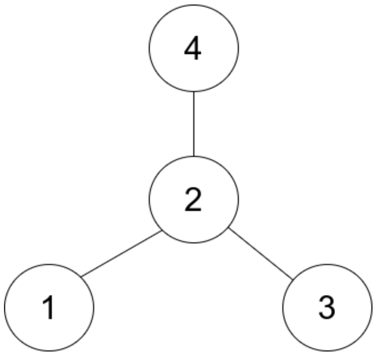
VERY EASY

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:

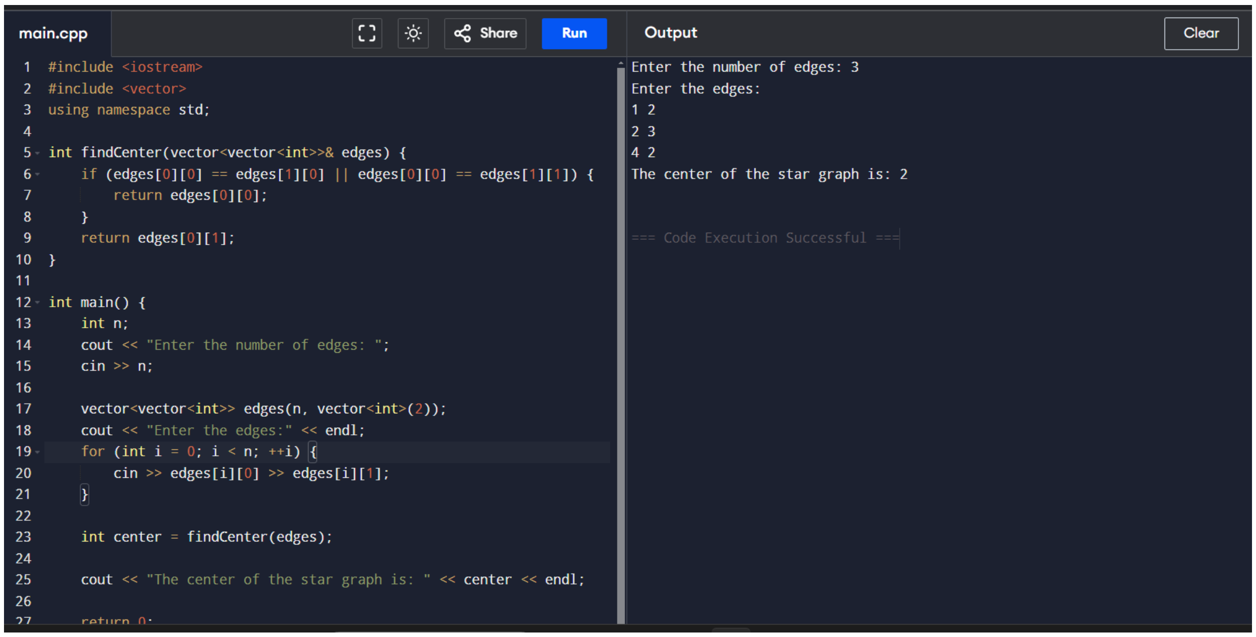


Input: edges = [[1,2],[2,3],[4,2]]
Output: 2
Explanation: As shown in the figure above, node 2 is connected to every other node, so 2 is the center.

Code:

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

int findCenter(vector<vector<int>>& edges) {
   if (edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1]) {
      return edges[0][0];
   }
   return edges[0][1];
}

int main() {

   int n;
   cout << "Enter the number of edges: ";
   cin >> n;

   vector<vector<int>> edges(n, vector<int>(2));
   cout << "Enter the edges:" << endl;
   for (int i = 0; i < n; ++i) {
      cin >> edges[i][0] >> edges[i][1];
   }

   int center = findCenter(edges);

   cout << "The center of the star graph is: " << center << endl;

   return 0;
}
```

Output:



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

int findCenter(vector<vector<int>>& edges) {
    if (edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1]) {
        return edges[0][0];
    }
    return edges[0][1];
}

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

    vector<vector<int>> edges(n, vector<int>(2));
    cout << "Enter the edges:" << endl;
    for (int i = 0; i < n; ++i) {
        cin >> edges[i][0] >> edges[i][1];
    }

    int center = findCenter(edges);

    cout << "The center of the star graph is: " << center << endl;

    return 0;
```

```
Enter the number of edges: 3
Enter the edges:
1 2
2 3
4 2
The center of the star graph is: 2

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

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.

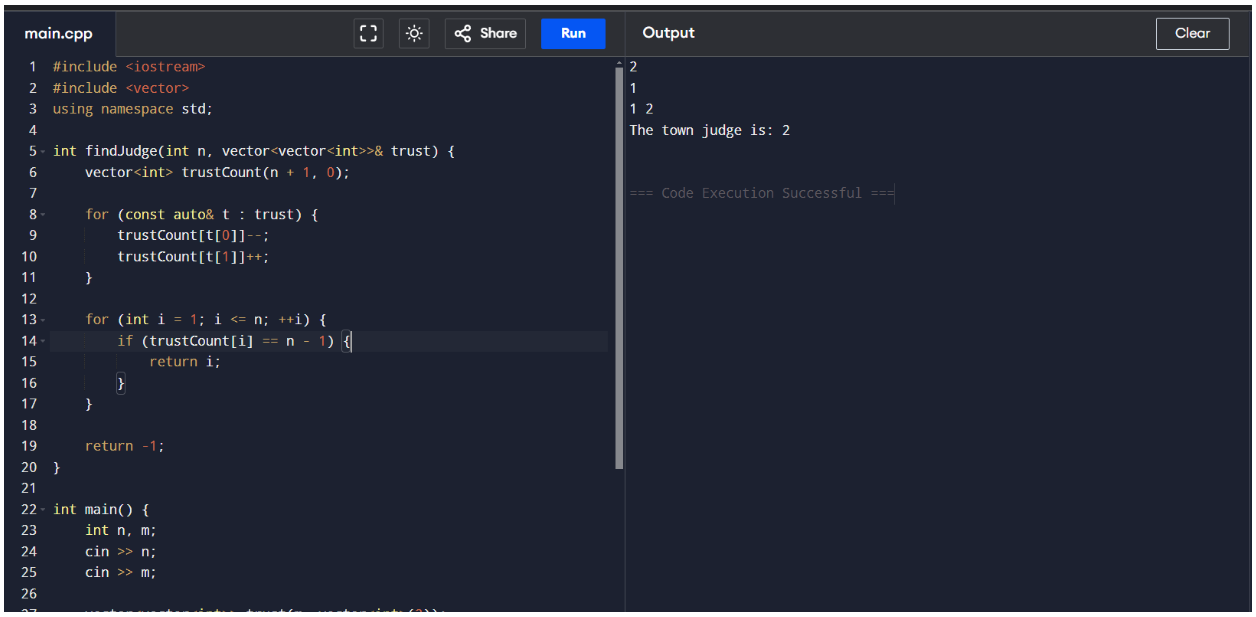
You are given an array trust where trust[i] = [ai, bi] representing that the person labeled ai trusts the person labeled bi. If a trust relationship does not exist in trust array, then such a trust relationship does not exist.

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

Code:

```
#include <iostream>

#include <vector>

using namespace std;


int findJudge(int n, vector<vector<int>>& trust) {

    vector<int> trustCount(n + 1, 0);


    for (const auto& t : trust) {

        trustCount[t[0]]--;

        trustCount[t[1]]++;

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

    return -1;
}

int main() {
    int n, m;
    cin >> n;
    cin >> m;

    vector<vector<int>> trust(m, vector<int>(2));
    for (int i = 0; i < m; ++i) {
        cin >> trust[i][0] >> trust[i][1];
    }
    int judge = findJudge(n, trust);
    if (judge == -1) {
        cout << "No town judge found." << endl;
    } else {
        cout << "The town judge is: " << judge << endl;
    }
    return 0;
}
```

OUTPUT:



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

int findJudge(int n, vector<vector<int>>& trust) {
    vector<int> trustCount(n + 1, 0);

    for (const auto& t : trust) {
        trustCount[t[0]]--;
        trustCount[t[1]]++;
    }

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

    return -1;
}

int main() {
    int n, m;
    cin >> n;
    cin >> m;

```

```
2
1
1 2
The town judge is: 2


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

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.

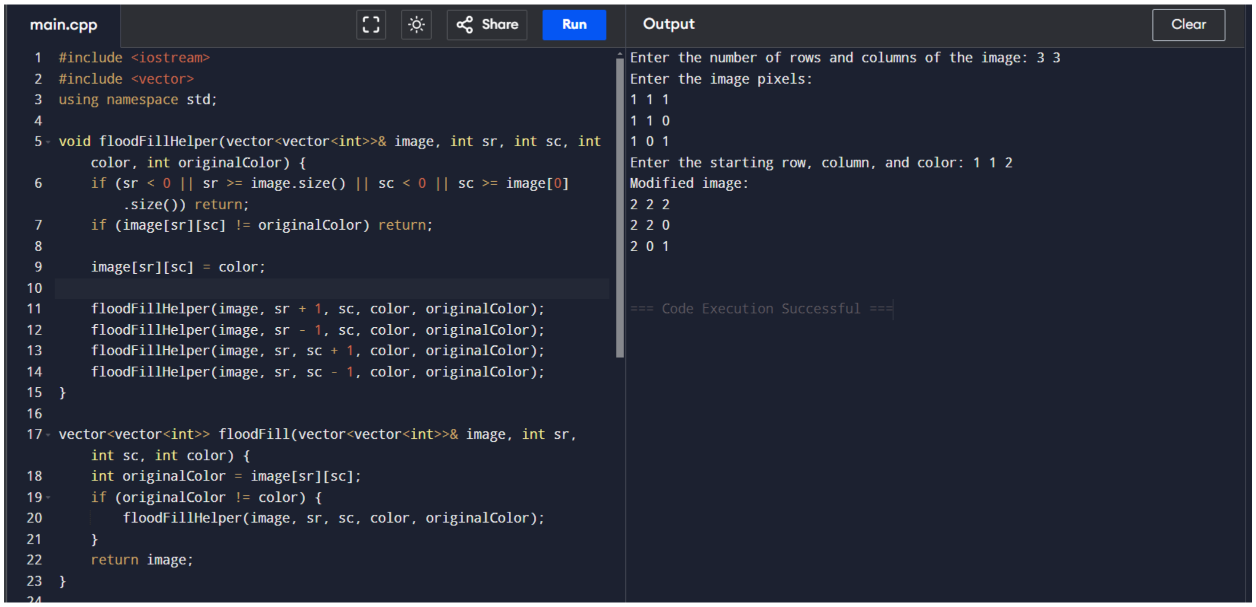
Example 1:

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]]

Code:

```
#include <iostream>
#include <vector>
using namespace std;
void floodFillHelper(vector<vector<int>>& image, int sr, int sc, int color, int originalColor) {
    if (sr < 0 || sr >= image.size() || sc < 0 || sc >= image[0].size()) return;
    if (image[sr][sc] != originalColor) return;
    image[sr][sc] = color;
    floodFillHelper(image, sr + 1, sc, color, originalColor);
    floodFillHelper(image, sr - 1, sc, color, originalColor);
    floodFillHelper(image, sr, sc + 1, color, originalColor);
    floodFillHelper(image, sr, sc - 1, color, originalColor);
}
vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int color) {
    int originalColor = image[sr][sc];
    if (originalColor != color) {
        floodFillHelper(image, sr, sc, color, originalColor);
    }   return image;
}
int main() {
    int m, n, sr, sc, color;
    cout << "Enter the number of rows and columns of the image: ";
    cin >> m >> n;
    vector<vector<int>> image(m, vector<int>(n));
    cout << "Enter the image pixels:" << endl;
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> image[i][j];
        }  }
    cout << "Enter the starting row, column, and color: ";
    cin >> sr >> sc >> color;
    vector<vector<int>> result = floodFill(image, sr, sc, color);
    cout << "Modified image:" << endl;
    for (const auto& row : result) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
    return 0;
}
```

OUTPUT:



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

void floodFillHelper(vector<vector<int>>& image, int sr, int sc, int
    color, int originalColor) {
    if (sr < 0 || sr >= image.size() || sc < 0 || sc >= image[0]
        .size()) return;
    if (image[sr][sc] != originalColor) return;

    image[sr][sc] = color;

    floodFillHelper(image, sr + 1, sc, color, originalColor);
    floodFillHelper(image, sr - 1, sc, color, originalColor);
    floodFillHelper(image, sr, sc + 1, color, originalColor);
    floodFillHelper(image, sr, sc - 1, color, originalColor);
}

vector<vector<int>> floodFill(vector<vector<int>>& image, int sr,
    int sc, int color) {
    int originalColor = image[sr][sc];
    if (originalColor != color) {
        floodFillHelper(image, sr, sc, color, originalColor);
    }
    return image;
}
```

