**ADVANCED PROGRAMMING-II**

**ASSIGNMENT-09**

**Q1. Number of Islands:  
Code:**

class Solution {

public:

    void dfs(vector<vector<char>>& grid, int i, int j) {

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

        if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] != '1')

            return;

        grid[i][j] = '0';

        dfs(grid, i + 1, j);

        dfs(grid, i - 1, j);

        dfs(grid, i, j + 1);

        dfs(grid, i, j - 1);

    }

    int numIslands(vector<vector<char>>& grid) {

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

        int islands = 0;

        for (int i = 0; i < m; ++i) {

            for (int j = 0; j < n; ++j) {

                if (grid[i][j] == '1') {

                    ++islands;

                    dfs(grid, i, j);

                }

            }

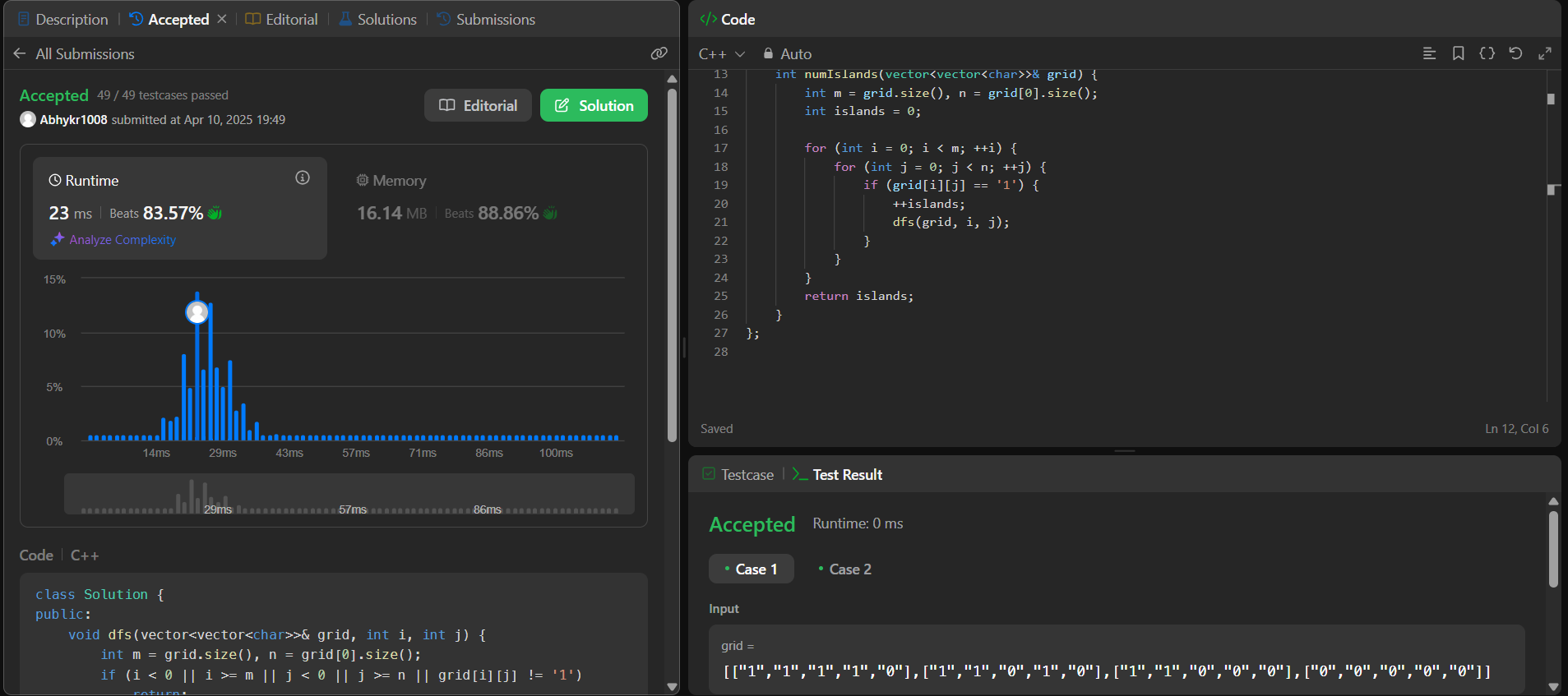
        }

        return islands;

    }

};

