DOMAIN WINTER WINNING CAMP ASSIGNMENT

Student Name: Jaya Jagriti **UID: 22BCS15730**
Branch: BE-CSE **Section/Group: 22BCS_FL_IOT-603/B**
Semester: 5th

DAY-7

DSA Questions(Trees)

Graph

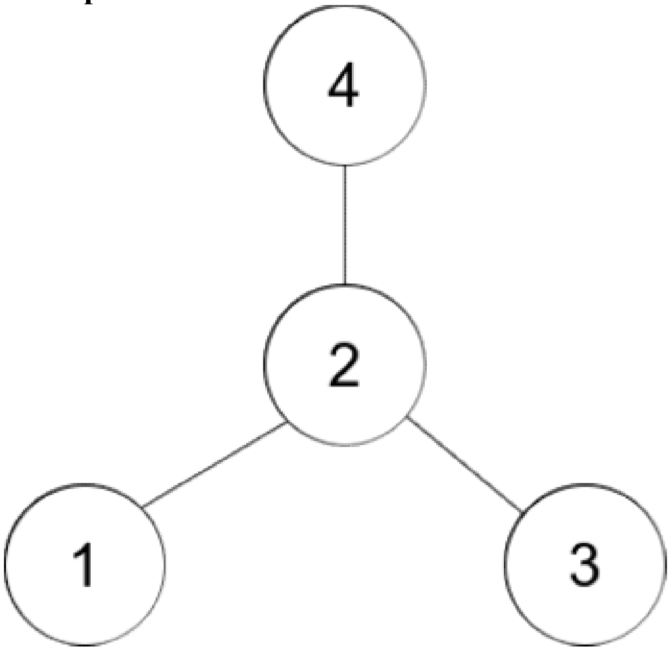
Very Easy:

1.Find Center of Star Graph

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.

Example 1:



Input: edges = [[1,2],[2,3],[4,2]]
Output: 2
Explanation: As shown in the figure above, node 2 is connected to every other node, so 2 is the center.

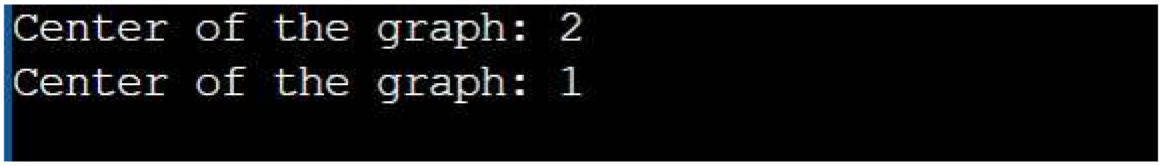
Example 2:
Input: edges = [[1,2],[5,1],[1,3],[1,4]]
Output: 1

Constraints:

2. 3 <= n <= 1e5
3. edges.length == n - 1
4. edges[i].length == 2
5. 1 <= ui, vi <= n
6. ui != vi
7. The given edges represent a valid star graph.

CODE:
```
#include <iostream>
#include <vector>

using namespace std;

int findCenter(vector<vector<int>>& edges) {
    // The center node will appear in both the first two edges.
    if (edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1]) {
        return edges[0][0];
    }
    return edges[0][1];
}

int main() {
    // Example 1
    vector<vector<int>> edges1 = {{1, 2}, {2, 3}, {4, 2}};
    cout << "Center of the graph: " << findCenter(edges1) << endl;

    // Example 2
    vector<vector<int>> edges2 = {{1, 2}, {5, 1}, {1, 3}, {1, 4}};
    cout << "Center of the graph: " << findCenter(edges2) << endl;

    return 0;
}
```