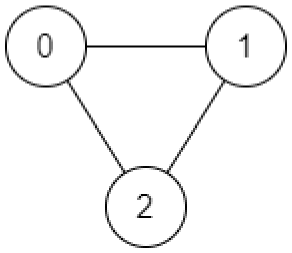
```
Enter the number of rows and columns of the image: 3 3
Enter the image pixels:
1 1 1
1 1 0
1 0 1
Enter the starting row, column, and color: 1 1 2
Modified image:
2 2 2
2 2 0
2 0 1


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

Easy:

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:



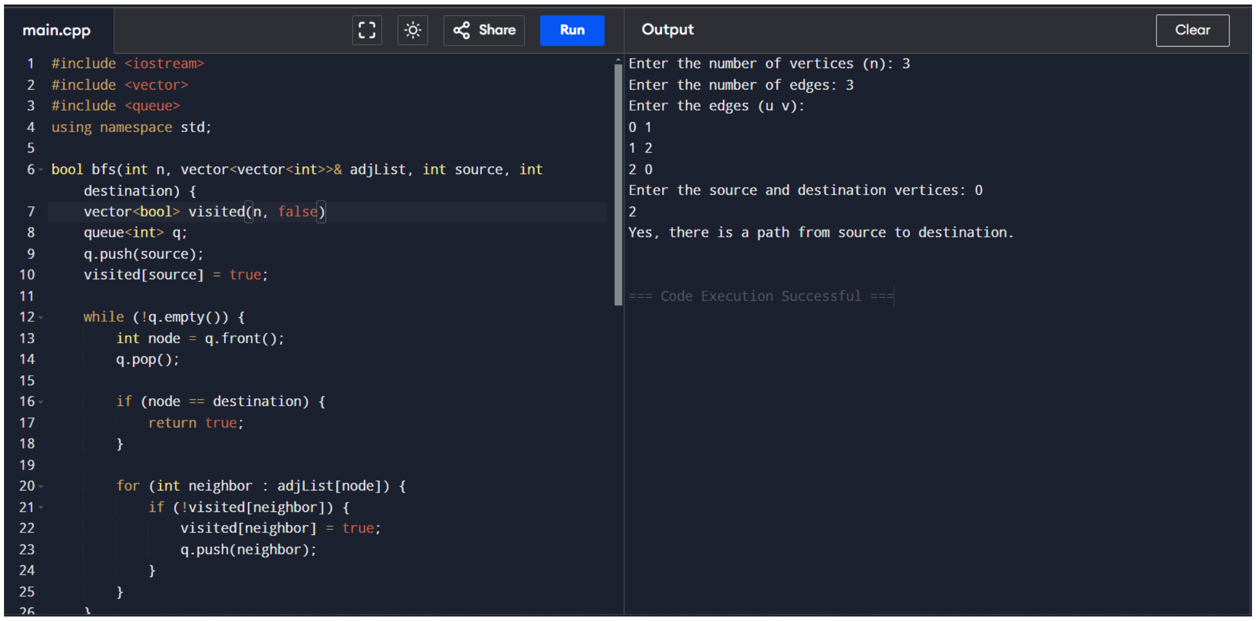
Input: n = 3, edges = [[0,1],[1,2],[2,0]], source = 0, destination = 2
Output: true

CODE:

```
#include <iostream>

#include <vector>

#include <queue>

using namespace std;


bool bfs(int n, vector<vector<int>>& adjList, int source, int destination) {

    vector<bool> visited(n, false);

    queue<int> q;

    q.push(source);

    visited[source] = true;


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

        if (node == destination) {
            return true;
        }

        for (int neighbor : adjList[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
    return false;
}

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

    for (auto& edge : edges) {
        adjList[edge[0]].push_back(edge[1]);
        adjList[edge[1]].push_back(edge[0]);
    }

    return bfs(n, adjList, source, destination);
}
```

```
int main() {
    int n, source, destination, e;
    cout << "Enter the number of vertices (n): ";
    cin >> n;

    cout << "Enter the number of edges: ";
    cin >> e;

    vector<vector<int>> edges(e, vector<int>(2));
    cout << "Enter the edges (u v): " << endl;
    for (int i = 0; i < e; i++) {
        cin >> edges[i][0] >> edges[i][1];
    }

    cout << "Enter the source and destination vertices: ";
    cin >> source >> destination;

    if (validPath(n, edges, source, destination)) {
        cout << "Yes, there is a path from source to destination." << endl;
    } else {
        cout << "No, there is no path from source to destination." << endl;
    }
    return 0;
}
```

OUTPUT:



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

bool bfs(int n, vector<vector<int>>& adjList, int source, int
    destination) {
    vector<bool> visited(n, false);
    queue<int> q;
    q.push(source);
    visited[source] = true;

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

        if (node == destination) {
            return true;
        }

        for (int neighbor : adjList[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
```

```
Enter the number of vertices (n): 3
Enter the number of edges: 3
Enter the edges (u v):
0 1
1 2
2 0
Enter the source and destination vertices: 0
2
Yes, there is a path from source to destination.


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

DFS of Graph

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 Depth First Traversal (DFS) starting from vertex 0, visiting vertices from left to right as per the adjacency list, and return a list containing the DFS traversal of the graph.

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

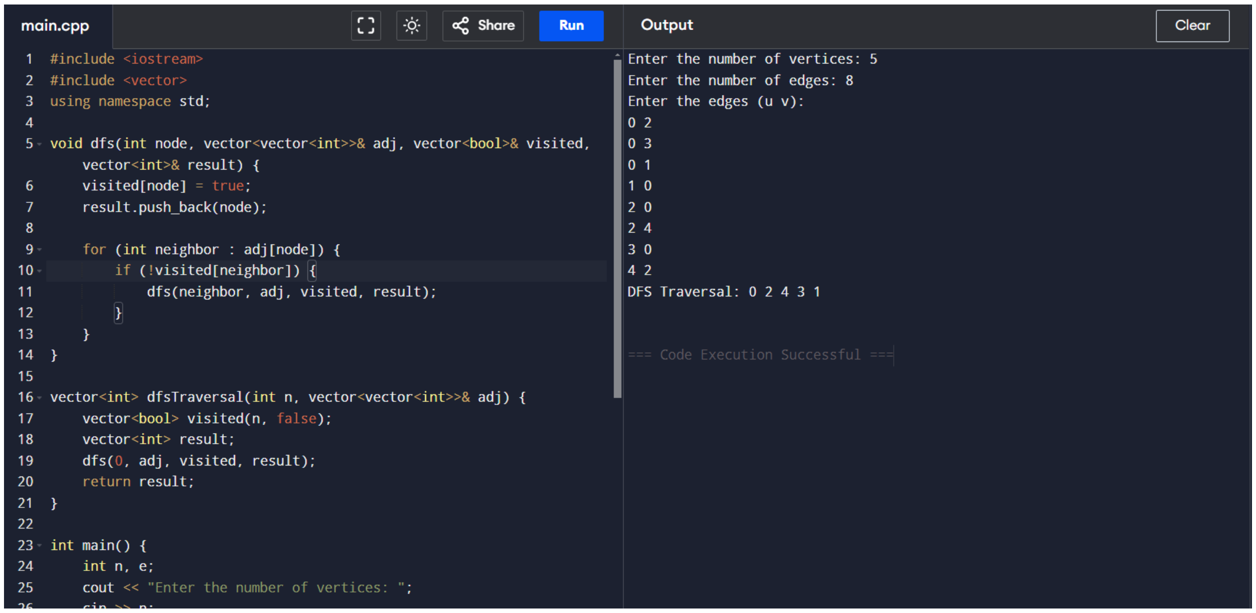
Example 1:

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

CODE:

```cpp
#include <iostream>

#include <vector>

using namespace std;


void dfs(int node, vector<vector<int>>& adj, vector<bool>& visited, vector<int>& result) {
    visited[node] = true;
    result.push_back(node);

