Experiment-9

Student Name: Gaurav Saini
UID: 22BCS11085
Branch: BE-CSE
Section/Group:NTPP_IOT_603_B
Semester:06
Date of Performance:04-04-2025

Subject Name: AP LAB-II
Subject Code: 22CSP-351

1. **Aim:**
   a. **Lowest Common Ancestor of a Binary Tree**
   b. **Number of Islands**
   c. **Course Schedule**

2. **Introduction to Graphs:**

- A graph is a data structure that represents relationships between objects using:
- Nodes (Vertices): Represent entities in the graph.
- Edges: Represent connections/relationships between nodes.

Types of Nodes

- Root Node: The starting point of a graph with no ancestors.
- Leaf Nodes: Nodes with no outgoing edges, only incoming connections.

3. Implementation/Code:

A. Lowest Common Ancestor of a Binary Tree

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (root == nullptr || root == p || root == q) {
            return root;
```

```
        }

        TreeNode* left = lowestCommonAncestor(root->left, p, q);

        TreeNode* right = lowestCommonAncestor(root->right, p, q);


        if (left != nullptr && right != nullptr) {

            return root;

        }

        return left != nullptr ? left : right;


    }

};
```

B. Number of Islands

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        int count = 0;

        function<void(int, int)> dfs = [&](int i, int j) {
            if (i < 0 || j < 0 || i >= m || j >= n || grid[i][j] == '0')
                return;

            grid[i][j] = '0';

            dfs(i + 1, j);
            dfs(i - 1, j);
            dfs(i, j + 1);
            dfs(i, j - 1);
```

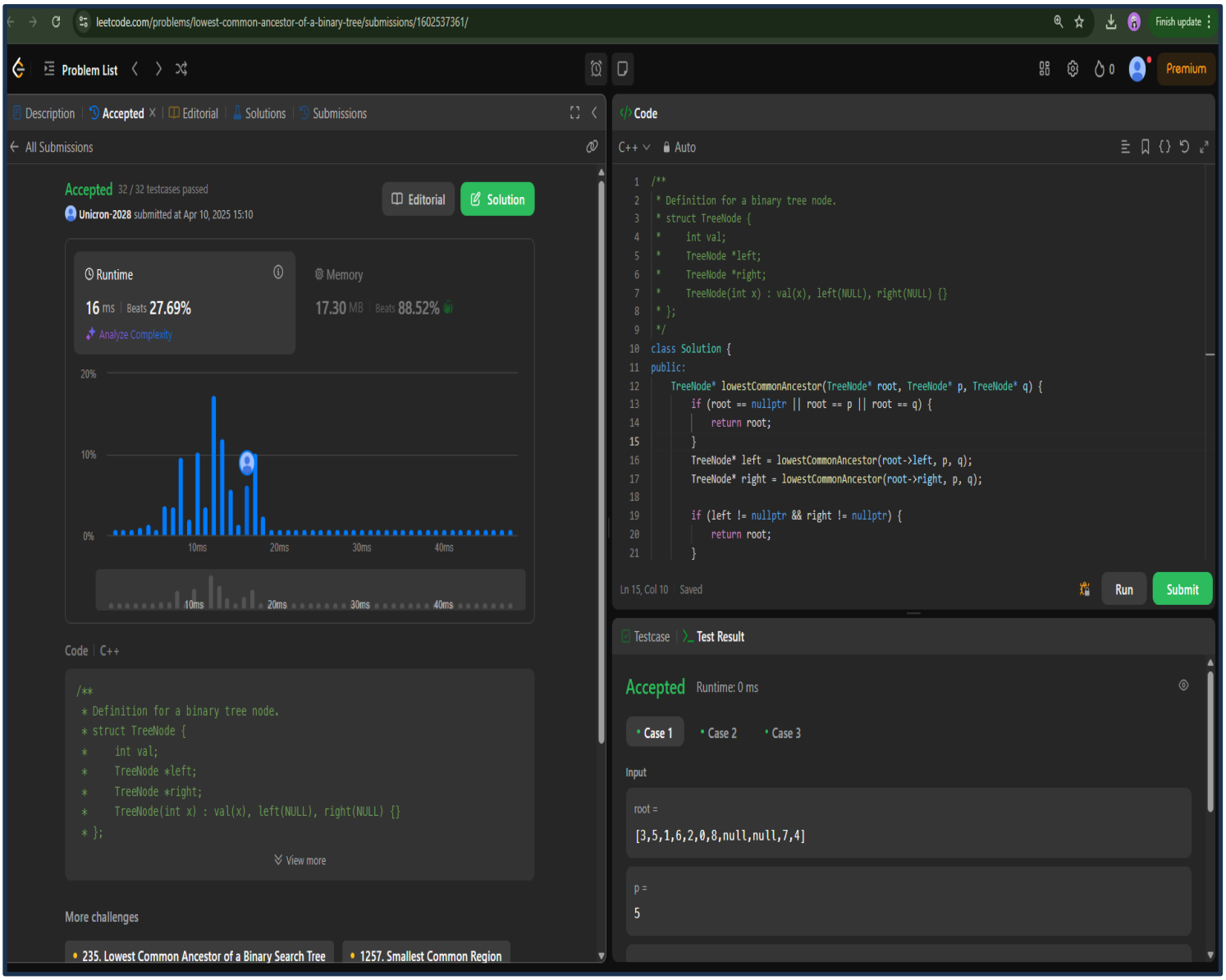
```
        };


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


        return count;
    }
};
```

