Experiment 9 (Advance Programming)

Submitted By: Ankit Kharb **Class:** IOT_614(B)

UID: 22BCS16964

Q-Number of Islands

Problem:
Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

Input: grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

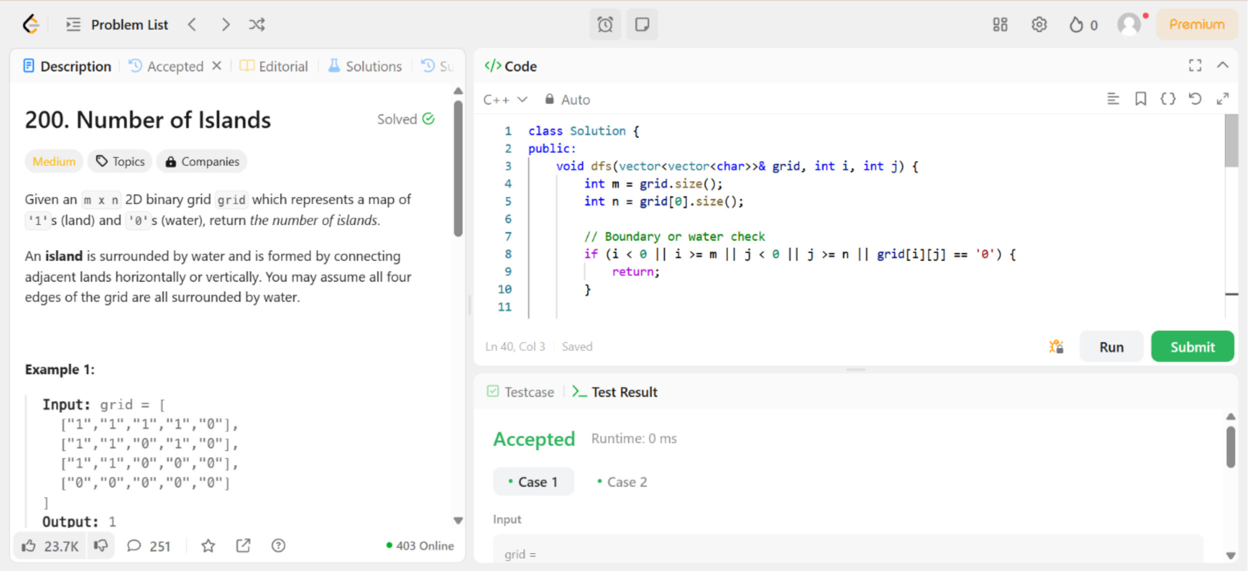
["0","0","0","0","0"]

]

Output: 1

Solution:

```
class Solution {
public:
    void dfs(vector<vector<char>>& grid, int i, int j) {
        int m = grid.size();
        int n = grid[0].size();

        // Boundary or water check
    if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] == '0') {
        return;
    }

    // Mark the current land cell as visited
    grid[i][j] = '0';

    // Explore all 4 directions
    dfs(grid, i - 1, j); // up
    dfs(grid, i + 1, j); // down
    dfs(grid, i, j - 1); // left
    dfs(grid, i, j + 1); // right
}

int numIslands(vector<vector<char>>& grid) {
    if (grid.empty()) return 0;

    int count = 0;
    int m = grid.size();
    int n = grid[0].size();

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (grid[i][j] == '1') {
                ++count;
                dfs(grid, i, j);
            }
            }
        }

        return count;
    }
};
```



200. Number of Islands

Solved

Given an `m x n` 2D binary grid `grid` which represents a map of `'1'`s (land) and `'0'`s (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
Input: grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
Output: 1
```

```
class Solution {
public:
    void dfs(vector<vector<char>>& grid, int i, int j) {
        int m = grid.size();
        int n = grid[0].size();