    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor, adj, visited, result);
        }
    }
}

vector<int> dfsTraversal(int n, vector<vector<int>>& adj) {
    vector<bool> visited(n, false);
    vector<int> result;
    dfs(0, adj, visited, result);
    return result;
}

int main() {
    int n, e;
    cout << "Enter the number of vertices: ";
    cin >> n;
    cout << "Enter the number of edges: ";
    cin >> e;

    vector<vector<int>> adj(n);

    cout << "Enter the edges (u v):" << endl;
    for (int i = 0; i < e; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<int> result = dfsTraversal(n, adj);
    cout << "DFS Traversal: ";
    for (int node : result) {
        cout << node << " ";
    }
    cout << endl;
    return 0;
}
```

OUTPUT:



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

void dfs(int node, vector<vector<int>>& adj, vector<bool>& visited,
    vector<int>& result) {
    visited[node] = true;
    result.push_back(node);

    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor, adj, visited, result);
        }
    }
}

vector<int> dfsTraversal(int n, vector<vector<int>>& adj) {
    vector<bool> visited(n, false);
    vector<int> result;
    dfs(0, adj, visited, result);
    return result;
}

int main() {
    int n, e;
    cout << "Enter the number of vertices: ";
```

```
Enter the number of vertices: 5
Enter the number of edges: 8
Enter the edges (u v):
0 2
0 3
0 1
1 0
2 0
2 4
3 0
4 2
DFS Traversal: 0 2 4 3 1


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

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.

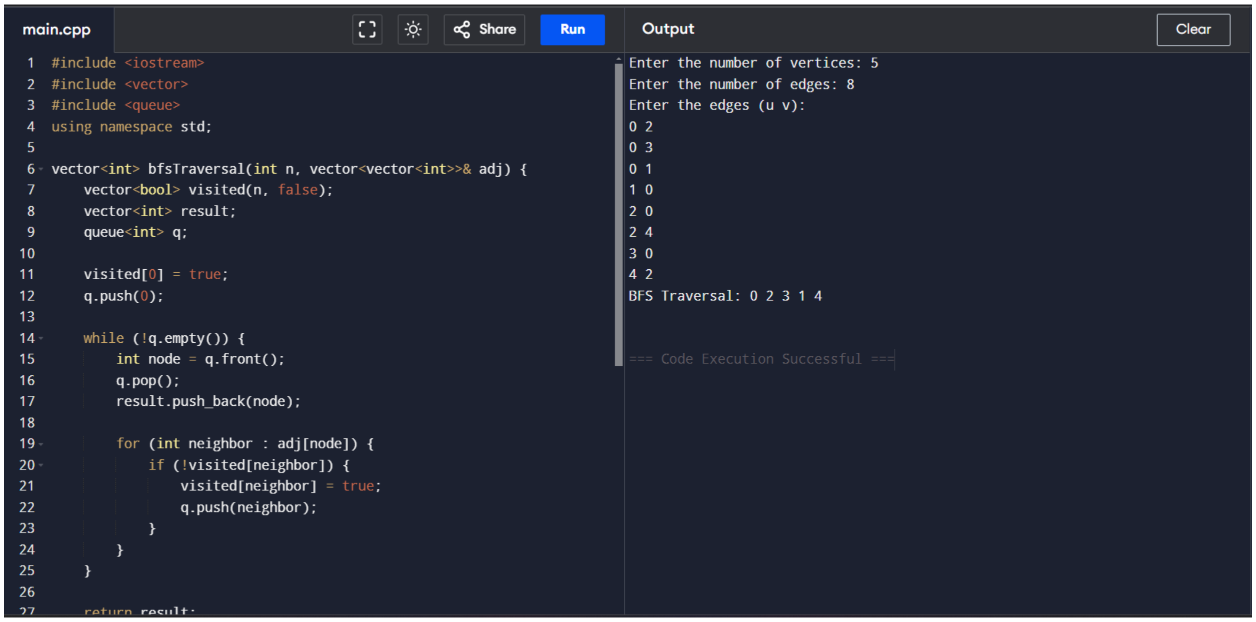
Note: Do traverse in the same order as they are in the adjacency list.

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

CODE:

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

vector<int> bfsTraversal(int n, vector<vector<int>>& adj) {
    vector<bool> visited(n, false);
    vector<int> result;
    queue<int> q;

    visited[0] = true;
    q.push(0);

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

        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }

    return result;
}

int main() {
    int n, e;
    cout << "Enter the number of vertices: ";
    cin >> n;
    cout << "Enter the number of edges: ";
    cin >> e;

    vector<vector<int>> adj(n);

    cout << "Enter the edges (u v):" << endl;
    for (int i = 0; i < e; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<int> result = bfsTraversal(n, adj);
    cout << "BFS Traversal: ";
    for (int node : result) {
        cout << node << " ";
    }
    cout << endl;

    return 0;
}
```

OUTPUT:



main.cpp

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

vector<int> bfsTraversal(int n, vector<vector<int>>& adj) {
    vector<bool> visited(n, false);
    vector<int> result;
    queue<int> q;

    visited[0] = true;
    q.push(0);

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

        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }

    return result;
```

Output

```
Enter the number of vertices: 5
Enter the number of edges: 8
Enter the edges (u v):
0 2
0 3
0 1
1 0
2 0
2 4
3 0
4 2
BFS Traversal: 0 2 3 1 4


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

Medium

01 Matrix

Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell.

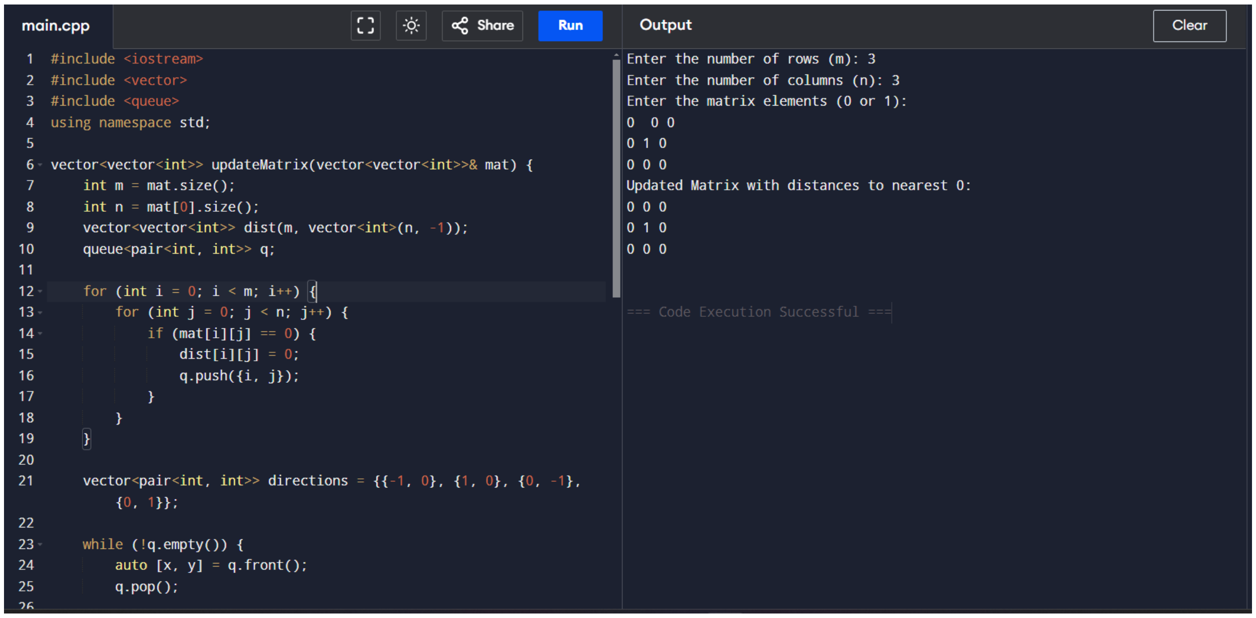
The distance between two adjacent cells is 1.

