

Day - 7

Name – Ishita

Faculty Name- Er. Rajni Devi

UID- 22BCS15353

Date- 28 Dec ,2024

Section – 620-B

Ques 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.

Code :-

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

int findJudge(int n, int trust[][2], int trustSize) {
    // Arrays to track the in-degree and out-degree for each person
    int in_degree[n + 1] = {0}; // Initialize all to 0
    int out_degree[n + 1] = {0}; // Initialize all to 0

    // Process the trust relationships
    for (int i = 0; i < trustSize; i++) {
        int a = trust[i][0]; // Person a trusts person b
        int b = trust[i][1]; // Person b is trusted by person a

        out_degree[a]++; // a trusts someone
        in_degree[b]++; // b is trusted by someone
    }

    // Find the person who has in-degree of n-1 and out-degree of 0
    for (int i = 1; i <= n; i++) {
```

```

        if (in_degree[i] == n - 1 && out_degree[i] == 0) {
            return i; // Person i is the judge
        }
    }

    return -1; // No judge found
}

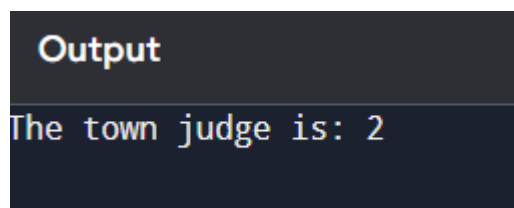
int main() {
    int n = 2;
    int trust[1][2] = {{1, 2}}; // Example trust relationship
    int trustSize = 1; // Number of trust relationships

    int judge = findJudge(n, trust, trustSize);
    cout << "The town judge is: " << judge << endl; // Output: 2

    return 0;
}

```

Output:



Ques 2:

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.

Code:

```

#include <iostream>

#include <unordered_map>

#include <vector>

using namespace std;

int findCenter(vector<vector<int>>& edges) {
    // Use a map to count the occurrences of each node
    unordered_map<int, int> count;
}

```

```

for (auto& edge : edges) {
    count[edge[0]]++;
    count[edge[1]]++;
}
for (auto& entry : count) {
    if (entry.second == edges.size()) {
        return entry.first; // The center node
    }
}

return -1; // In case there's no center (shouldn't happen in a star graph)
}

int main() {
    vector<vector<int>> edges1 = {{1, 2}, {2, 3}, {4, 2}};
    cout << "Center of star graph 1: " << findCenter(edges1) << endl; // Output: 2
    vector<vector<int>> edges2 = {{1, 2}, {5, 1}, {1, 3}, {1, 4}};
    cout << "Center of star graph 2: " << findCenter(edges2) << endl; // Output: 1
    return 0;
}

```

Output:

```

Output
Center of star graph 1: 2
Center of star graph 2: 1

```

Ques 3:

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.

Code:

```

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

vector<int> bfsOfGraph(int n, vector<vector<int>>& adj) {

    vector<int> result;

    vector<bool> visited(n, false);

    queue<int> q;

    q.push(0);

    visited[0] = true;

    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() {

    vector<vector<int>> adj = {{2, 3, 1}, {0}, {0, 4}, {0}, {2}};

    int n = adj.size();

```

```

vector<int> bfsResult = bfsOfGraph(n, adj);

for (int node : bfsResult) {

    cout << node << " ";

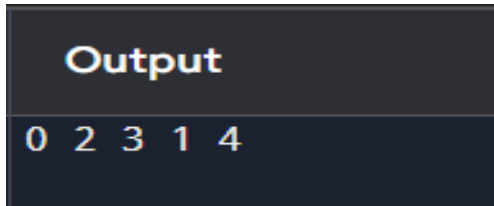
}

return 0;

}

```

Output:



```

Output
0 2 3 1 4

```

Ques 4:

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.

Code:

```

#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);

