished_vertices, adjacent_element);
        }
    }
    finished_vertices.push(start);
}
void dfs2(vector<int> *edgesT, int start, unordered_set<int> &visited, unordered_set<int> *component)
{
    visited.insert(start);
    component->insert(start);
    for (int i = 0; i < edgesT[start].size(); i++)
    {
        int adjacent = edgesT[start][i];
        if (visited.count(adjacent) == 0)
        {
            dfs2(edgesT, adjacent, visited, component);
        }
    }
}
unordered_set<unordered_set<int> *> *getSCC(vector<int> *edges, vector<int> *edgesT, int n)
{
    unordered_set<int> visited;
    stack<int> finished_vertices;
    for (int i = 0; i < n; i++)
    {
        if (visited.count(i) == 0)
        {
            dfs(edges, visited, finished_vertices, i);
        }
    }
    unordered_set<unordered_set<int> *> *output = new unordered_set<unordered_set<int> *>();
    visited.clear();
    while (!finished_vertices.empty())
    {
        int element = finished_vertices.top();
        finished_vertices.pop();
        if (visited.count(element) != 0)
        {
            continue;
        }
        unordered_set<int> *component = new unordered_set<int>();
        dfs2(edgesT, element, visited, component);
        output->insert(component);
    }
    return output;
}
int main()
{
    int n; // number of vertices
    cin >> n;
    vector<int> *edges = new vector<int>[n];
    vector<int> *edgesT = new vector<int>[n];
    int m; // number of eges
    cin >> m;
    for (int i = 0; i < m; i++)
    {
        int j, k;
        cin >> j >> k;
        edges[j - 1].push_back(k - 1);
        edgesT[k - 1].push_back(j - 1);
    }
    unordered_set<unordered_set<int> *> *components = getSCC(edges, edgesT, n);

    unordered_set<unordered_set<int> *>::iterator it1 = components->begin();
    while (it1 != components->end())
    {
        unordered_set<int> *component = *it1;
        unordered_set<int>::iterator it2 = component->begin();
        while (it2 != component->end())
        {
            cout << *it2 + 1 << " ";
            it2++;
        }
        cout << endl;
        delete component;
        it1++;
    }

    delete components;
    delete[] edges;
    delete[] edgesT;
}
```

12-Bonus : Bipartite

```
#include <iostream>
#include <vector>
#include <unordered_set>
using namespace std;
bool bipartite(vector<int> *edges, int n)
{
    if (n == 0)
    {
        return true;
    }
    unordered_set<int> sets[2];
    vector<int> pending;
    sets[0].insert(0);
    pending.push_back(0);
    while (pending.size() > 0)
    {
        int current = pending.back();
        pending.pop_back(); // here pending vector is being used as a stack
        int set_of_current_element = sets[0].count(current) > 0 ? 0 : 1;
        // is the current element is in pending then it has to be in one of the sets, either 0 or
1.
        // now iterating over all of its neighbours
        for (int i = 0; i < edges[current].size(); i++)
        {
            int adjacent = edges[current][i];
            if (sets[0].count(adjacent) == 0 && sets[1].count(adjacent) == 0)
            {
                sets[1 - set_of_current_element].insert(adjacent);
                pending.push_back(adjacent);
            }
            else // in this case the element is in one of the sets
            {
                if (sets[set_of_current_element].count(adjacent) > 0)
                {
                    return false;
                }
            }
        }
    }
    return true;
    // in this code we have considered that or graph is fully connected. there are not disconnected components
}
int main()
{
    while (true)
    {
        int n;
        cin >> n;
        if (n == 0)
        {
            return 0;
        }
        vector<int> *edges = new vector<int>[n];
        int m;
        cin >> m;
        for (int i = 0; i < m; i++)
        {
            int j, k;
            cin >> j >> k;
            edges[j - 1].push_back(k - 1); // considering that the vertices are numbered from 1 to n.
            edges[k - 1].push_back(j - 1); // for internal calculation we will consider numbering from 0 to n-1.
        }
        bool ans = bipartite(edges, n);
        if (ans)
        {
            cout << "Bicolorable" << endl;
        }
        else
        {
            cout << "Not Bicolorable" << endl;
        }
        delete[] edges;
    }
}
```

13-Bonus : Tarzans

```
#include <bits/stdc++.h>
#define NIL -1
```

```
using namespace std;

vector<vector<int>> components;

class Graph
{
    int V;
    list<int> *adj;

    // A Recursive DFS based function used by SCC()
    void SCCUtil(int u, int disc[], int low[], stack<int> *st, bool stackMember[])
    {
        // A static variable is used for simplicity, we can avoid use
        // of static variable by passing a pointer.
        static int time = 0;

        // Initialize discovery time and low value
        disc[u] = low[u] = ++time;
        st->push(u);
        stackMember[u] = true;

        // Go through all vertices adjacent to this
        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
        {
            int v = *i; // v is current adjacent of 'u'

            // If v is not visited yet, then recur for it
            if (disc[v] == -1)
            {
                SCCUtil(v, disc, low, st, stackMember);

                // Check if the subtree rooted with 'v' has a
                // connection to one of the ancestors of 'u'
                // Case 1 (per above discussion on Disc and Low value)
                low[u] = min(low[u], low[v]);
            }

            // Update low value of 'u' only of 'v' is still in stack
            // (i.e. it's a back edge, not cross edge).
            // Case 2 (per above discussion on Disc and Low value)
            else if (stackMember[v] == true)
                low[u] = min(low[u], disc[v]);
        }

        // head node found, pop the stack and print an SCC
        int w = 0; // To store stack extracted vertices
        vector<int> temp;
        if (low[u] == disc[u])
        {
            while (st->top() != u)
            {
                w = (int)st->top();
                temp.push_back((w));
                stackMember[w] = false;
                st->pop();
            }
            w = (int)st->top();
            temp.push_back(w);
            stackMember[w] = false;
            st->pop();
        }
        components.push_back(temp);
    }

public:
    Graph(int V)
    {
        this->V = V;
        adj = new list<int>[V];
    } // Constructor
    void addEdge(int v, int w)
    {
        adj[v].push_back(w);
    } // function to add an edge to graph
    void SCC()
    {
        int *disc = new int[V];
        int *low = new int[V];
        bool *stackMember = new bool[V];
        stack<int> *st = new stack<int>();

        // Initialize disc and low, and stackMember arrays
        for (int i = 0; i < V; i++)
        {
            disc[i] = NIL;
            low[i] = NIL;
            stackMember[i] = false;
        }

        // Call the recursive helper function to find strongly
        // connected components in DFS tree with vertex 'i'
        for (int i = 0; i < V; i++)
            if (disc[i] == NIL)
                SCCUtil(i, disc, low, st, stackMember);
    } // prints strongly connected components
};

// Driver program to test above function
int main()
{
    int t;
    cin >> t;
    while (t--)
    {
        int n, m;
        cin >> n >> m;
        Graph g1(n);
        for (int i = 0; i < m; i++)
        {
            int a, b;
            g1.addEdge(a, b);
            g1.addEdge(b, a);
        }
        g1.SCC();

        for (int i = 0; i < components.size(); i++)
        {
            for (int j = 0; j < components[i].size(); j++)
            {
                cout << components[i][j] << " ";
            }
            cout << endl;
        }
    }
    return 0;
}
```