Assignment

Name: MINHAJUL ABEDIN BHUIYAN
Student ID: 210042148
Department: Computer Science & Engineering
Program: Software Engineering
Course: Data Structures and Algorithms Lab
Course No: CSE 4304
Lab Group: B
Lab No: 8
Topic: Graph Basic

ISLAMIC UNIVERSITY of TECHNOLOGY
Boardbazar, Gazipur, Bangladesh.

Problem A: Back to Underworld

The Vampires and Lykans are fighting each other to death. The war has become so fierce that, none knows who will win. The humans want to know who will survive finally. But humans are afraid of going to the battlefield.

So, they made a plan. They collected the information from the newspapers of Vampires and Lykans. They found the information about all the dual fights. Dual fight means a fight between a Lykan and a Vampire. They know the name of the dual fighters, but don't know which one of them is a Vampire or a Lykan.

So, the humans listed all the rivals. They want to find the maximum possible number of Vampires or Lykans.

Input

Input starts with an integer **T (≤ 10)**, denoting the number of test cases.

Each case contains an integer **n (1 ≤ n ≤ 10^5)**, denoting the number of dual fights. Each of the next **n** lines will contain two different integers **u v (1 ≤ u, v ≤ 20000)** denoting there was a fight between **u** and **v**. No rival will be reported more than once.

For each case, print the case number and the maximum possible members of any race.

Sample:

Input	Output
2 2 1 2 2 3 3 1 2 2 3 4 2	Case 1: 2 Case 2: 3

Dataset is huge, use faster I/O methods.

Solution:

```cpp
#include <iostream>
#include <bits/stdc++.h>

using namespace std;

int main()
{
    int t;
    cin >> t;

    for (int case_num = 1; case_num <= t; case_num++)
    {
        int n;
        cin >> n;

        unordered_map<int, int> rivalry_count;
        vector<int> fighters;

        for (int i = 0; i < n; i++)
        {
            int u, v;
            cin >> u >> v;
            fighters.push_back(u);
            fighters.push_back(v);
        }

        int max_rivals = 0;
        for (int fighter : fighters)
        {
            int count = ++rivalry_count[fighter];
            max_rivals = max(max_rivals, count);
        }

        cout << "Case " << case_num << ": " << max_rivals << endl;
    }

    return 0;
}
```

Explanation:

- The code addresses a scenario where humans want to predict the maximum number of either Vampires or Lykans that could exist after a series of battles. Each battle, or "dual fight," is between a fighter whose identity as a Vampire or Lykan is unknown.

- For each test case:

1. It starts by reading the number of dual fights that occurred (`n`) and initializes two important data structures:

   - An "unordered map" called `rivalry_count` is used to keep count of how many times each fighter has participated in a duel.

   - A "vector" named `fighters` is employed to maintain a list of all fighters involved in these duels.

2. The code proceeds to process each individual duel:

   - It records the two fighters involved in the duel.

   - It updates the duel count for each fighter in the `rivalry_count` map, essentially keeping track of how many times each fighter has fought.

   - Simultaneously, it maintains a running count of the maximum number of duels that any fighter has been involved in.

3. Once all the duels have been analyzed, the code is ready to provide the result for that specific test case. It does this by:

   - Displaying the case number (to identify which test case is being reported).

   - Reporting the maximum number of members that could belong to either the Vampires or Lykans, which is determined based on the highest duel count recorded in the `rivalry_count` map.

In summary, the code assists in processing information about dual fights, recording the participation of each fighter, and determining the largest possible group of either Vampires or Lykans for each test case.

Problem B: Kefa and Park

Kefa decided to celebrate his first big salary by going to the restaurant.

He lives by an unusual park. The park is a rooted tree consisting of n vertices with the root at vertex 1. Vertex 1 also contains Kefa's house. Unfortunaely for our hero, the park also contains cats. Kefa has already found out what are the vertices with cats in them.

The leaf vertices of the park contain restaurants. Kefa wants to choose a restaurant where he will go, but unfortunately he is very afraid of cats, so there is no way he will go to the restaurant if the path from the restaurant to his house contains more than m **consecutive** vertices with cats.

Your task is to help Kefa count the number of restaurants where he can go.

Input

The first line contains two integers, n and m ($2 \le n \le 10^5$, $1 \le m \le n$) — the number of vertices of the tree and the maximum number of consecutive vertices with cats that is still ok for Kefa.

The second line contains n integers $a_1, a_2, ..., a_n$, where each a_i either equals to 0 (then vertex i has no cat), or equals to 1 (then vertex i has a cat).