        // Boundary or water check
        if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] == '0') {
            return;
        }

```

Accepted Runtime: 0 ms

Input

grid =

Q-Word Ladder

Problem:

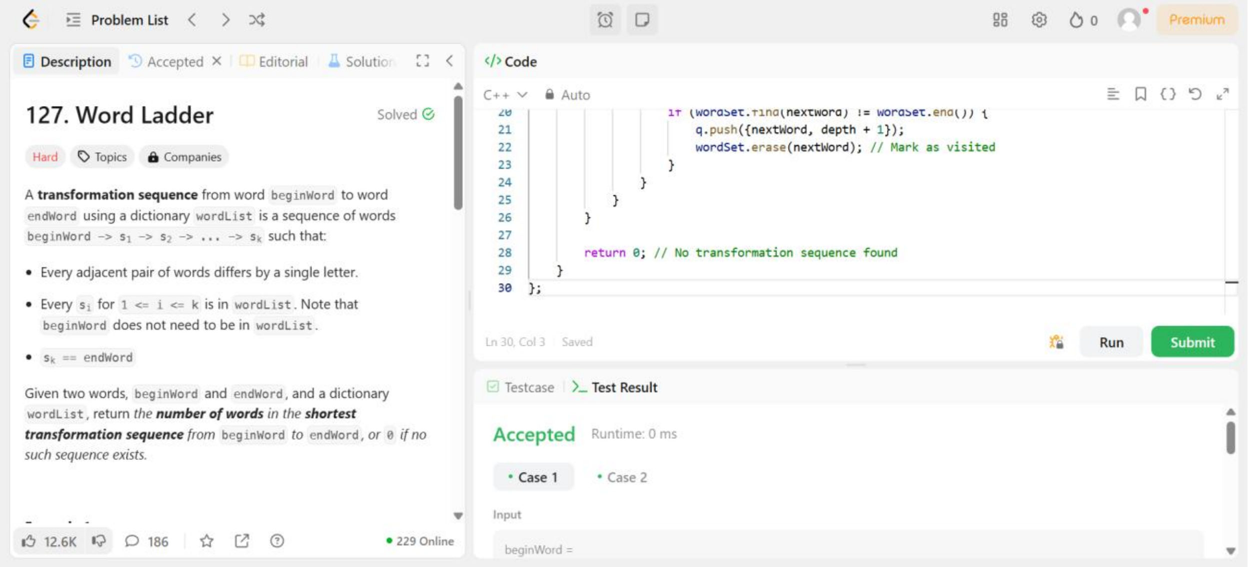
A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s_1 -> s_2 -> ... -> s_k such that:

- Every adjacent pair of words differs by a single letter.
- Every s_i for $1 <= i <= k$ is in wordList. Note that beginWord does not need to be in wordList.
- s_k == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return *the **number of words** in the **shortest transformation sequence** from* beginWord *to* endWord*, or* 0 *if no such sequence exists.*

Solution:

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
        unordered_set<string> wordSet(wordList.begin(), wordList.end());
        if (wordSet.find(endWord) == wordSet.end()) return 0;

        queue<pair<string, int>> q; // Pair of (current word, current depth)
        q.push({beginWord, 1});

        while (!q.empty()) {
            auto [word, depth] = q.front();
            q.pop();

            if (word == endWord) return depth;

            for (int i = 0; i < word.size(); ++i) {
                string nextWord = word;
                for (char c = 'a'; c <= 'z'; ++c) {
                    nextWord[i] = c;
                    if (wordSet.find(nextWord) != wordSet.end()) {
                        q.push({nextWord, depth + 1});
                        wordSet.erase(nextWord); // Mark as visited
                    }
                }
            }
        }

        return 0; // No transformation sequence found
    }
};
```



127. Word Ladder

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every `si` for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- `sk == endWord`

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *the **number of words** in the **shortest transformation sequence** from* `beginWord` *to* `endWord`*, or* `0` *if no such sequence exists.*

```
                    if (wordSet.find(nextWord) != wordSet.end()) {
                        q.push({nextWord, depth + 1});
                        wordSet.erase(nextWord); // Mark as visited
                    }
                }
            }
        }

        return 0; // No transformation sequence found
    }
};
```