Example 1:
Input: mat = [[0,0,0],[0,1,0],[0,0,0]]
Output: [[0,0,0],[0,1,0],[0,0,0]]

CODE:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
    int m = mat.size();
    int n = mat[0].size();
    vector<vector<int>> dist(m, vector<int>(n, -1));
    queue<pair<int, int>> q;

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (mat[i][j] == 0) {
                dist[i][j] = 0;
                q.push({i, j});
            }  }  }
    vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();

        for (auto& dir : directions) {
            int newX = x + dir.first;
            int newY = y + dir.second;

            if (newX >= 0 && newX < m && newY >= 0 && newY < n && dist[newX][newY] == -1) {
                dist[newX][newY] = dist[x][y] + 1;
                q.push({newX, newY});
            }} }
    return dist;}
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<int>> mat(m, vector<int>(n));
    cout << "Enter the matrix elements (0 or 1):\n";
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cin >> mat[i][j];
        }}

    vector<vector<int>> result = updateMatrix(mat);

    cout << "Updated Matrix with distances to nearest 0:\n";
    for (auto& row : result) {
        for (int cell : row) {
            cout << cell << " ";
        }
        cout << endl;
    }

    return 0;
}
```

OUTPUT:



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

vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
    int m = mat.size();
    int n = mat[0].size();
    vector<vector<int>> dist(m, vector<int>(n, -1));
    queue<pair<int, int>> q;

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (mat[i][j] == 0) {
                dist[i][j] = 0;
                q.push({i, j});
            }
        }
    }

    vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1},
        {0, 1}};

    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();
```

```
Enter the number of rows (m): 3
Enter the number of columns (n): 3
Enter the matrix elements (0 or 1):
0  0 0
0 1 0
0 0 0
Updated Matrix with distances to nearest 0:
0 0 0
0 1 0
0 0 0


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

Course Schedule II

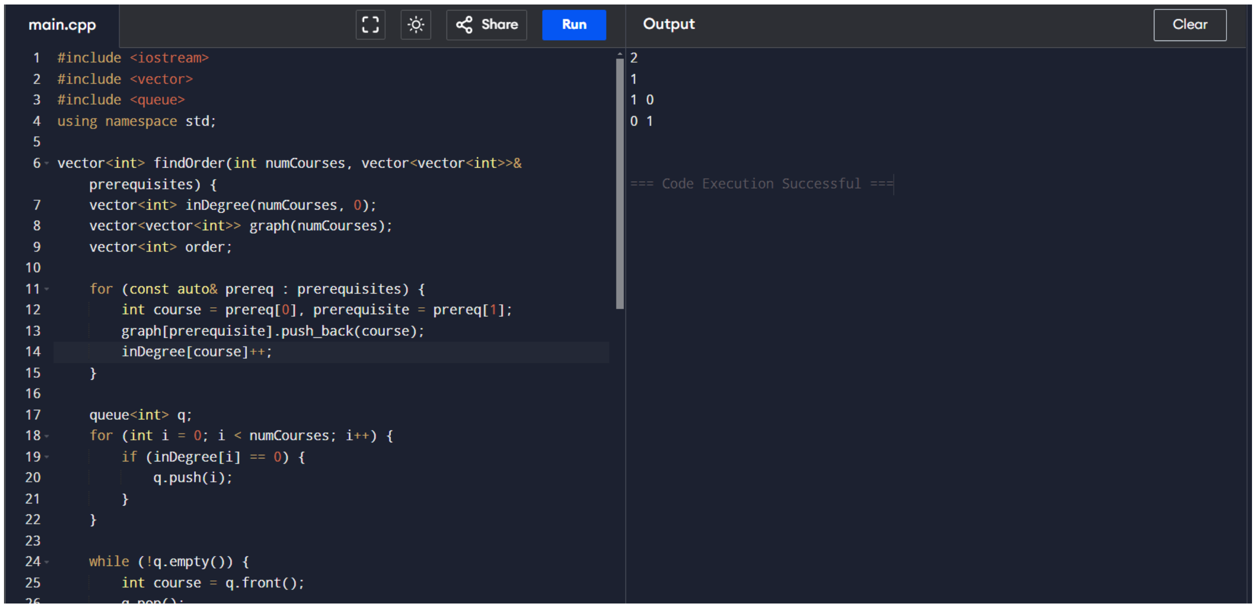
There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.
Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.
Example 1:
Input: numCourses = 2, prerequisites = [[1,0]]
Output: [0,1]

CODE:

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

vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
    vector<int> inDegree(numCourses, 0);
    vector<vector<int>> graph(numCourses);
    vector<int> order;

    for (const auto& prereq : prerequisites) {
        int course = prereq[0], prerequisite = prereq[1];
        graph[prerequisite].push_back(course);
        inDegree[course]++;
    }

    queue<int> q;
    for (int i = 0; i < numCourses; i++) {
        if (inDegree[i] == 0) {
            q.push(i);
        }
    }

    while (!q.empty()) {
        int course = q.front();
        q.pop();
        order.push_back(course);
        for (int neighbor : graph[course]) {
            inDegree[neighbor]--;
            if (inDegree[neighbor] == 0) {
                q.push(neighbor);
            }
        }
    }

    return order.size() == numCourses ? order : vector<int>();
}

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

    int numPrerequisites;
    cin >> numPrerequisites;
    vector<vector<int>> prerequisites(numPrerequisites, vector<int>(2));
    for (int i = 0; i < numPrerequisites; i++) {
        cin >> prerequisites[i][0] >> prerequisites[i][1];
    }
    vector<int> result = findOrder(numCourses, prerequisites)
    if (result.empty()) {
        cout << "It is impossible to finish all courses.\n";
    } else {
        for (int course : result) {
            cout << course << " ".   }
        cout << endl;
    }
    return 0;
}
```

OUTPUT:



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

vector<int> findOrder(int numCourses, vector<vector<int>>&
    prerequisites) {
    vector<int> inDegree(numCourses, 0);
    vector<vector<int>> graph(numCourses);
    vector<int> order;

    for (const auto& prereq : prerequisites) {
        int course = prereq[0], prerequisite = prereq[1];
        graph[prerequisite].push_back(course);
        inDegree[course]++;
    }

    queue<int> q;
    for (int i = 0; i < numCourses; i++) {
        if (inDegree[i] == 0) {
            q.push(i);
        }
    }

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

```
2
1
1 0
0 1

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

Minimum Height Trees

A tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.

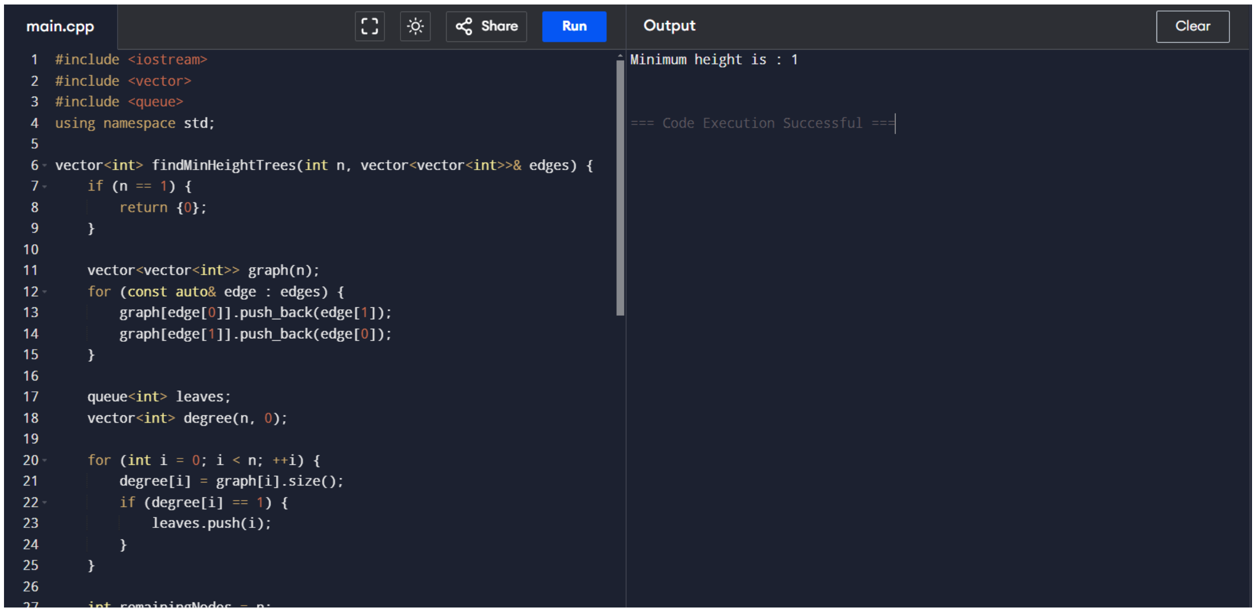
Given a tree of n nodes labelled from 0 to n - 1, and an array of n - 1 edges where edges[i] = [ai, bi] indicates that there is an undirected edge between the two nodes ai and bi in the tree, you can choose any node of the tree as the root. When you select a node x as the root, the result tree has height h. Among all possible rooted trees, those with minimum height (i.e. min(h)) are called minimum height trees (MHTs).

Return a list of all MHTs' root labels. You can return the answer in any order.
The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

CODE:

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

vector<int> findMinHeightTrees(int n, vector<vector<int>>& edges) {
    if (n == 1) {
        return {0};
    }

    vector<vector<int>> graph(n);
    for (const auto& edge : edges) {
        graph[edge[0]].push_back(edge[1]);
        graph[edge[1]].push_back(edge[0]);
    }

    queue<int> leaves;
    vector<int> degree(n, 0);

    for (int i = 0; i < n; ++i) {
        degree[i] = graph[i].size();
        if (degree[i] == 1) {
            leaves.push(i);
        }
    }

    int remainingNodes = n;
    while (remainingNodes > 2) {
            int leavesSize = leaves.size();
            remainingNodes -= leavesSize;

            for (int i = 0; i < leavesSize; ++i) {
                int leaf = leaves.front();
                leaves.pop();

                for (int neighbor : graph[leaf]) {
                    if (--degree[neighbor] == 1) {
                        leaves.push(neighbor);
                    }}}
        vector<int> result;
        while (!leaves.empty()) {
            result.push_back(leaves.front());
            leaves.pop();
        }
        return result;
    }
    int main() {
        int n = 4;
        vector<vector<int>> edges = {{1,0}, {1,2}, {1,3}};
        vector<int> result = findMinHeightTrees(n, edges);
        for (int node : result) {
            cout << node << " ";
        }
        cout << endl;
        return 0;
    }
```

OUTPUT:



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

vector<int> findMinHeightTrees(int n, vector<vector<int>>& edges) {
    if (n == 1) {
        return {0};
    }

    vector<vector<int>> graph(n);
    for (const auto& edge : edges) {
        graph[edge[0]].push_back(edge[1]);
        graph[edge[1]].push_back(edge[0]);
    }

    queue<int> leaves;
    vector<int> degree(n, 0);

    for (int i = 0; i < n; ++i) {
        degree[i] = graph[i].size();
        if (degree[i] == 1) {
            leaves.push(i);
        }
    }

    int remainingNodes = n;
```

```
Minimum height is : 1


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

HARD

Accounts Merge

Given a list of accounts where each element accounts[i] is a list of strings, where the first element accounts[i][0] is a name, and the rest of the elements are emails representing emails of the account.

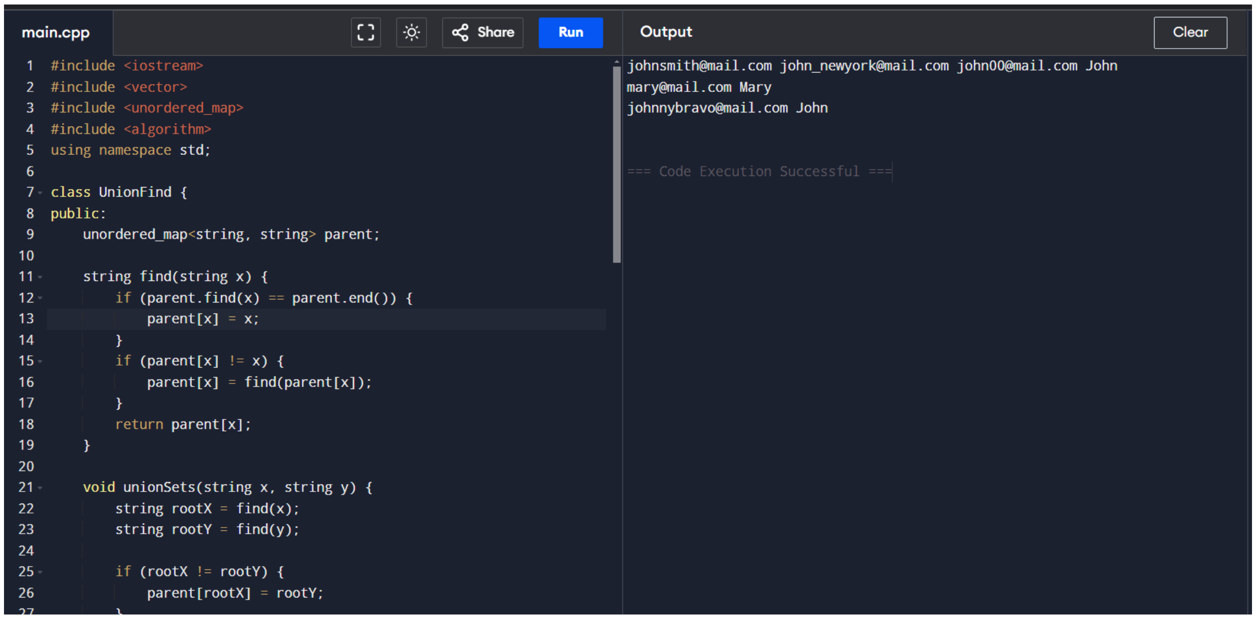
Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name. After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order. The accounts themselves can be returned in any order.
Example 1:

Input: accounts =
[["John","johnsmith@mail.com","john_newyork@mail.com"],["John","johnsmith@mail.com","john00@mail.com"],["Mary","mary@mail.com"],["John","johnnybravo@mail.com"]]
Output:
[["John","john00@mail.com","john_newyork@mail.com","johnsmith@mail.com"],["Mary","mary@mail.com"],["John","johnnybravo@mail.com"]]

CODE:

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

class UnionFind {
public:
    unordered_map<string, string> parent;

    string find(string x) {
        if (parent.find(x) == parent.end()) {
            parent[x] = x;
        }
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void unionSets(string x, string y) {
        string rootX = find(x);
        string rootY = find(y);

        if (rootX != rootY) {
            parent[rootX] = rootY;
        }
    }
};

vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {
    UnionFind uf;
    unordered_map<string, string> emailToName;

    for (const auto& account : accounts) {
        string name = account[0];
        for (int i = 1; i < account.size(); ++i) {
            string email = account[i];
            emailToName[email] = name;

            if (i == 1) {
                continue;
            }
            uf.unionSets(account[1], email);
        }
    }

    unordered_map<string, vector<string>> mergedAccounts;
    for (const auto& pair : emailToName) {
        string rootEmail = uf.find(pair.first);
        mergedAccounts[rootEmail].push_back(pair.first);
    }

    vector<vector<string>> result;
    for (auto& pair : mergedAccounts) {
        string name = emailToName[pair.second[0]];
        pair.second.push_back(name);
        sort(pair.second.begin(), pair.second.end());
        reverse(pair.second.begin(), pair.second.end());
        result.push_back(pair.second);}
    return result;
}
int main() {
    vector<vector<string>> accounts = {
        {"John", "johnsmith@mail.com", "john_newyork@mail.com"},
        {"John", "johnsmith@mail.com", "john00@mail.com"},
        {"Mary", "mary@mail.com"},
        {"John", "johnnybravo@mail.com"}
    };
    vector<vector<string>> result = accountsMerge(accounts);
    for (const auto& account : result) {
        for (const auto& email : account) {
            cout << email << " ";
        }
        cout << endl;}
    return 0;
}
```

OUTPUT:


main.cpp Share Run Output Clear

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

class UnionFind {
public:
    unordered_map<string, string> parent;

    string find(string x) {
        if (parent.find(x) == parent.end()) {
            parent[x] = x;
        }
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void unionSets(string x, string y) {
        string rootX = find(x);
        string rootY = find(y);

        if (rootX != rootY) {
            parent[rootX] = rootY;
```

```
johnsmith@mail.com john_newyork@mail.com john00@mail.com John
mary@mail.com Mary
johnnybravo@mail.com John


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

Rotting Oranges

You are given an m x n grid where each cell can have one of three values:

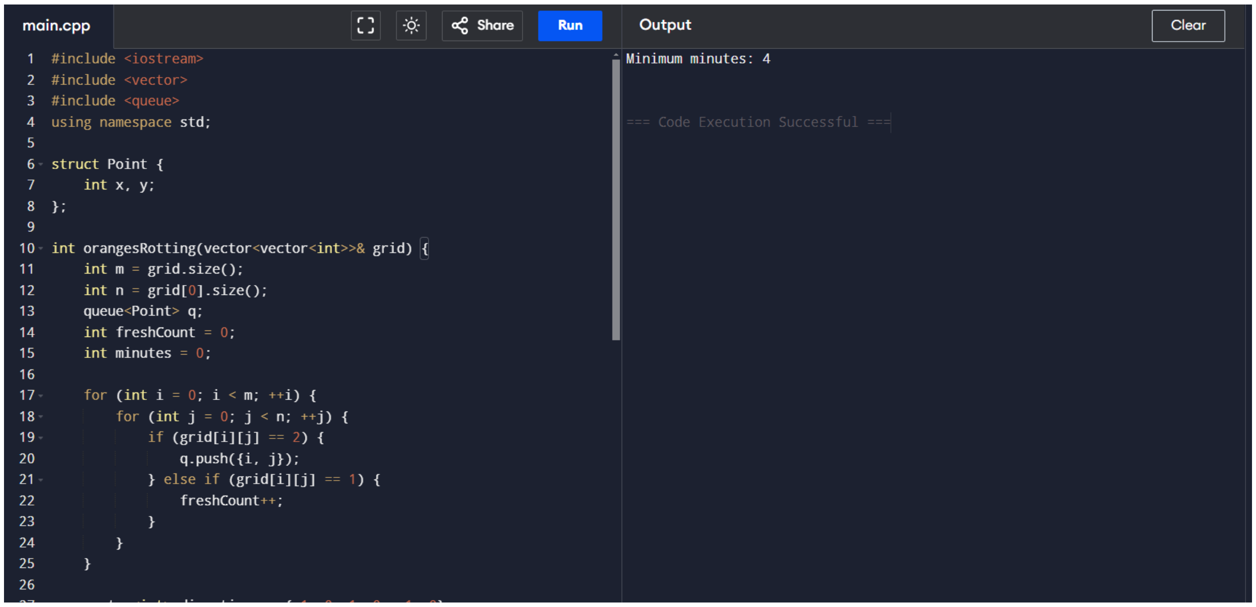
0 representing an empty cell,
1 representing a fresh orange, or
2 representing a rotten orange.
Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1.
Example
Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4

CODE:

```cpp
#include <iostream>

#include <vector>

#include <queue>

using namespace std;
```

```
struct Point {
    int x, y;
};
int orangesRotting(vector<vector<int>>& grid) {
    int m = grid.size();
    int n = grid[0].size();
    queue<Point> q;
    int freshCount = 0;
    int minutes = 0;
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (grid[i][j] == 2) {
                q.push({i, j});
            } else if (grid[i][j] == 1) {
                freshCount++;
            } }  }
    vector<int> directions = {-1, 0, 1, 0, -1, 0};
    while (!q.empty() && freshCount > 0) {
        int size = q.size();
        minutes++;

        for (int i = 0; i < size; ++i) {
            Point p = q.front();
            q.pop();

            for (int d = 0; d < 4; ++d) {
                int newX = p.x + directions[d];
                int newY = p.y + directions[d + 1];
                if (newX >= 0 && newX < m && newY >= 0 && newY < n && grid[newX][newY] == 1) {
                    grid[newX][newY] = 2;
                    freshCount--;
                    q.push({newX, newY});
                }
            }
        }
    }

    return freshCount == 0 ? minutes : -1;
}
int main() {
    vector<vector<int>> grid = {{2, 1, 1}, {1, 1, 0}, {0, 1, 1}};
    int result = orangesRotting(grid);
    cout << "Minimum minutes: " << result << endl;
    return 0;
}
```
OUTPUT:



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

struct Point {
    int x, y;
};

int orangesRotting(vector<vector<int>>& grid) {
    int m = grid.size();
    int n = grid[0].size();
    queue<Point> q;
    int freshCount = 0;
    int minutes = 0;

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (grid[i][j] == 2) {
                q.push({i, j});
            } else if (grid[i][j] == 1) {
                freshCount++;
            }
        }
    }

```

```
Minimum minutes: 4


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

Evaluate Division

You are given an array of variable pairs equations and an array of real numbers values, where equations[i] = [Ai, Bi] and values[i] represent the equation Ai / Bi = values[i]. Each Ai or Bi is a string that represents a single variable.

You are also given some queries, where queries[j] = [Cj, Dj] represents the jth query where you must find the answer for Cj / Dj = ?.

Return the answers to all queries. If a single answer cannot be determined, return -1.0.

Note: The input is always valid. You may assume that evaluating the queries will not result in division by zero and that there is no contradiction.

Note: The variables that do not occur in the list of equations are undefined, so the answer cannot be determined for them.
Example 1:

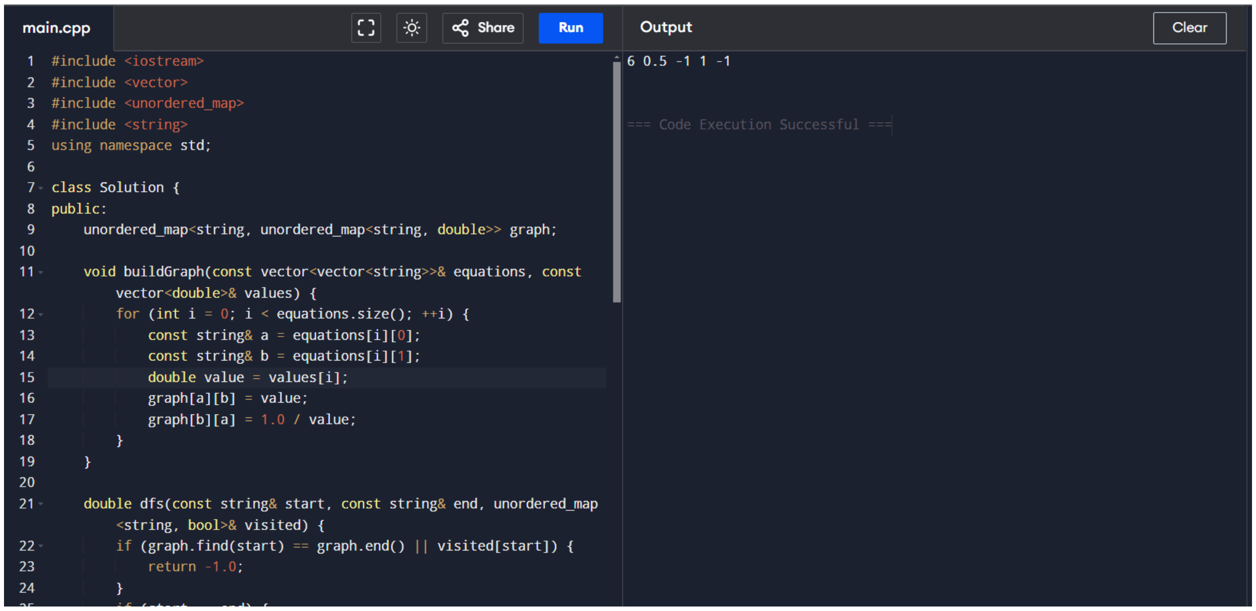
Input: equations = [["a","b"],["b","c"]], values = [2.0,3.0], queries = [["a","c"],["b","a"],["a","e"],["a","a"],["x","x"]]
Output: [6.00000,0.50000,-1.00000,1.00000,-1.00000]

CODE:

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

class Solution {
public:
    unordered_map<string, unordered_map<string, double>> graph;

    void buildGraph(const vector<vector<string>>& equations, const vector<double>& values) {
        for (int i = 0; i < equations.size(); ++i) {
            const string& a = equations[i][0];
            const string& b = equations[i][1];
            double value = values[i];
            graph[a][b] = value;
            graph[b][a] = 1.0 / value;
        }
    }

    double dfs(const string& start, const string& end, unordered_map<string, bool>& visited) {
        if (graph.find(start) == graph.end() || visited[start]) {
            return -1.0;
        }
        if (start == end) {
            return 1.0;
        }
        visited[start] = true;
        for (const auto& neighbor : graph[start]) {
            double result = dfs(neighbor.first, end, visited);
            if (result != -1.0) {
                return result * neighbor.second;
            }
        }
        return -1.0; }
    vector<double> calcEquation(const vector<vector<string>>& equations, const vector<double>& values, const vector<vector<string>>& queries) {
        buildGraph(equations, values);
        vector<double> result;
        for (const auto& query : queries) {
            unordered_map<string, bool> visited;
            result.push_back(dfs(query[0], query[1], visited));
        }
        return result ; } };
int main() {
    Solution solution;
    vector<vector<string>> equations = {{"a", "b"}, {"b", "c"}};
    vector<double> values = {2.0, 3.0};
    vector<vector<string>> queries = {{"a", "c"}, {"b", "a"}, {"a", "e"}, {"a", "a"}, {"x", "x"}};
    vector<double> result = solution.calcEquation(equations, values, queries);
    for (double res : result) {
        cout << res << " ";
    }
    cout << endl;
    return 0;
```

}

OUTPUT:



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

class Solution {
public:
    unordered_map<string, unordered_map<string, double>> graph;

    void buildGraph(const vector<vector<string>>& equations, const
        vector<double>& values) {
        for (int i = 0; i < equations.size(); ++i) {
            const string& a = equations[i][0];
            const string& b = equations[i][1];
            double value = values[i];
            graph[a][b] = value;
            graph[b][a] = 1.0 / value;
        }
    }

    double dfs(const string& start, const string& end, unordered_map
        <string, bool>& visited) {
        if (graph.find(start) == graph.end() || visited[start]) {
            return -1.0;
        }
```

```
6 0.5 -1 1 -1


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