Next $n - 1$ lines contains the edges of the tree in the format "x_i y_i" (without the quotes) ($1 \le x_i, y_i \le n$, $x_i \ne y_i$), where x_i and y_i are the vertices of the tree, connected by an edge.

It is guaranteed that the given set of edges specifies a tree.

Output

A single integer — the number of distinct leaves of a tree the path to which from Kefa's home contains at most m consecutive vertices with cats.

Sample 1

Input	Output
4 1 1 1 0 0 1 2 1 3 1 4	2

Sample 2

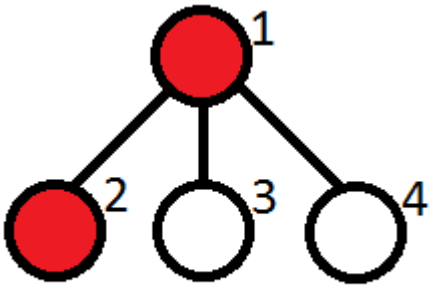
Input	Output
7 1 1 0 1 1 0 0 0 1 2 1 3 2 4 2 5 3 6 3 7	2

Note

Let us remind you that a *tree* is a connected graph on *n* vertices and *n* - 1 edge. A *rooted* tree is a tree with a special vertex called *root*. In a rooted tree among any two vertices connected by an edge, one vertex is a parent (the one closer to the root), and the other one is a child. A vertex is called a *leaf*, if it has no children.

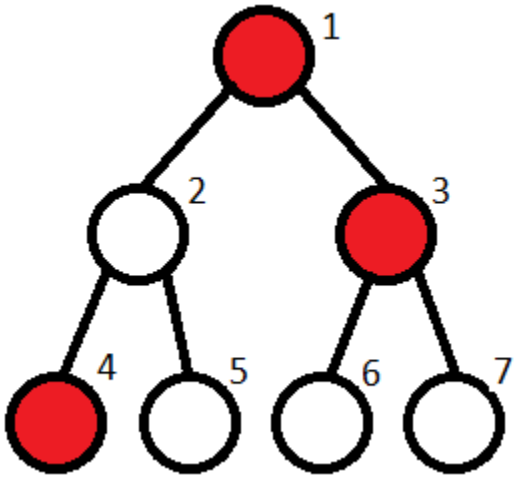
Note to the first sample test:

The vertices containing cats are marked red. The restaurants are at vertices 2, 3, 4. Kefa can't go only to the restaurant located at vertex 2.



Note to the second sample test:

The restaurants are located at vertices 4, 5, 6, 7. Kefa can't go to restaurants 6, 7.



Solution:

```cpp
#include <bits/stdc++.h>

using namespace std;

const int MAX_NODES = 100005;

vector<int> adjacency[MAX_NODES];
int restaurantCount, totalNodes, catStatus[MAX_NODES], maxConsecutiveCats;

void exploreTree(int currentNode, int consecutiveCats, int parent)
{
    if (catStatus[currentNode])
        consecutiveCats++;
    else
        consecutiveCats = 0;

    if (consecutiveCats > maxConsecutiveCats)
        return;

    if (adjacency[currentNode].size() == 1 && currentNode != 1)
    {
        if (consecutiveCats <= maxConsecutiveCats)
            restaurantCount++;
        return;
    }

    for (int nextNode : adjacency[currentNode])
    {
        if (nextNode != parent)
        {
            exploreTree(nextNode, consecutiveCats, currentNode);
        }
    }
}

int main()
{
    cin >> totalNodes >> maxConsecutiveCats;

    for (int i = 1; i <= totalNodes; i++)
        cin >> catStatus[i];

    for (int i = 1; i < totalNodes; i++)
```

```
    {
        int nodeA, nodeB;
        cin >> nodeA >> nodeB;
        adjacency[nodeA].push_back(nodeB);
        adjacency[nodeB].push_back(nodeA);
    }

    exploreTree(1, 0, 0);

    cout << restaurantCount;

    return 0;
}
```

Explanation:

- The code is designed to help Kefa find suitable restaurants in a tree-like park. This park has vertices, with some of them containing cats, and Kefa wants to visit leaf vertices (restaurants) while ensuring that there are not too many consecutive vertices with cats on the path from his house.

- It starts by reading input data, which includes details about the tree structure, information about the presence of cats at different vertices, and Kefa's tolerance for consecutive cats on the path to a restaurant.

- The code constructs the tree structure based on the provided information about how vertices are connected.