Accepted Runtime: 0 ms

beginWord =

Q-Surrounded Regions

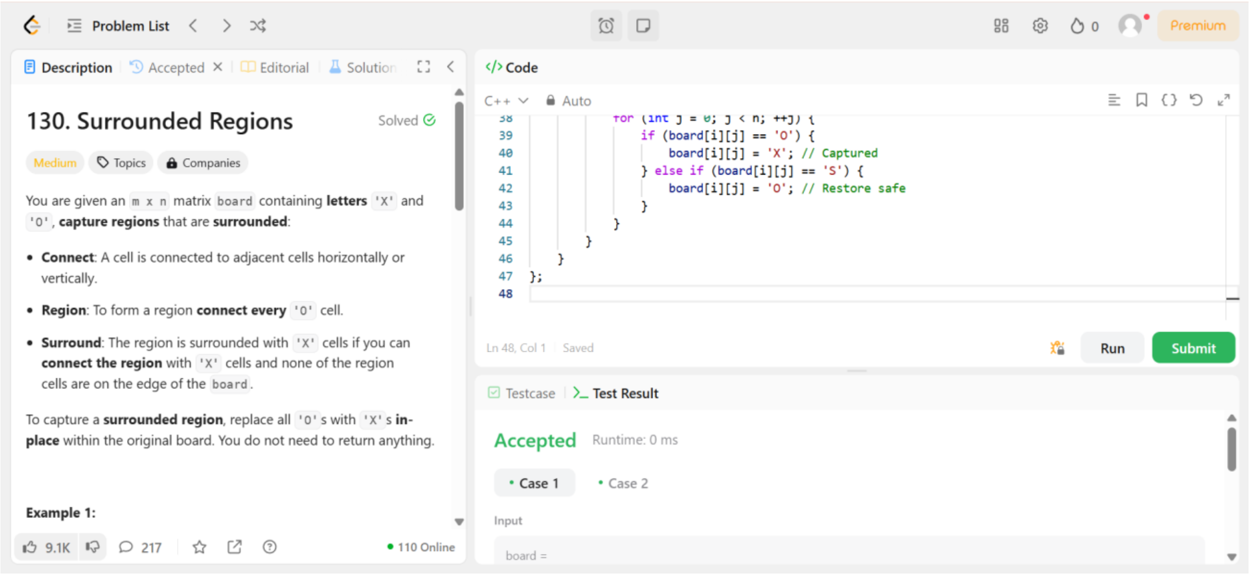
You are given an m x n matrix board containing **letters** 'X' and 'O', **capture regions** that are **surrounded**:

- **Connect**: A cell is connected to adjacent cells horizontally or vertically.
- **Region**: To form a region **connect every** 'O' cell.
- **Surround**: The region is surrounded with 'X' cells if you can **connect the region** with 'X' cells and none of the region cells are on the edge of the board.

To capture a **surrounded region**, replace all 'O's with 'X's **in-place** within the original board. You do not need to return anything.

Solution:

```
class Solution {
public:
    void dfs(vector<vector<char>>& board, int i, int j) {
        int m = board.size();
        int n = board[0].size();

        if (i < 0 || i >= m || j < 0 || j >= n || board[i][j] != 'O') {
            return;
        }

        board[i][j] = 'S'; // Temporarily mark as safe

        // Explore 4 directions
        dfs(board, i - 1, j);
        dfs(board, i + 1, j);
        dfs(board, i, j - 1);
        dfs(board, i, j + 1);
    }

    void solve(vector<vector<char>>& board) {
        if (board.empty()) return;

        int m = board.size();
        int n = board[0].size();

        // Step 1: Mark border-connected 'O's
        for (int i = 0; i < m; ++i) {
            if (board[i][0] == 'O') dfs(board, i, 0);
            if (board[i][n - 1] == 'O') dfs(board, i, n - 1);
        }
        for (int j = 0; j < n; ++j) {
            if (board[0][j] == 'O') dfs(board, 0, j);
            if (board[m - 1][j] == 'O') dfs(board, m - 1, j);
        }

        // Step 2: Flip the regions
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (board[i][j] == 'O') {
                    board[i][j] = 'X'; // Captured
                } else if (board[i][j] == 'S') {
                    board[i][j] = 'O'; // Restore safe
                }
            }
        }
    }
};
```



130. Surrounded Regions

Solved

Medium · Topics · Companies

You are given an `m x n` matrix `board` containing **letters** `'X'` and `'O'`, **capture regions** that are **surrounded**:

- **Connect**: A cell is connected to adjacent cells horizontally or vertically.
- **Region**: To form a region **connect every** `'O'` cell.
- **Surround**: The region is surrounded with `'X'` cells if you can **connect the region** with `'X'` cells and none of the region cells are on the edge of the `board`.