        }

    }

}

```

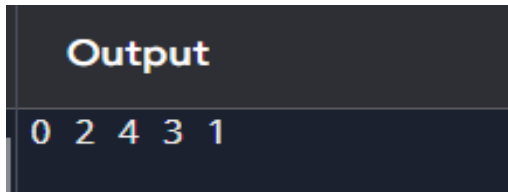
```

vector<int> dfsOfGraph(int n, vector<vector<int>>& adj) {
    vector<int> result;
    vector<bool> visited(n, false);
    // Start DFS from node 0
    dfs(0, adj, visited, result);
    return result;
}

int main() {
    vector<vector<int>> adj = {{2, 3, 1}, {0}, {0, 4}, {0}, {2}};
    int n = adj.size();
    vector<int> dfsResult = dfsOfGraph(n, adj);
    for (int node : dfsResult) {
        cout << node << " ";
    }
    return 0;
}

```

Output:



```

Output
0 2 4 3 1

```

Ques 5:

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.

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.

Code:

```

#include <iostream>
#include <vector>

```

```

#include <queue>
using namespace std;
vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
    vector<vector<int>> adj(numCourses);
    vector<int> indegree(numCourses, 0);
    vector<int> order;
    for (auto& prereq : prerequisites) {
        adj[prereq[1]].push_back(prereq[0]);
        indegree[prereq[0]]++;
    }
    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 : adj[course]) {
            indegree[neighbor]--;
            if (indegree[neighbor] == 0) {
                q.push(neighbor);
            }
        }
    }
    if (order.size() != numCourses) {
        return {};
    }
}

```

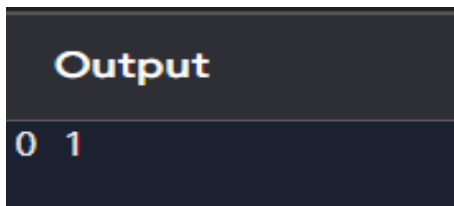
```

        return order;
    }

int main() {
    int numCourses = 2;
    vector<vector<int>> prerequisites = {{1, 0}};
    vector<int> courseOrder = findOrder(numCourses, prerequisites);
    for (int course : courseOrder) {
        cout << course << " ";
    }
    return 0;
}

```

Output:



The screenshot shows a dark-themed terminal window. At the top, the word "Output" is displayed in a light blue font. Below it, the numbers "0 1" are printed in a light blue font, representing the output of the provided C++ code.

Ques 6:

Word Search

Given an $m \times n$ 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.

Code:

```

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

bool dfs(vector<vector<char>>& board, string& word, int i, int j, int index) {
    if (index == word.size()) return true; // Word found
    if (i < 0 || j < 0 || i >= board.size() || j >= board[0].size() || board[i][j] != word[index])
        return false; // Out of bounds or mismatch
}

```



```

char temp = board[i][j]; // Temporarily mark the cell
board[i][j] = '#';
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);

board[i][j] = temp; // Restore the cell
return found;
}

bool exist(vector<vector<char>>& board, string word) {
    int m = board.size();
    int n = board[0].size();
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (board[i][j] == word[0] && dfs(board, word, i, j, 0)) {
                return true;
            }
        }
    }
    return false;
}

int main() {
vector<vector<char>> board = {
    {'A', 'B', 'C', 'E'},
    {'S', 'F', 'C', 'S'},
    {'A', 'D', 'E', 'E'}
};

string word = "ABCCED";
if (exist(board, word)) {
    cout << "true" << endl;
} else {

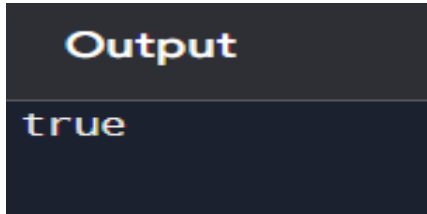
```

```

        cout << "false" << endl;
    }
    return 0;
}

```

Output:



Ques 7:

Rotting Oranges

You are given an $m \times n$ 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.

Code:

```

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

int orangesRotting(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();

    queue<pair<int, int>> q;

    int fresh = 0, minutes = 0;

    // Count fresh oranges and enqueue rotten ones
    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) {
        fresh++;
    }
}
}

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

// BFS traversal
while (!q.empty() && fresh > 0) {
    int size = q.size();
    minutes++;
    for (int i = 0; i < size; i++) {
        auto [x, y] = q.front();
        q.pop();
        for (auto [dx, dy] : directions) {
            int nx = x + dx, ny = y + dy;
            if (nx >= 0 && ny >= 0 && nx < m && ny < n && grid[nx][ny] == 1) {
                grid[nx][ny] = 2;
                fresh--;
                q.push({nx, ny});
            }
        }
    }
}
}

```

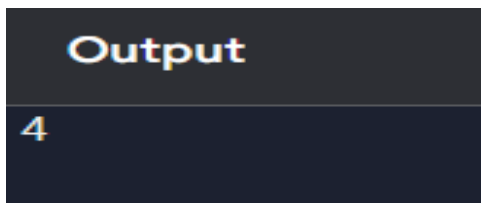
```

    return fresh == 0 ? minutes : -1;
}

int main() {
    vector<vector<int>> grid = {{2, 1, 1}, {1, 1, 0}, {0, 1, 1}};
    cout << orangesRotting(grid) << endl;
    return 0;
}

```

Output:



Ques 8:

Network Delay Time

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 $times[i] = (u_i, v_i, w_i)$, where u_i is the source node, v_i is the target node, and w_i 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.

Code:

```

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

int networkDelayTime(vector<vector<int>>& times, int n, int k) {
    // Create adjacency list
    vector<vector<pair<int, int>>> adj(n + 1);
    for (auto& edge : times) {
        adj[edge[0]].emplace_back(edge[1], edge[2]);
    }
}

```

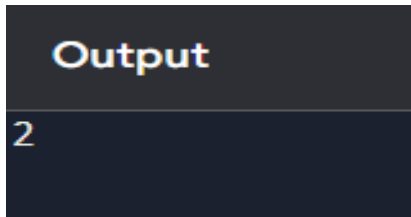
```

    }
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    vector<int> dist(n + 1, INT_MAX);
    dist[k] = 0;
    pq.emplace(0, k);
    while (!pq.empty()) {
        auto [time, node] = pq.top();
        pq.pop();
        if (time > dist[node]) continue;
        for (auto& [neighbor, weight] : adj[node]) {
            if (dist[node] + weight < dist[neighbor]) {
                dist[neighbor] = dist[node] + weight;
                pq.emplace(dist[neighbor], neighbor);
            }
        }
    }
    int maxTime = 0;
    for (int i = 1; i <= n; i++) {
        if (dist[i] == INT_MAX) return -1;
        maxTime = max(maxTime, dist[i]);
    }
    return maxTime;
}

int main() {
    vector<vector<int>> times = {{2, 1, 1}, {2, 3, 1}, {3, 4, 1}};
    int n = 4, k = 2;
    cout << networkDelayTime(times, n, k) << endl;
    return 0;
}

```

Output:



Very Hard

Ques 9:

Redundant Connection

In this problem, a tree is an undirected graph that is connected and has no cycles.

You are given a graph that started as a tree with n nodes labeled from 1 to n , with one additional edge added. The added edge has two different vertices chosen from 1 to n , and was not an edge that already existed. The graph is represented as an array `edges` of length n where `edges[i] = [ai, bi]` indicates that there is an edge between nodes ai and bi in the graph.

Return an edge that can be removed so that the resulting graph is a tree of n nodes. If there are multiple answers, return the answer that occurs last in the input.

Code:

```
#include <iostream>

#include <vector>

using namespace std;

class UnionFind {
private:
    vector<int> parent, rank;

public:
    UnionFind(int n) {
        parent.resize(n + 1);
        rank.resize(n + 1, 0);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
        }
    }
}
```

```

int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]); // Path compression
    }
    return parent[x];
}

bool unionSet(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);
    if (rootX == rootY) return false; // Cycle detected
    if (rank[rootX] > rank[rootY]) {
        parent[rootY] = rootX;
    } else if (rank[rootX] < rank[rootY]) {
        parent[rootX] = rootY;
    } else {
        parent[rootY] = rootX;
        rank[rootX]++;
    }
    return true;
}

};

vector<int> findRedundantConnection(vector<vector<int>>& edges) {
    int n = edges.size();
    UnionFind uf(n);

```

```

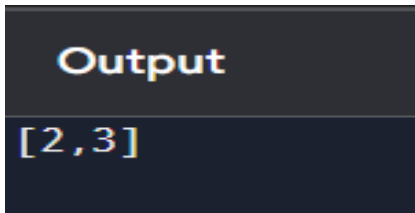
for (auto& edge : edges) {
    if (!uf.unionSet(edge[0], edge[1])) {
        return edge; // Edge causing the cycle
    }
}

return {};
}

int main() {
    vector<vector<int>> edges = {{1, 2}, {1, 3}, {2, 3}};
    vector<int> result = findRedundantConnection(edges);
    cout << "[" << result[0] << "," << result[1] << "]" << endl;
    return 0;
}

```

Output:



```

Output
[2,3]

```

Ques 10:

Shortest Path in Binary Matrix

Shortest Path in Binary Matrix Given an $n \times n$ binary matrix grid, return the length of the shortest clear path in the matrix. If there is no clear path, return -1.

A clear path in a binary matrix is a path from the top-left cell (i.e., $(0, 0)$) to the bottom-right cell (i.e., $(n - 1, n - 1)$) such that:

All the visited cells of the path are 0.

All the adjacent cells of the path are 8-directionally connected (i.e., they are different and they share an edge or a corner).

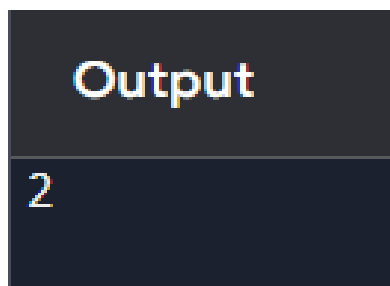
The length of a clear path is the number of visited cells of this path.

Code:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
int shortestPathBinaryMatrix(vector<vector<int>>& grid) {
    int n = grid.size();
    if (grid[0][0] != 0 || grid[n - 1][n - 1] != 0) {
        return -1; // Start or end cell is blocked
    }
    vector<pair<int, int>> directions = {
        {-1, -1}, {-1, 0}, {-1, 1},
        {0, -1},      {0, 1},
        {1, -1}, {1, 0}, {1, 1}
    };
    queue<pair<int, int>> q;
    q.push({0, 0});
    grid[0][0] = 1; // Mark visited and store the distance
    while (!q.empty()) {
        auto [x, y] = q.front();
        int distance = grid[x][y];
        q.pop();
        if (x == n - 1 && y == n - 1) {
            return distance; // Reached the bottom-right corner
        }
        for (auto [dx, dy] : directions) {
            int nx = x + dx, ny = y + dy;
            if (nx >= 0 && ny >= 0 && nx < n && ny < n && grid[nx][ny] == 0) {
                q.push({nx, ny});
                grid[nx][ny] = distance + 1; // Update distance
            }
        }
    }
}
```

```
    }  
    return -1; // No path found  
}  
int main() {  
    vector<vector<int>> grid = {{0, 1}, {1, 0}};  
    cout << shortestPathBinaryMatrix(grid) << endl;  
    return 0;  
}
```

Output:



Output

2