OUTPUT:



```
Center of the graph: 2
Center of the graph: 1
```

2. 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

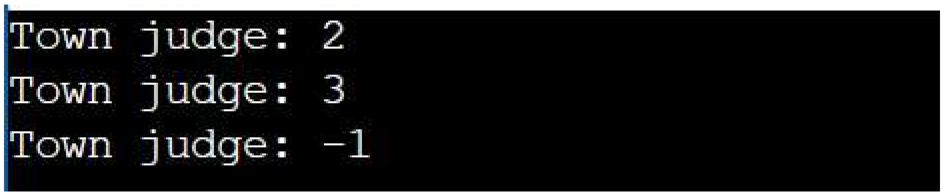
Example 3:
Input: n = 3, trust = [[1,3],[2,3],[3,1]]
Output: -1

Constraints:

1. 1 <= n <= 1000
2. 0 <= trust.length <= 1e4
3. trust[i].length == 2
4. All the pairs of trust are unique.
5. ai != bi
6. 1 <= ai, bi <= n

CODE:
```
#include <iostream>
#include <vector>

using namespace std;

int findJudge(int n, vector<vector<int>>& trust) {
    // Create two arrays to track trust counts.
    vector<int> trustCount(n + 1, 0);

    // Process the trust relationships.
    for (const auto& t : trust) {
        trustCount[t[0]]--; // The person who trusts someone loses a point.
        trustCount[t[1]]++; // The person who is trusted gains a point.
    }

    // Check for the town judge.
    for (int i = 1; i <= n; ++i) {
        if (trustCount[i] == n - 1) {
            return i; // The judge is trusted by everyone else and trusts nobody.
        }
    }

    return -1; // No judge found.
}

int main() {
    // Example 1
    int n1 = 2;
    vector<vector<int>> trust1 = {{1, 2}};
    cout << "Town judge: " << findJudge(n1, trust1) << endl;

    // Example 2
    int n2 = 3;
    vector<vector<int>> trust2 = {{1, 3}, {2, 3}};
    cout << "Town judge: " << findJudge(n2, trust2) << endl;

    // Example 3
    int n3 = 3;
    vector<vector<int>> trust3 = {{1, 3}, {2, 3}, {3, 1}};
    cout << "Town judge: " << findJudge(n3, trust3) << endl;

    return 0;
}
```

OUTPUT:



```
Town judge: 2
Town judge: 3
Town judge: -1
```

3.Flood Fill - link

You are given an image represented by an m x 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]]

Explanation:
From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.

Note the bottom corner is not colored 2, because it is not horizontally or vertically connected to the starting pixel.

Example 2:

Input: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0
Output: [[0,0,0],[0,0,0]]

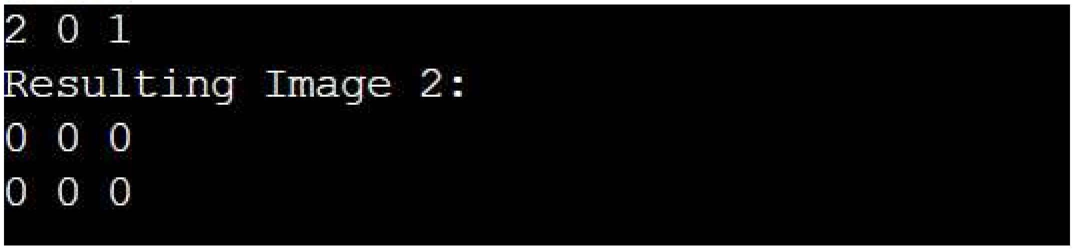
Explanation:
The starting pixel is already colored with 0, which is the same as the target color. Therefore, no changes are made to the image.

Constraints:
- m == image.length
- n == image[i].length
- 1 <= m, n <= 50
- 0 <= image[i][j], color < 2^16
- 0 <= sr < m
- 0 <= sc < n