To capture a **surrounded region**, replace all `'O'`s with `'X'`s **in-place** within the original board. You do not need to return anything.

Example 1:

```
            for (int j = 0; j < n; ++j) {
                if (board[i][j] == 'O') {
                    board[i][j] = 'X'; // Captured
                } else if (board[i][j] == 'S') {
                    board[i][j] = 'O'; // Restore safe
                }
            }
        }
    }
};
```

Accepted   Runtime: 0 ms

• Case 1   • Case 2

Input

board =

Q-Binary Tree Maximum Path Sum

Problem:

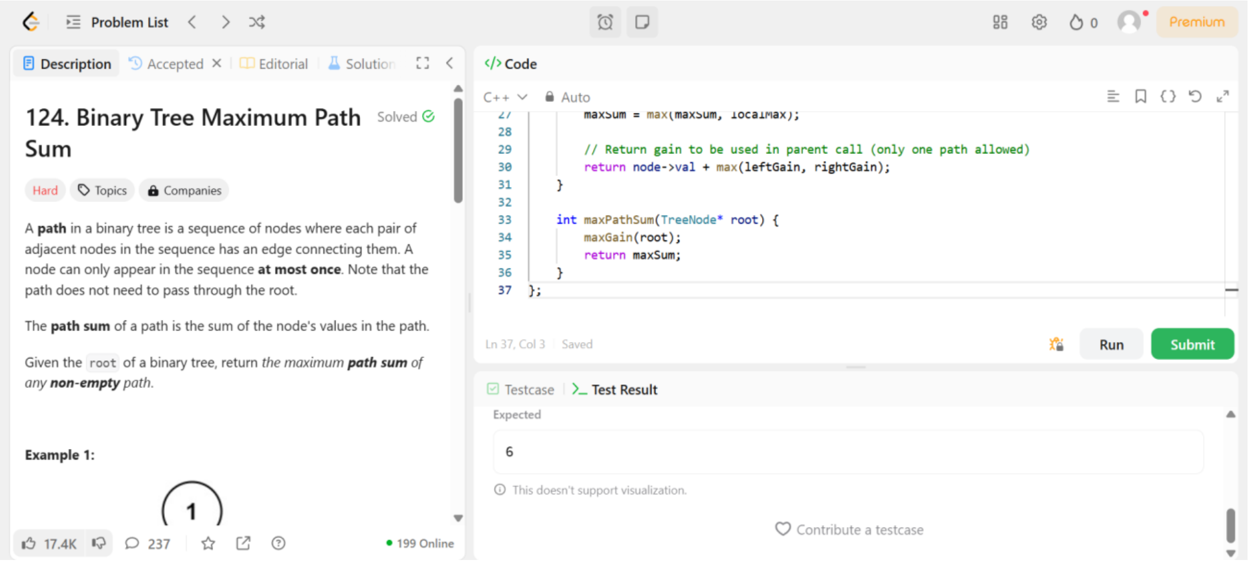
A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

Given the root of a binary tree, return *the maximum **path sum** of any **non-empty** path*.

Solution:

```
class Solution {
public:
    int maxSum = INT_MIN;

    int maxGain(TreeNode* node) {
        if (!node) return 0;

        // Only consider positive gains; discard negative paths
        int leftGain = max(maxGain(node->left), 0);
        int rightGain = max(maxGain(node->right), 0);

        // Max path using this node as root of the path
        int localMax = node->val + leftGain + rightGain;

        // Update global max
        maxSum = max(maxSum, localMax);

        // Return gain to be used in parent call (only one path allowed)
        return node->val + max(leftGain, rightGain);
    }

    int maxPathSum(TreeNode* root) {
        maxGain(root);
        return maxSum;
    }
}
```



124. Binary Tree Maximum Path Sum

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

Given the `root` of a binary tree, return *the maximum* ***path sum*** *of any* ***non-empty*** *path.*

Example 1:

```
        maxSum = max(maxSum, localMax);

        // Return gain to be used in parent call (only one path allowed)
        return node->val + max(leftGain, rightGain);
    }

    int maxPathSum(TreeNode* root) {
        maxGain(root);
        return maxSum;
    }
};
```

Expected

6

This doesn't support visualization.

