Department of Computer Science and Engineering

Name:- Ripunjai Tiwari   UID:- 22BCS10216   Section:- 22KPIT-901-A

DAY-7

Q.1 **Find the Town Judge**

In a town, there are n people labeled from 1 to n. There is a rumor that one of these people is secretly the town judge.

If the town judge exists, then:

1. The town judge trusts nobody.
2. Everybody (except for the town judge) trusts the town judge.
3. There is exactly one person that satisfies properties 1 and 2.

You are given an array trust where trust[i] = [ai, bi] representing that the person labeled ai trusts the person labeled bi. If a trust relationship does not exist in trust array, then such a trust relationship does not exist.

Return the label of the town judge if the town judge exists and can be identified, or return -1 otherwise.

Example 1:

Input: n = 2, trust = [[1,2]]

Output: 2

Example 2:

Input: n = 3, trust = [[1,3],[2,3]]

Output: 3

Example 3:

Input: n = 3, trust = [[1,3],[2,3],[3,1]]

Output: -1

Constraints:

- $1 <= n <= 1000$
- $0 <= trust.length <= 1e4$
- trust[i].length == 2
- All the pairs of trust are unique.
- ai != bi
- $1 <= ai, bi <= n$

Program Code:-

```cpp
#include <iostream>
#include <vector>
using namespace std;

int findJudge(int n, vector<vector<int>>& trust)
   { if (n == 1 && trust.empty()) {
      return 1; // If there's only one person and no trust relationships, they are the judge.
   }

   vector<int> trustCounts(n + 1, 0); // Array to count trust balance for each person.

   for (const auto& t : trust) {
      int a = t[0], b = t[1];
      trustCounts[a]--; // a trusts someone, so decrement their trust count.
      trustCounts[b]++; // b is trusted by someone, so increment their trust count.
   }

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

```cpp
        if (trustCounts[i] == n - 1) {
            return i; // The judge is trusted by everyone except themselves.
        }
    }

    return -1; // No judge found.
}

int main()
{ int n = 3;
    vector<vector<int>> trust = {{1, 3}, {2, 3}};

    int judge = findJudge(n, trust);
    cout << "The town judge is: " << judge << endl;

    return 0;
}
```
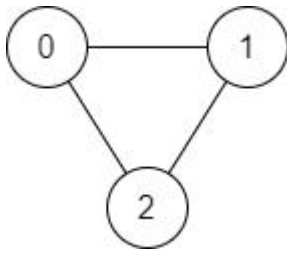
Output:-

```
The town judge is: 3
```

Q2 **Find if Path Exists in Graph**

There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to n - 1 (inclusive). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [ui, vi] denotes a bi-directional edge between vertex ui and vertex vi. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.

You want to determine if there is a valid path that exists from vertex source to vertex destination.

Given edges and the integers n, source, and destination, return true if there is a valid path from source to destination, or false otherwise.

Example 1:

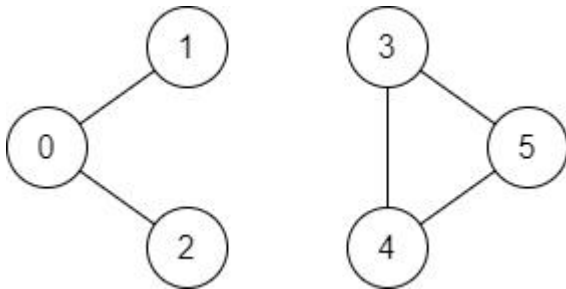**Input**: n = 3, edges = [[0,1],[1,2],[2,0]], source = 0, destination = 2

Output: true

Explanation: There are two paths from vertex 0 to vertex 2:

- 0 → 1 → 2

- 0 → 2

Example 2:

Input: n = 6, edges = [[0,1],[0,2],[3,5],[5,4],[4,3]], source = 0, destination = 5

Output: false

Explanation: There is no path from vertex 0 to vertex 5.

Constraints:

- 1 <= n <= 2 * 1e5
- 0 <= edges.length <= 2 * 1e5
- edges[i].length == 2
- 0 <= ui, vi <= n - 1
- ui != vi
- 0 <= source, destination <= n - 1
- There are no duplicate edges.
- There are no self edges.

Program Code:-

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
using namespace std;

bool validPath(int n, vector<vector<int>>& edges, int source, int destination)
   { if (source == destination) {
      return true; // If source and destination are the same, the path exists.
   }

   // Create an adjacency list for the graph.
   unordered_map<int, vector<int>> graph;
   for (const auto& edge : edges) {
      graph[edge[0]].push_back(edge[1]);
      graph[edge[1]].push_back(edge[0]);
   }

   // BFS to check for a path from source to destination.
   queue<int> q;
   vector<bool> visited(n, false);

   q.push(source);
   visited[source] = true;

   while (!q.empty()) {
      int current = q.front();
      q.pop();

      for (const int& neighbor : graph[current])
         { if (!visited[neighbor]) {
            if (neighbor == destination) {
               return true; // Found the destination.
            }
            q.push(neighbor);
            visited[neighbor] = true;
         }
      }
   }

   return false; // No path found.
}
```

```cpp
int main()
  { int n, m;
  cout << "Enter number of nodes (n): ";
  cin >> n;
  cout << "Enter number of edges: ";
  cin >> m;

  vector<vector<int>> edges;
  cout << "Enter the edges (u v):" << endl;
  for (int i = 0; i < m; ++i) {
      int u, v;
      cin >> u >> v;
      edges.push_back({u, v});
  }

  int source, destination;
  cout << "Enter source: ";
  cin >> source;
  cout << "Enter destination: ";
  cin >> destination;

  bool result = validPath(n, edges, source, destination);
  cout << (result ? "true" : "false") << endl;

  return 0;
}}
```