CODE:
```
#include <iostream>
#include <vector>
```

```
using namespace std;

void dfs(vector<vector<int>>& image, int sr, int sc, int originalColor, int newColor) {
    // Check bounds and color conditions
    if (sr < 0 || sr >= image.size() || sc < 0 || sc >= image[0].size() || image[sr][sc] != originalColor ||
image[sr][sc] == newColor) {
        return;
    }

    // Change the color of the current pixel
    image[sr][sc] = newColor;

    // Recursive calls for the 4 directions
    dfs(image, sr + 1, sc, originalColor, newColor); // Down
    dfs(image, sr - 1, sc, originalColor, newColor); // Up
    dfs(image, sr, sc + 1, originalColor, newColor); // Right
    dfs(image, sr, sc - 1, originalColor, newColor); // Left
}

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() {
    // Example 1
    vector<vector<int>> image1 = {{1, 1, 1}, {1, 1, 0}, {1, 0, 1}};
    int sr1 = 1, sc1 = 1, color1 = 2;
    vector<vector<int>> result1 = floodFill(image1, sr1, sc1, color1);

    cout << "Resulting Image 1:" << endl;
    for (const auto& row : result1) {
        for (int pixel : row) {
            cout << pixel << " ";
        }
        cout << endl;
    }

    // Example 2
    vector<vector<int>> image2 = {{0, 0, 0}, {0, 0, 0}};
    int sr2 = 0, sc2 = 0, color2 = 0;
    vector<vector<int>> result2 = floodFill(image2, sr2, sc2, color2);

    cout << "Resulting Image 2:" << endl;
    for (const auto& row : result2) {
        for (int pixel : row) {
            cout << pixel << " ";
        }
        cout << endl;
    }

    return 0;
}
```

OUTPUT:



```
2 0 1
Resulting Image 2:
0 0 0
0 0 0
```

Easy

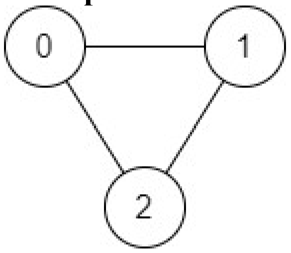
4.Find if Path Exists in Graph

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.

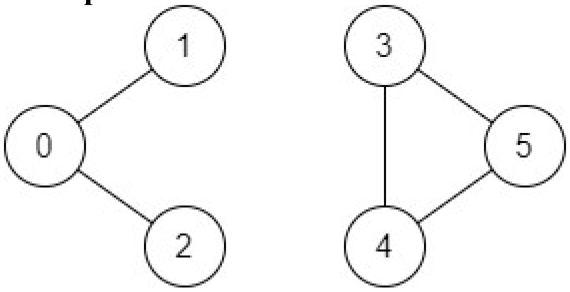
Example 1:



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:

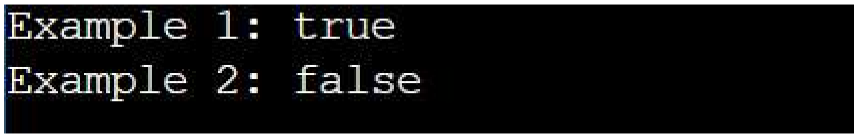
- 1 <= n <= 2 * 1e5
- 0 <= edges.length <= 2 * 1e5
- edges[i].length == 2
- 0 <= ui, vi <= n - 1
- ui != vi
- 0 <= source, destination <= n - 1
- There are no duplicate edges.
- There are no self edges.

CODE:

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <unordered_set>

using namespace std;

bool validPath(int n, vector<vector<int>>& edges, int source, int destination) {
   // Create an adjacency list to represent 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]);
   }

   // Use BFS to find the path from source to destination
   queue<int> q;
   unordered_set<int> visited;

   q.push(source);
   visited.insert(source);
```

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

        // If we reach the destination, return true
        if (current == destination) {
            return true;
        }

        // Visit all neighbors of the current node
        for (int neighbor : graph[current]) {
            if (!visited.count(neighbor)) {
                q.push(neighbor);
                visited.insert(neighbor);
            }
        }
    }

    // If the destination was not reached, return false
    return false;
}

int main() {
    // Example 1
    int n1 = 3;
    vector<vector<int>> edges1 = {{0, 1}, {1, 2}, {2, 0}};
    int source1 = 0, destination1 = 2;
    cout << "Example 1: " << (validPath(n1, edges1, source1, destination1) ? "true" : "false") << endl;

    // Example 2
    int n2 = 6;
    vector<vector<int>> edges2 = {{0, 1}, {0, 2}, {3, 5}, {5, 4}, {4, 3}};
    int source2 = 0, destination2 = 5;
    cout << "Example 2: " << (validPath(n2, edges2, source2, destination2) ? "true" : "false") << endl;

    return 0;
}
```