C. Course Schedule

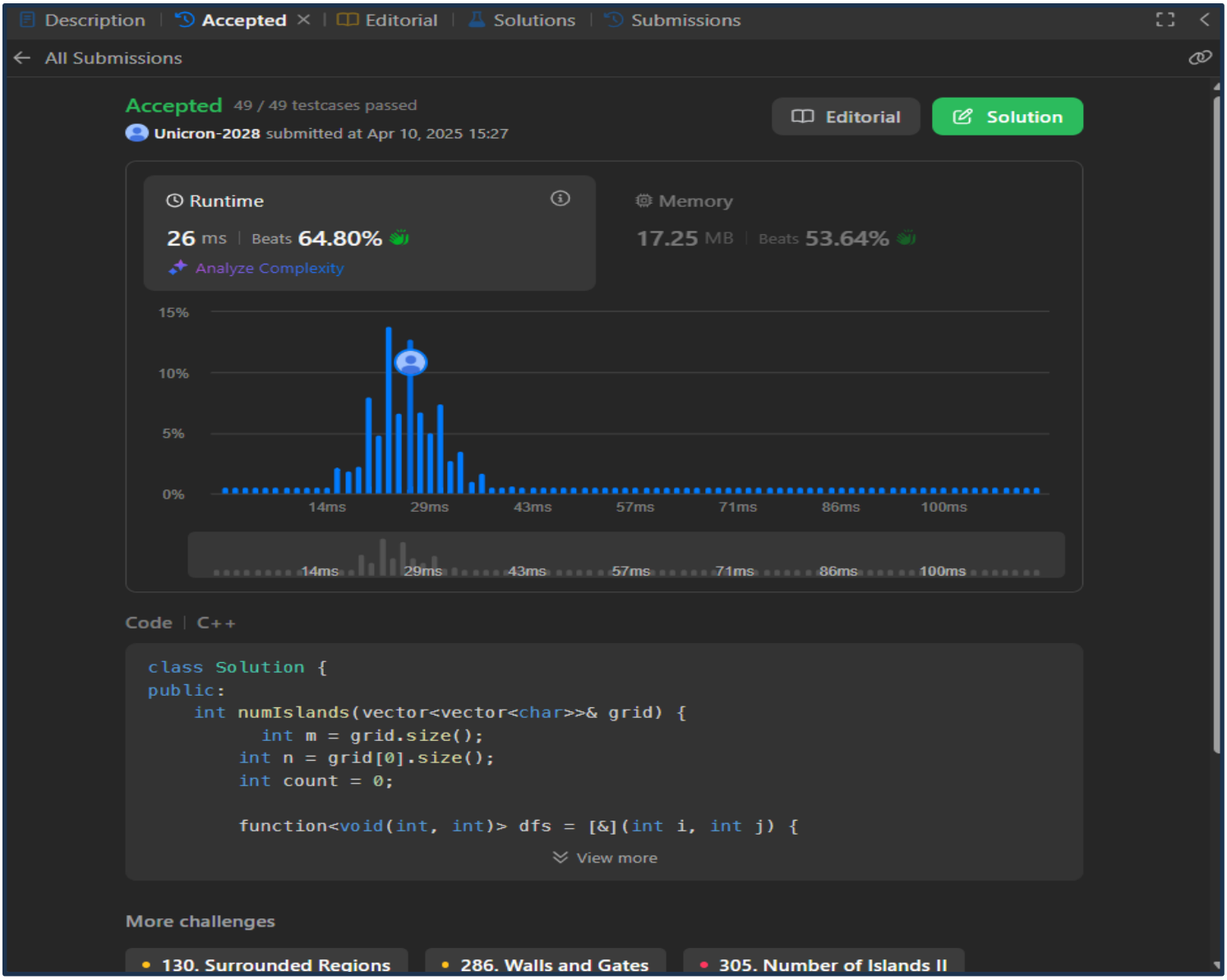
```
class Solution {
public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        vector<vector<int>> adj(numCourses);
        vector<int> indegree(numCourses, 0);

        for (auto& pre : prerequisites) {
            adj[pre[1]].push_back(pre[0]);
            indegree[pre[0]]++;
        }

        queue<int> q;

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

        int count = 0;

        while (!q.empty()) {
            int curr = q.front();
            q.pop();
            count++;

            for (int next : adj[curr]) {
                indegree[next]--;
                if (indegree[next] == 0)
                    q.push(next);
            }
        }
        return count == numCourses;
    }
};
```