Output:-


```
Enter number of nodes (n): 6
Enter number of edges: 5
Enter the edges (u v):
0 1
0 2
3 5
5 4
4 3
Enter source: 0
Enter destination: 5
```

```
false
```

Given an m x n grid of characters board and a string word, return true if word exists in the grid.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:

Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"

Output: true

Example 2:

Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"

Output: true

Example 3:

Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"

Output: false

Constraints:

- m == board.length
- n = board[i].length
- 1 <= m, n <= 6
- 1 <= word.length <= 15
- board and word consists of only lowercase and uppercase English letters.

Follow up: Could you use search pruning to make your solution faster with a larger board?

Program Code:-

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;

bool dfs(vector<vector<char>>& board, string& word, int i, int j, int index) {
    // Check if the current position is out of bounds or the character doesn't match
    if (i < 0 || i >= board.size() || j < 0 || j >= board[0].size() || board[i][j] != word[index])
        { return false;
    }

    // Check if all characters in the word have been found
    if (index == word.size() - 1) {
        return true;
    }

    // Temporarily mark the current cell as visited
```

```cpp
    char temp = board[i][j];
    board[i][j] = '#';

    // Explore all four possible directions (up, down, left, right)
    bool found = dfs(board, word, i + 1, j, index + 1) ||
            dfs(board, word, i - 1, j, index + 1) ||
            dfs(board, word, i, j + 1, index + 1) ||
            dfs(board, word, i, j - 1, index + 1);

    // Restore the original value of the cell
    board[i][j] = temp;

    return found;
}

bool exist(vector<vector<char>>& board, string word)
    { for (int i = 0; i < board.size(); i++) {
        for (int j = 0; j < board[0].size(); j++) {
            // Start a DFS if the first character matches
            if (board[i][j] == word[0] && dfs(board, word, i, j, 0))
                { return true;
            }
        }
    }
    return false;
}

int main()
    { int m, n;
    cout << "Enter the number of rows (m): ";
    cin >> m;
    cout << "Enter the number of columns (n): ";
    cin >> n;

    vector<vector<char>> board(m, vector<char>(n));
    cout << "Enter the board characters row by row:" << endl;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            { cin >> board[i][j];
        }
    }

    string word;
    cout << "Enter the word to search: ";
    cin >> word;
```

```
    if (exist(board, word))
      { cout << "true" << endl;
    } else {
      cout << "false" << endl;
    }

    return 0;
}
```

Output:

```
Enter the number of rows (m): 3
Enter the number of columns (n): 4
Enter the board characters row by row:
A B C E
S F C S
A D E E
Enter the word to search: ABCCED
```

```
true
```

Q4. Max Area of Island

You are given an m x n binary matrix grid. An island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The area of an island is the number of cells with a value 1 in the island.

Return the maximum area of an island in grid. If there is no island, return 0.

Example 1:

0	0	1	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	0	1	0	1	0	0
0	1	0	0	1	1	0	0	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0

Input: grid =
[[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,1,1,0,1,0,0,0,0,0,0,0,0],[0,1,0,0,1,1,0,0,1,0,1,0,0],[0,1,0,0,1,1,0,0,1,1,1,0,0],[0,0,0,0,0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,0,0,0,0,0,0,1,1,0,0,0,0]]

Output: 6

Explanation: The answer is not 11, because the island must be connected 4-directionally.

Example 2:

Input: grid = [[0,0,0,0,0,0,0,0]]

Output: 0

Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 50
- grid[i][j] is either 0 or 1.

Program Code:-

```cpp
#include <iostream>
#include <vector>
using namespace std;

int dfs(vector<vector<int>>& grid, int i, int j) {
    // Check if the current position is out of bounds or water (0)
    if (i < 0 || i >= grid.size() || j < 0 || j >= grid[0].size() || grid[i][j] == 0)
        { return 0;
    }