OUTPUT:



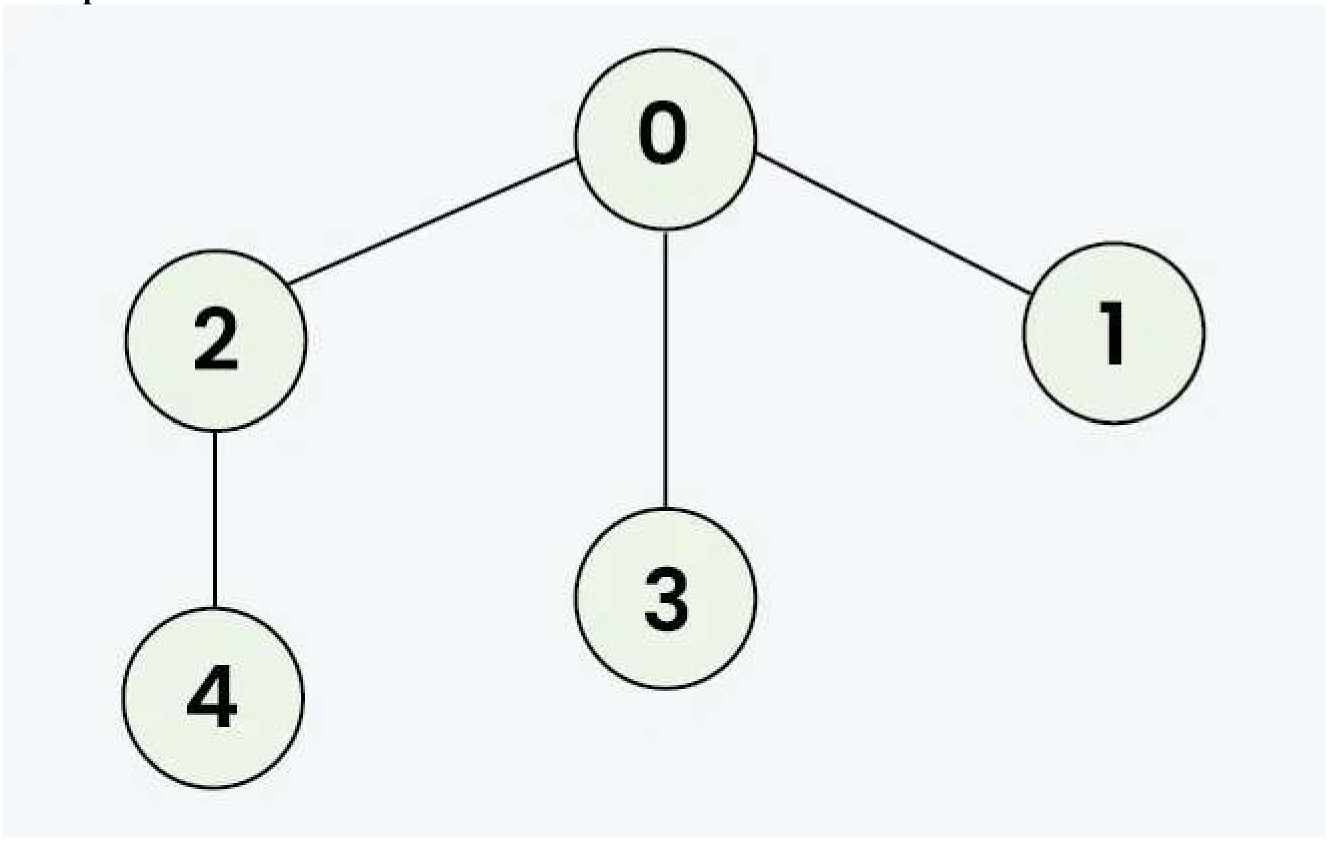
```
Example 1: true
Example 2: false
```

5.BFS of graph link

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.