- The key function in the code is `exploreTree`, which recursively traverses the tree, starting from Kefa's house (vertex 1). It keeps track of the number of consecutive vertices with cats along the path. If this count exceeds Kefa's tolerance, the path is discontinued.

- When a leaf vertex (restaurant) is reached, and the number of consecutive cats on the path is within the tolerance, the code increments the `restaurantCount`. This variable represents the count of distinct restaurants that Kefa can visit while ensuring the consecutive cat constraint.

- After exploring the entire tree, the code prints the value of `restaurantCount`, providing Kefa with the number of suitable restaurants he can visit without exceeding the consecutive cat tolerance.

In summary, the code helps Kefa navigate the tree-like park, identifies appropriate restaurants, and ensures that he doesn't encounter too many consecutive cats on his path to these restaurants, all while making efficient use of recursive tree traversal.

Problem C: Oil Deposits

The GeoSurvComp geologic survey company is responsible for detecting underground oil deposits. GeoSurvComp works with one large rectangular region of land at a time, and creates a grid that divides the land into numerous square plots. It then analyzes each plot separately, using sensing equipment to determine whether or not the plot contains oil. A plot containing oil is called a pocket. If two pockets are adjacent, then they are part of the same oil deposit. Oil deposits can be quite large and may contain numerous pockets. Your job is to determine how many different oil deposits are contained in a grid.

Input

The input file contains one or more grids. Each grid begins with a line containing m and n, the number of rows and columns in the grid, separated by a single space. If m = 0 it signals the end of the input; otherwise $1 \leq m \leq 100$ and $1 \leq n \leq 100$. Following this are m lines of n characters each (not counting the end-of-line characters). Each character corresponds to one plot, and is either '*', representing the absence of oil, or '@', representing an oil pocket.

Output

For each grid, output the number of distinct oil deposits. Two different pockets are part of the same oil deposit if they are adjacent horizontally, vertically, or diagonally. An oil deposit will not contain more than 100 pockets.

Sample

Input	Output
1 1 * 3 5 *@*@* **@** *@*@* 1 8 @@****@* 5 5 ****@ *@@*@ *@**@ @@@*@ @@**@ 0 0	0 1 2 2

Solution:

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

const int MAX_M = 100;
const int MAX_N = 100;

int m, n;
char grid[MAX_M][MAX_N];
bool visited[MAX_M][MAX_N];

// Define possible neighbor directions (horizontally, vertically, and
diagonally)
int dr[] = {1, -1, 0, 0, 1, -1, 1, -1};
int dc[] = {0, 0, 1, -1, 1, -1, -1, 1};

// Function to perform BFS to explore the oil deposit
void bfs(int r, int c)
{
    visited[r][c] = true;
    queue<pair<int, int>> q;
    q.push({r, c});

    while (!q.empty())
    {
        int curR = q.front().first;
        int curC = q.front().second;
        q.pop();

        for (int i = 0; i < 8; i++)
        {
            int nr = curR + dr[i];
            int nc = curC + dc[i];

            if (nr >= 0 && nr < m && nc >= 0 && nc < n && !visited[nr][nc] &&
grid[nr][nc] == '@')
            {
                visited[nr][nc] = true;
                q.push({nr, nc});
            }
        }
    }
}
```

```
int countOilDeposits()
{
    int count = 0;

    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (!visited[i][j] && grid[i][j] == '@')
            {
                count++;
                bfs(i, j);
            }
        }
    }

    return count;
}

int main()
{
    while (cin >> m >> n && m > 0)
    {
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                cin >> grid[i][j];
                visited[i][j] = false;
            }
        }