4. Output

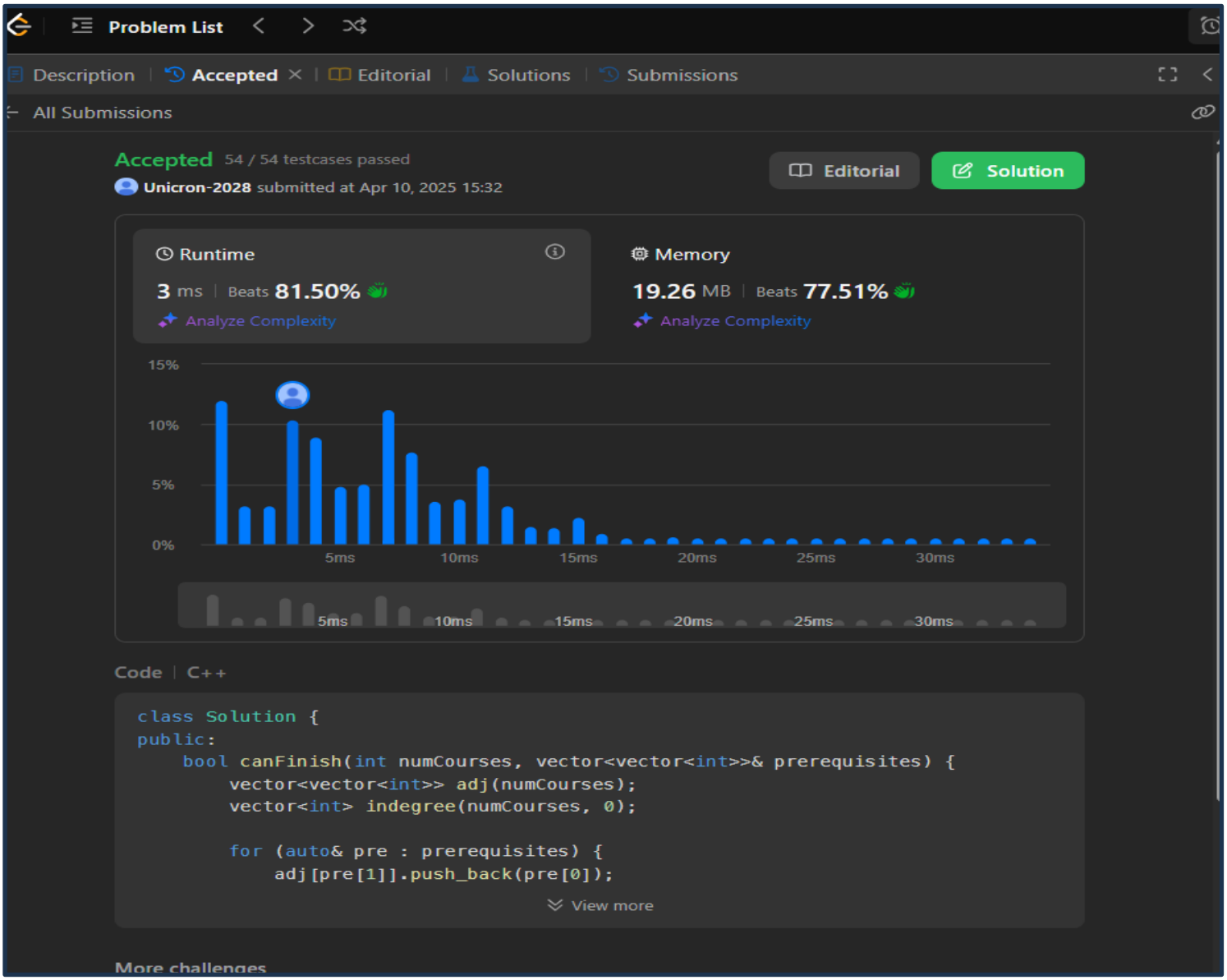
A. Lowest Common Ancestor of a Binary Tree



B. Number Of Islands



C. Course Schedule



5. Learning Outcomes:

➢ **Graph Representation:**

- Learn how to represent a graph using adjacency list or matrix.
- Understand the direction of edges in directed vs undirected graphs.

➢ **DFS (Depth-First Search):**

- Apply recursive DFS to explore connected components (e.g., islands).
- Use DFS for marking visited nodes and exploring all reachable areas from a given node.

➢ **BFS (Breadth-First Search):**

- Implement BFS for topological sorting (Kahn's Algorithm).
- Learn how to manage in-degree of nodes to determine processing order.

➢ **Cycle Detection in Graphs:**

- Use DFS or BFS (Kahn's) to detect cycles in directed graphs.
- Understand the importance of detecting cycles in scheduling/dependency problems.