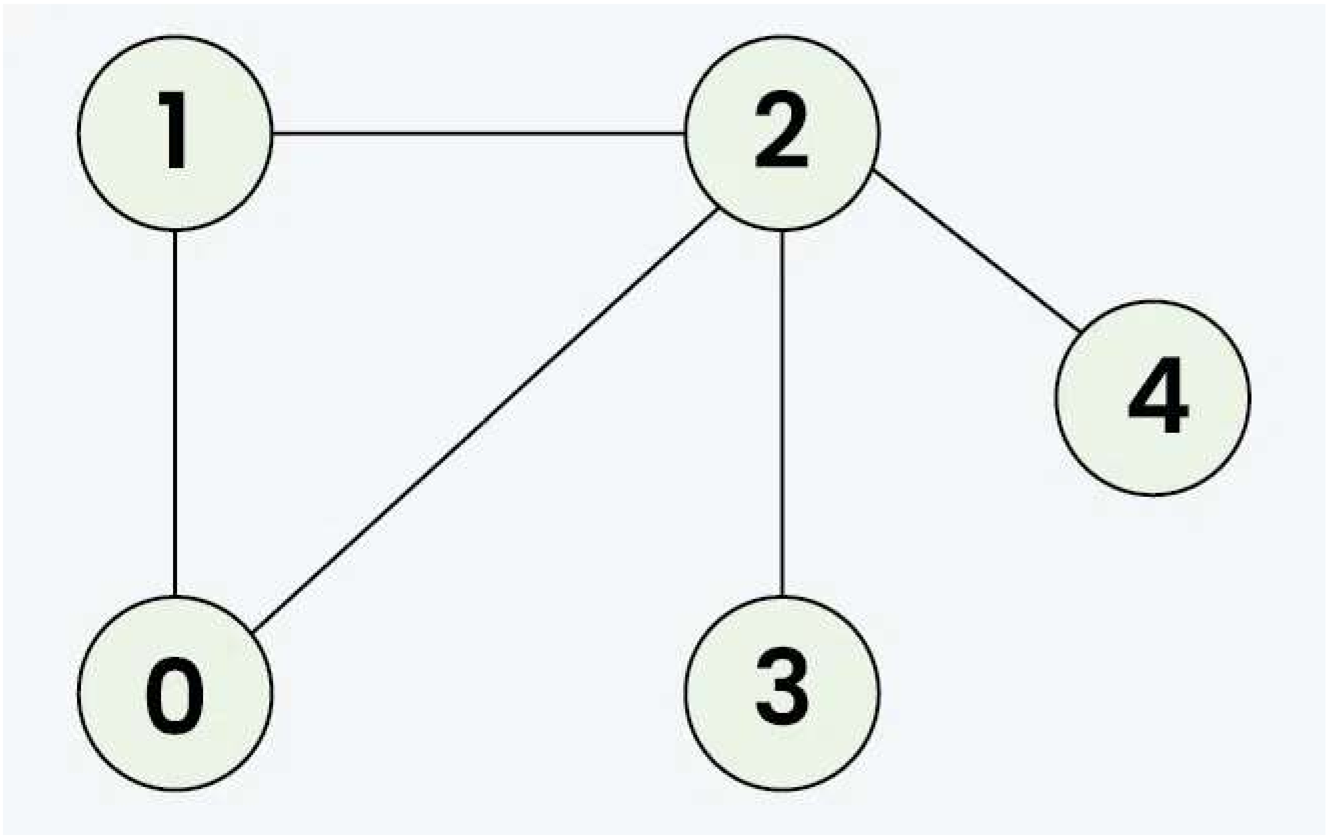
Note: Do traverse in the same order as they are in the adjacency list.

