

DOMAIN WINTER WINNING CAMP

ASSIGNMENT DAY 7

1) There is an undirected star graph consisting of n nodes labeled from 1 to n . A star graph is a graph where there is one center node and exactly $n - 1$ edges that connect the center node with every other node.

You are given a 2D integer array `edges` where each `edges[i] = [ui, vi]` indicates that there is an edge between the nodes `ui` and `vi`. Return the center of the given star graph.

SOLUTION

```
#include <iostream>
#include <vector>
using namespace std;
int findCenter(vector<vector<int>>& edges) {
    // The center node will appear in the first two edges
    if (edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1]) {
        return edges[0][0];
    } else {
        return edges[0][1];
    }
}
int main() {
    vector<vector<int>> edges = {{1, 2}, {2, 3}, {4, 2}};

    int center = findCenter(edges);
    cout << "The center of the star graph is: " << center << endl;

    return 0;
}
```

OUTPUT

```
The center of the star graph is: 2
```

```
=== Code Execution Successful ===
```

2) 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

SOLUTION

```
#include <iostream>
#include <vector>
using namespace std;
int findJudge(int n, vector<vector<int>>& trust) {
    vector<int> trustCount(n + 1, 0);
    for (const auto& relation : trust) {
        int a = relation[0];
        int b = relation[1];
        trustCount[a]--;
        trustCount[b]++;
    }
}
```

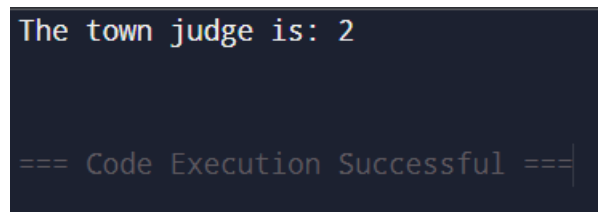
```

    for (int i = 1; i <= n; ++i) {
        if (trustCount[i] == n - 1) {
            return i;
        }
    }
    return -1;
}

int main() {
    // Example input
    int n = 2;
    vector<vector<int>> trust = {{1, 2}};
    int judge = findJudge(n, trust);
    cout << "The town judge is: " << judge << endl;
    return 0;
}

```

OUTPUT



```

The town judge is: 2

=== Code Execution Successful ===

```

3) You are given an image represented by an $m \times n$ grid of integers `image`, where `image[i][j]` represents the pixel value of the image. You are also given three integers `sr`, `sc`, and `color`. Your task is to perform a flood fill on the image starting from the pixel `image[sr][sc]`.

To perform a flood fill:

Begin with the starting pixel and change its color to `color`.

Perform the same process for each pixel that is directly adjacent (pixels that share a side with the original pixel, either horizontally or vertically) and shares the same color as the starting pixel.

Keep repeating this process by checking neighboring pixels of the updated pixels and modifying their color if it matches the original color of the starting pixel.

The process stops when there are no more adjacent pixels of the original color to update.

Return the modified image after performing the flood fill.

Example 1:

Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2

Output: [[2,2,2],[2,2,0],[2,0,1]]

SOLUTION

```
#include <iostream>
#include <vector>
using namespace std;
class Solution {
public:
    void dfs(vector<vector<int>>& image, int sr, int sc, int originalColor, int
newColor) {
        if (sr < 0 || sr >= image.size() || sc < 0 || sc >= image[0].size() ||
image[sr][sc] != originalColor) {
            return;
        }
        image[sr][sc] = newColor;
        dfs(image, sr - 1, sc, originalColor, newColor); // Up
        dfs(image, sr + 1, sc, originalColor, newColor); // Down
        dfs(image, sr, sc - 1, originalColor, newColor); // Left
        dfs(image, sr, sc + 1, originalColor, newColor); // Right
    }
    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int
color) {
        int originalColor = image[sr][sc];
        if (originalColor != color) {
            dfs(image, sr, sc, originalColor, color);
        }
    }
};
```

```

    }
    return image;
}
};

int main() {
    vector<vector<int>> image = {
        {1, 1, 1},
        {1, 1, 0},
        {1, 0, 1}
    };
    int sr = 1, sc = 1, color = 2;
    Solution solution;
    vector<vector<int>> result = solution.floodFill(image, sr, sc, color);
    for (const auto& row : result) {
        for (const auto& pixel : row) {
            cout << pixel << " ";
        }
        cout << endl;
    }
    return 0;
}

```

OUTPUT

```

2 2 2
2 2 0
2 0 1

=== Code Execution Successful ===

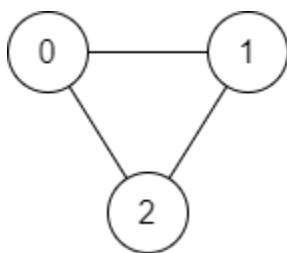
```

4) 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:

- $0 \rightarrow 1 \rightarrow 2$

- $0 \rightarrow 2$

SOLUTION:

```
#include <iostream>
```

```
#include <vector>
```

```
#include <queue>
```

```
#include <unordered_map>
```

```
using namespace std;
```

```
bool validPath(int n, vector<vector<int>>& edges, int source, int destination) {
```

```
    unordered_map<int, vector<int>> adjList;
```

```
    for (const auto& edge : edges) {
```

```
        adjList[edge[0]].push_back(edge[1]);
```

```
        adjList[edge[1]].push_back(edge[0]);
```

```
    }
```

```

vector<bool> visited(n, false);
queue<int> q;
q.push(source);
visited[source] = true;
while (!q.empty()) {
    int current = q.front();
    q.pop();
    if (current == destination) {
        return true;
    }
    for (int neighbor : adjList[current]) {
        if (!visited[neighbor]) {
            visited[neighbor] = true;
            q.push(neighbor);
        }
    }
}
return false;
}

int main() {
    int n = 3;
    vector<vector<int>> edges = {{0, 1}, {1, 2}, {2, 0}};
    int source = 0;
    int destination = 2;
    bool result = validPath(n, edges, source, destination);
    cout << (result ? "true" : "false") << endl;
    return 0;
}

```

OUTPUT:

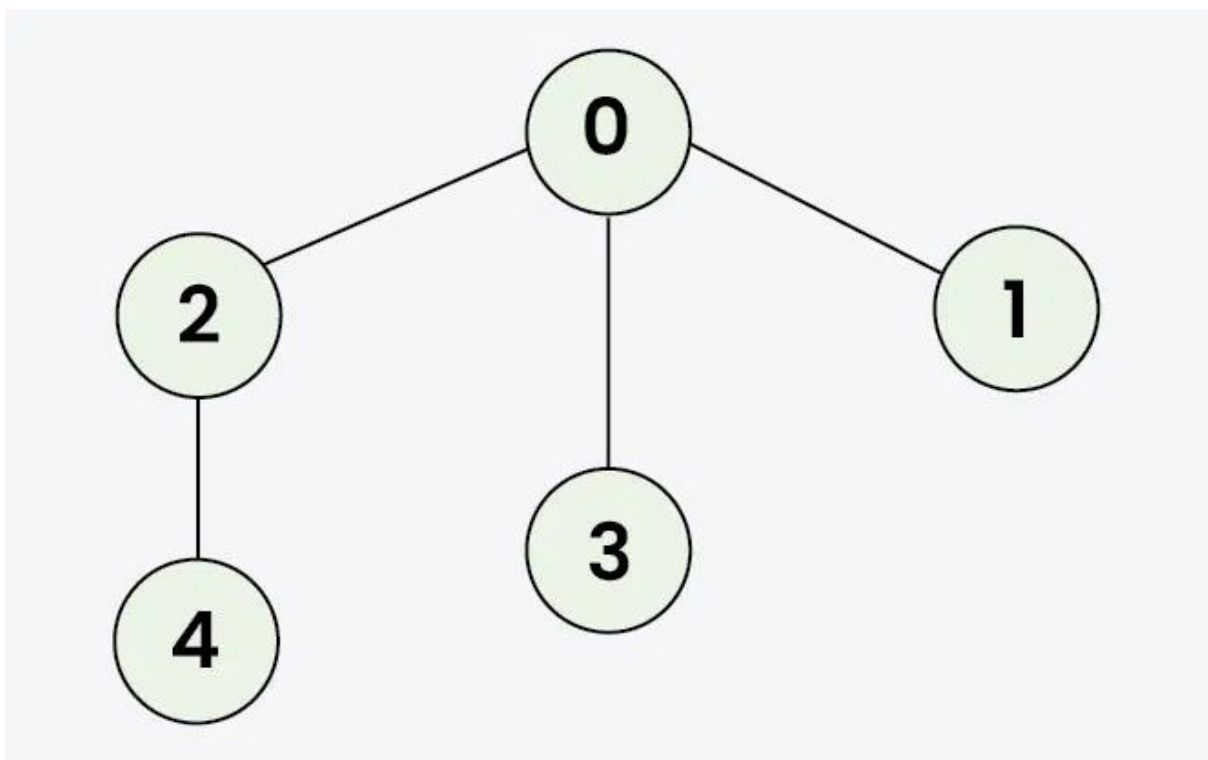
```
true
```

```
=== Code Execution Successful ===
```

5) Given a connected undirected graph represented by an adjacency list `adj`, which is a vector of vectors where each `adj[i]` represents the list of vertices connected to vertex `i`. Perform a Breadth First Traversal (BFS) starting from vertex `0`, visiting vertices from left to right according to the adjacency list, and return a list containing the BFS traversal of the graph.

Note: Do traverse in the same order as they are in the adjacency list.

Example 1:



Input: `adj = [[2,3,1], [0], [0,4], [0], [2]]`

Output: `[0, 2, 3, 1, 4]`

SOLUTION:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
vector<int> bfsTraversal(int n, vector<vector<int>>& adj) {
    vector<int> bfsResult;
    vector<bool> visited(n, false);
    queue<int> q;
    q.push(0);
    visited[0] = true;
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        bfsResult.push_back(current);
        for (int neighbor : adj[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
    return bfsResult;
}
int main() {
    vector<vector<int>> adj = {
        {2, 3, 1},
        {0},
        {0, 4},
        {0},
        {2}
```

```
};  
int n = adj.size();  
vector<int> result = bfsTraversal(n, adj);  
cout << "BFS Traversal: ";  
for (int node : result) {  
    cout << node << " ";  
}  
cout << endl;  
return 0;  
}
```

OUTPUT:

```
BFS Traversal: 0 2 3 1 4
```

```
=== Code Execution Successful ===|
```