        int result = countOilDeposits();
        cout << result << endl;
    }

    return 0;
}
```

Explanation:

- The code is engineered to handle a series of grids, each of which represents a distinct area of land. Within these grids, the primary objective is to identify and quantify distinct underground oil deposits accurately.

- For each individual grid, the code systematically conducts the following steps:

  - It initiates by extracting crucial grid information, namely the number of rows (`m`) and columns (`n`). These dimensions establish the grid's structure.

  - The presence of oil is indicated by the symbol '@', whereas unoccupied land is denoted by '*'.

- The central goal of the code is to ascertain how many discrete oil deposits are present within the grid. An oil deposit is characterized as a group of oil pockets that are interconnected through horizontal, vertical, or diagonal adjacency.

- To achieve this, the code employs a Breadth-First Search (BFS) strategy. The exploration begins from an oil pocket, which is marked as visited. Subsequently, neighboring plots are methodically inspected to identify and connect adjacent oil pockets.

- The process of exploration persists until all oil pockets within the same deposit are covered. This ensures that they are counted as a unified oil deposit rather than individual pockets.

- Upon the thorough examination of the entire grid, the code diligently computes and communicates the total count of distinct oil deposits identified.

- This procedural cycle iterates for each grid within the input. The code proceeds to process grids until it encounters a specific case where 'm' is set to 0, which distinctly marks the completion of the input and the termination of the program.

In summary, the code is engineered to systematically assess grids for the presence of oil deposits, employing BFS to establish connectivity among adjacent pockets, and culminating in the precise enumeration of separate oil deposits within each grid. All of this is carried out in strict accordance with the connectivity criteria stipulated in the problem statement.

Problem D: Travelling Cost

The government of **Spoj_land** has selected a number of locations in the city for road construction and numbered those locations as 0, 1, 2, 3, ... 500.

Now, they want to construct roads between various pairs of location (say **A** and **B**) and have fixed the cost for travelling between those pair of locations from either end as **W unit**.

Now, Rohit being a curious boy wants to find the minimum cost for travelling from location **U** (source) to **Q** number of other locations (destination).

Input

First line contains **N**, the number of roads that government constructed. Next N line contains three integers **A**, B, and **W**.

A and B represent the locations between which the road was constructed and W is the fixed cost for travelling from A to B or from B to A.

Next line contains an integer **U** from where Rohit wants to travel to other locations. Next line contains **Q**, the number of queries (finding cost) that he wants to perform. Next **Q** lines contain an integer **V** (destination) for which minimum cost is to be found **from U**.

Output

Print the required answer in each line. If he can't travel from location U to V by any means then, print '**NO PATH**' without quotes.

Sample

Input	**Output**
7 0 1 4 0 3 8 1 4 1 1 2 2 4 2 3 2 5 3 3 4 2 0 4 1 4 5 7	4 5 9 NO PATH

Solution:

```
#include <bits/stdc++.h>

using namespace std;

const int N = 505;
const long long INF = 1e15;

long long distances[N][N];

void floydWarshall(int n)
{
    for (int k = 0; k < n; ++k)
    {
        for (int i = 0; i < n; ++i)
        {
            for (int j = 0; j < n; ++j)
            {
                distances[i][j] = min(distances[i][j], distances[i][k] +
distances[k][j]);
            }
        }
    }
}

int main()
{
    ios_base::sync_with_stdio(false);
    int n;
    cin >> n;

    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
        {
            distances[i][j] = INF;
        }
    }

    for (int i = 0; i < n; ++i)
    {
        int a, b;
        long long c;
        cin >> a >> b >> c;
```

```
        distances[a][b] = min(distances[a][b], c);
        distances[b][a] = min(distances[b][a], c);
    }

    floydWarshall(n);

    int u, q;
    cin >> u >> q;

    while (q--)
    {
        int v;
        cin >> v;

        if (distances[u][v] != INF)
        {
            cout << distances[u][v] << endl;
        }
        else
        {
            cout << "NO PATH" << endl;
        }
    }

    return 0;
}
```

Explanation:

The problem involves a city with various locations marked by numbers from 0 to 500. The government is building roads between these locations, and they've set fixed costs for traveling between any pair of locations.

- The objective is to find the minimum cost of traveling from one specific location, U (the source), to multiple other locations (the destinations).

- The code is based on the Floyd-Warshall algorithm, which calculates the shortest paths between all pairs of locations.

- It starts by reading the data: the number of roads constructed (N) and the details of each road (locations A and B, and the cost W).

- The code initializes a 2D array called 'distances' to store the minimum costs between locations. It sets initial values to infinity (INF) for all pairs.

- Then, it updates 'distances' with the road costs read from the input.

- The core of the code is the Floyd-Warshall algorithm, which iteratively explores all possible paths between locations and updates the minimum costs. This step ensures that you can find the shortest path between any pair of locations.

- It takes into account the road construction costs and efficiently calculates the shortest paths.

- The code then reads Rohit's starting location (U) and the number of queries (Q) for which he wants to find the minimum costs.

- For each query, it reads a destination location (V) and checks if there's a valid path between U and V.

- If there's a valid path, it prints the minimum cost for traveling from U to V. If no path exists, it prints "NO PATH."

- The code repeats this process for all queries, providing the minimum costs or indicating when a path doesn't exist.

In summary, the code efficiently calculates the minimum costs of traveling between various locations in a city using the Floyd-Warshall algorithm. It provides answers for multiple queries and handles cases where there is no valid path between locations.