Example 1:



Input: adj = [[2,3,1], [0], [0,4], [0], [2]]
Output: [0, 2, 3, 1, 4]
Explanation: Starting from 0, the BFS traversal will follow these steps:
Visit 0 → Output: 0
Visit 2 (first neighbor of 0) → Output: 0, 2
Visit 3 (next neighbor of 0) → Output: 0, 2, 3
Visit 1 (next neighbor of 0) → Output: 0, 2, 3,
Visit 4 (neighbor of 2) → Final Output: 0, 2, 3, 1, 4

Example 2



Input: adj = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]
Output: [0, 1, 2, 3, 4]
Explanation: Starting from 0, the BFS traversal proceeds as follows:
Visit 0 → Output: 0
Visit 1 (the first neighbor of 0) → Output: 0, 1
Visit 2 (the next neighbor of 0) → Output: 0, 1, 2
Visit 3 (the first neighbor of 2 that hasn't been visited yet) → Output: 0, 1, 2, 3
Visit 4 (the next neighbor of 2) → Final Output: 0, 1, 2, 3, 4
Input: adj = [[1], [0, 2, 3], [1], [1, 4], [3]]
Output: [0, 1, 2, 3, 4]
Explanation: Starting the BFS from vertex 0:
Visit vertex 0 → Output: [0]
Visit vertex 1 (first neighbor of 0) → Output: [0, 1]
Visit vertex 2 (first unvisited neighbor of 1) → Output: [0, 1, 2]
Visit vertex 3 (next neighbor of 1) → Output: [0, 1, 2, 3]
Visit vertex 4 (neighbor of 3) → Final Output: [0, 1, 2, 3, 4]
Constraints:

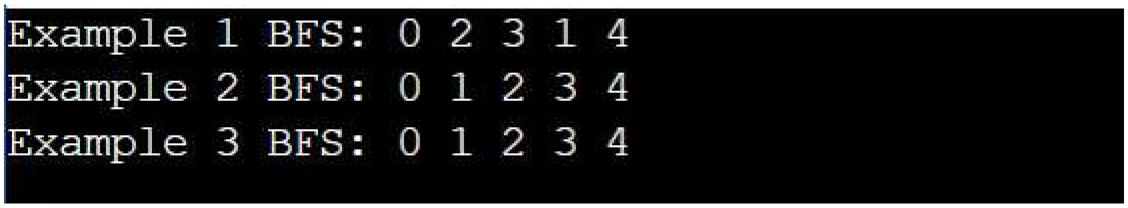
- 1 ≤ adj.size() ≤ 1e4
- 1 ≤ adj[i][j] ≤ 1e4

CODE:

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

using namespace std;
```

```
vector<int> bfsTraversal(int n, vector<vector<int>>& adj) {
    vector<int> result;         // To store BFS traversal order
    vector<bool> visited(n, false); // To track visited nodes
    queue<int> q;               // Queue for BFS

    // Start BFS from vertex 0
    q.push(0);
    visited[0] = true;

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

        // Visit all neighbors of the current node
        for (int neighbor : adj[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }

    return result;
}

int main() {
    // Example 1
    vector<vector<int>> adj1 = {{2, 3, 1}, {0}, {0, 4}, {0}, {2}};
    vector<int> result1 = bfsTraversal(adj1.size(), adj1);
    cout << "Example 1 BFS: ";
    for (int node : result1) {
        cout << node << " ";
    }
    cout << endl;

    // Example 2
    vector<vector<int>> adj2 = {{1, 2}, {0, 2}, {0, 1, 3, 4}, {2}, {2}};
    vector<int> result2 = bfsTraversal(adj2.size(), adj2);
    cout << "Example 2 BFS: ";
    for (int node : result2) {
        cout << node << " ";
    }
    cout << endl;

    // Example 3
    vector<vector<int>> adj3 = {{1}, {0, 2, 3}, {1}, {1, 4}, {3}};
    vector<int> result3 = bfsTraversal(adj3.size(), adj3);
    cout << "Example 3 BFS: ";
    for (int node : result3) {
        cout << node << " ";
    }
    cout << endl;

    return 0;
}
```

OUTPUT:



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