    // Mark the current cell as visited by setting it to 0
    grid[i][j] = 0;

    // Explore all four directions and sum up the area
    int area = 1;
    area += dfs(grid, i + 1, j); // Down
    area += dfs(grid, i - 1, j); // Up
    area += dfs(grid, i, j + 1); // Right
    area += dfs(grid, i, j - 1); // Left

    return area;
}

int maxAreaOfIsland(vector<vector<int>>& grid)
    { int maxArea = 0;

    for (int i = 0; i < grid.size(); i++) {
        for (int j = 0; j < grid[0].size(); j++) {
            if (grid[i][j] == 1) { // If it's land, start DFS
                maxArea = max(maxArea, dfs(grid, i, j));
            }
        }
    }

    return maxArea;
}

int main()
    { int m, n;
    cout << "Enter the number of rows (m): ";
    cin >> m;
    cout << "Enter the number of columns (n): ";
    cin >> n;
```

```cpp
    vector<vector<int>> grid(m, vector<int>(n));
    cout << "Enter the grid values row by row (0 or 1):" << endl;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            { cin >> grid[i][j];
        }
    }

    cout << "Maximum area of island: " << maxAreaOfIsland(grid) << endl;

    return 0;
}
```
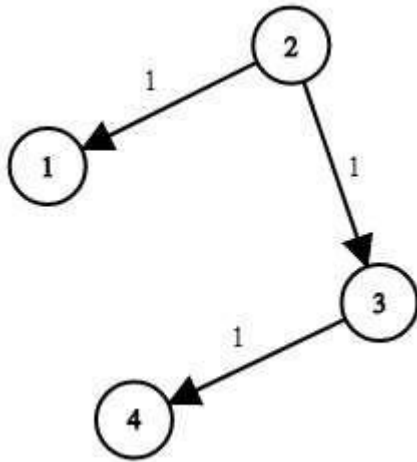
Output:-

Q5. <u>**Network Delay Time**</u>

You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k. Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

Example 1:

Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2

Output: 2

Example 2:

Input: times = [[1,2,1]], n = 2, k = 1

Output: 1

Example 3:

Input: times = [[1,2,1]], n = 2, k = 2

Output: -1

Constraints:

- 1 <= k <= n <= 100
- 1 <= times.length <= 6000
- times[i].length == 3
- 1 <= ui, vi <= n
- ui != vi
- 0 <= wi <= 100
- All the pairs (ui, vi) are unique. (i.e., no multiple edges.)

Program Code:-
```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
```

```cpp
#include <climits>

using namespace std;

int networkDelayTime(vector<vector<int>>& times, int n, int k) {
    // Create an adjacency list
    unordered_map<int, vector<pair<int, int>>> graph;
    for (const auto& time : times) {
        graph[time[0]].emplace_back(time[1], time[2]);
    }
    // Min-heap to store (time, node)
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> minHeap;
    minHeap.emplace(0, k);

    // Distance array to keep track of the shortest time to each node
    vector<int> dist(n + 1, INT_MAX);
    dist[k] = 0;

    while (!minHeap.empty()) {
        auto [currentTime, node] = minHeap.top();
        minHeap.pop();

        // If this time is already longer than the shortest known time, skip
        if (currentTime > dist[node]) continue;

        // Traverse neighbors
        for (const auto& [neighbor, weight] : graph[node])
            { if (currentTime + weight < dist[neighbor]) {
                dist[neighbor] = currentTime + weight;
                minHeap.emplace(dist[neighbor], neighbor);
            }
        }
    }
    // Find the maximum time needed to reach any node
    int maxTime = 0;
    for (int i = 1; i <= n; ++i) {
        if (dist[i] == INT_MAX) return -1; // If a node is unreachable
```

```cpp
        maxTime = max(maxTime, dist[i]);
    }
    return maxTime;
}

int main()
    { int n, m, k;
    cout << "Enter the number of nodes (n): ";
    cin >> n;
    cout << "Enter the number of edges: ";
    cin >> m;

    vector<vector<int>> times(m, vector<int>(3));
    cout << "Enter the edges (source, target, weight):" << endl;
    for (int i = 0; i < m; ++i) {
        cin >> times[i][0] >> times[i][1] >> times[i][2];
    }
    cout << "Enter the starting node (k): ";
    cin >> k;

    int result = networkDelayTime(times, n, k);
    cout << "The minimum time for all nodes to receive the signal is: " << result
<< endl;

    return 0;
}
```

Output:-

```
Enter the number of nodes (n): 4
Enter the number of edges: 3
Enter the edges (source, target, weight):
2 1 1
2 3 1
3 4 1
Enter the starting node (k): 2

The minimum time for all nodes to receive the signal is: 2
```