Contribute a testcase

Q-Friend Circles

Problem:

There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b, and city b is connected directly with city c, then city a is connected indirectly with city c.

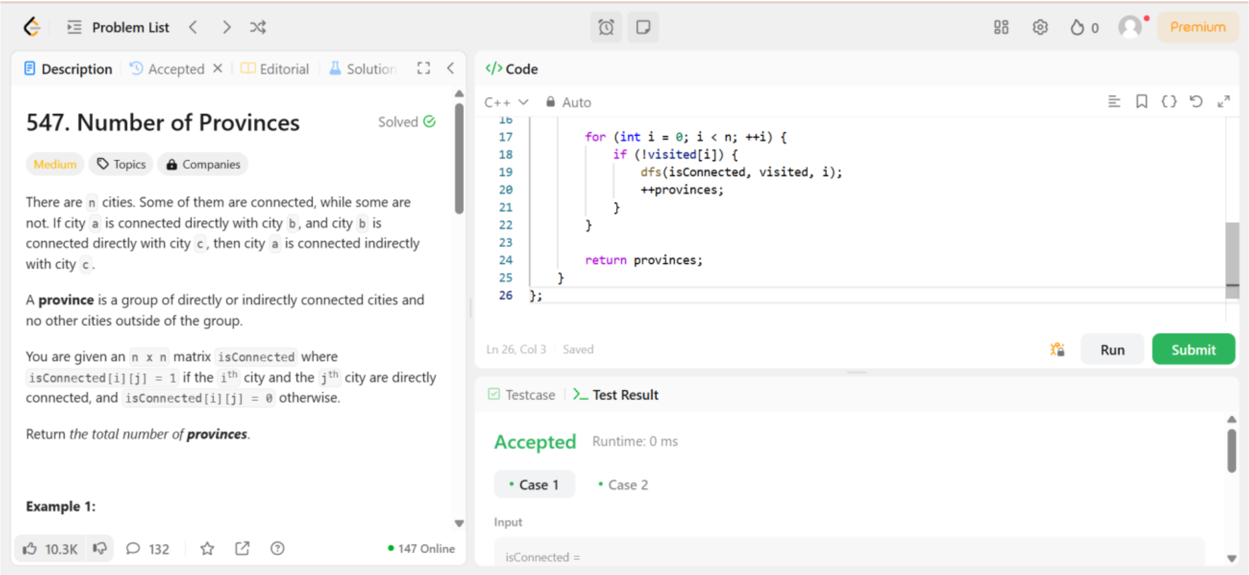
A **province** is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an n x n matrix isConnected where isConnected[i][j] = 1 if the i^{th} city and the j^{th} city are directly connected, and isConnected[i][j] = 0 otherwise.

Return *the total number of **provinces***.

Solution:

```
class Solution {

public:

    void dfs(vector<vector<int>>& isConnected, vector<bool>& visited, int city) {

        visited[city] = true;

        for (int j = 0; j < isConnected.size(); ++j) {

            if (isConnected[city][j] == 1 && !visited[j]) {

                dfs(isConnected, visited, j);

            }

        }

    }


    int findCircleNum(vector<vector<int>>& isConnected) {

        int n = isConnected.size();

        vector<bool> visited(n, false);

        int provinces = 0;


        for (int i = 0; i < n; ++i) {
            if (!visited[i]) {
                dfs(isConnected, visited, i);
                ++provinces;
            }
        }

        return provinces;
    }
};    vector<vector<int>> skyline = getSkyline(buildings);

    cout << "Skyline: ";
    for (const auto& point : skyline) {
        cout << "[" << point[0] << ", " << point[1] << "] ";
    }
    cout << endl;

    return 0;
}
```



547. Number of Provinces

Solved

There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b, and city b is connected directly with city c, then city a is connected indirectly with city c.

A **province** is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an n x n matrix isConnected where isConnected[i][j] = 1 if the i^{th} city and the j^{th} city are directly connected, and isConnected[i][j] = 0 otherwise.

Return *the total number of* ***provinces***.

Example 1:

```
        for (int i = 0; i < n; ++i) {
            if (!visited[i]) {
                dfs(isConnected, visited, i);
                ++provinces;
            }
        }

        return provinces;
    }
};
```

Ln 26, Col 3 | Saved

Accepted Runtime: 0 ms

Input

isConnected =