**Screenshot:**



**Q2.Word Ladder:**

**Code:**

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;

        q.push({beginWord, 1});

        while (!q.empty()) {

            auto [word, steps] = q.front();

            q.pop();

            for (int i = 0; i < word.size(); ++i) {

                string temp = word;

                for (char c = 'a'; c <= 'z'; ++c) {

                    temp[i] = c;

                    if (temp == endWord) return steps + 1;

                    if (wordSet.find(temp) != wordSet.end()) {

                        q.push({temp, steps + 1});

                        wordSet.erase(temp);

                    }

                }

            }

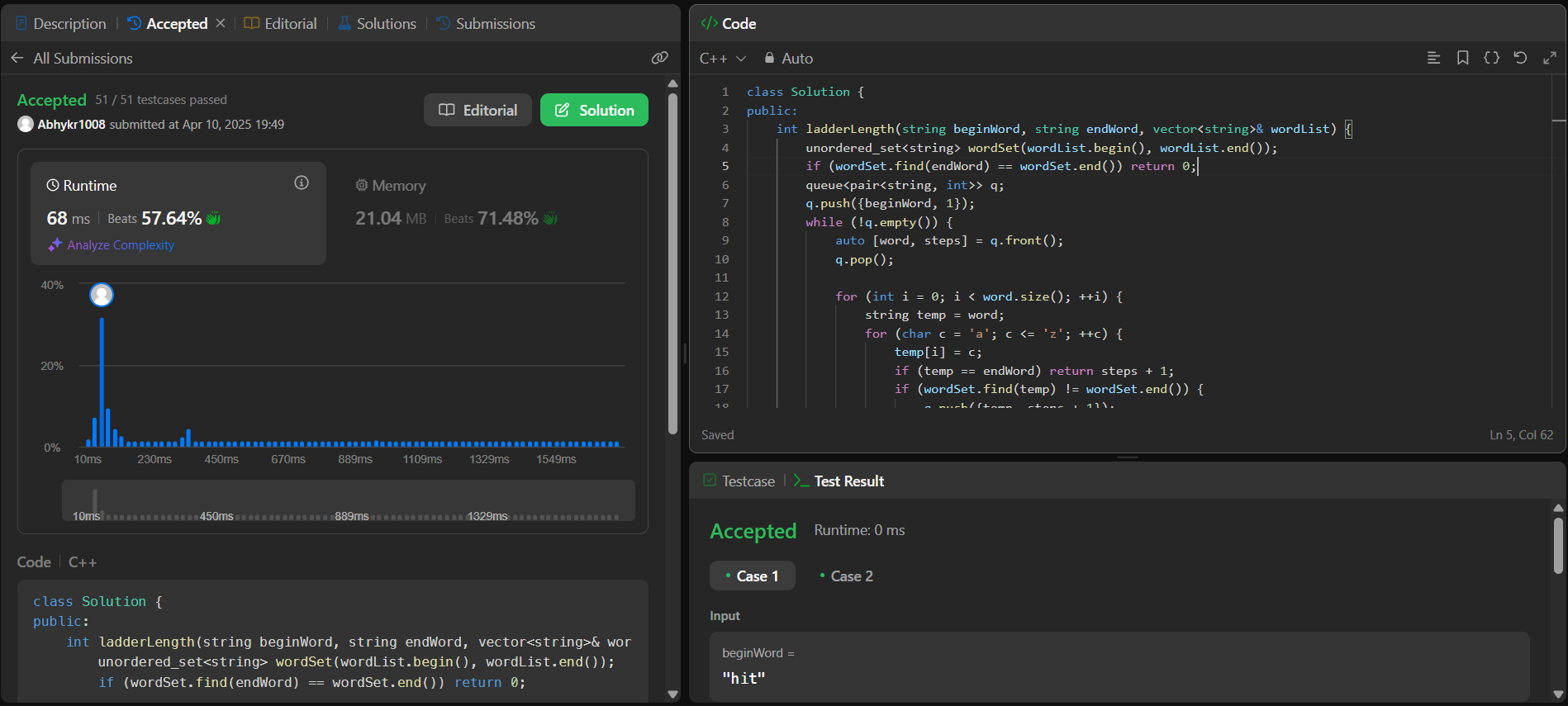
        }

        return 0;

    }

};

**Screenshot:**



**Q3. Surrounded Regions:**

**Code:**

class Solution {

public:

    void dfs(vector<vector<char>>& board, int i, int j) {

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

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

            return;

        board[i][j] = '#';

        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) {

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

        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);

        }

        for (int i = 0; i < m; ++i) {

            for (int j = 0; j < n; ++j) {

                if (board[i][j] == 'O') board[i][j] = 'X';

                else if (board[i][j] == '#') board[i][j] = 'O';

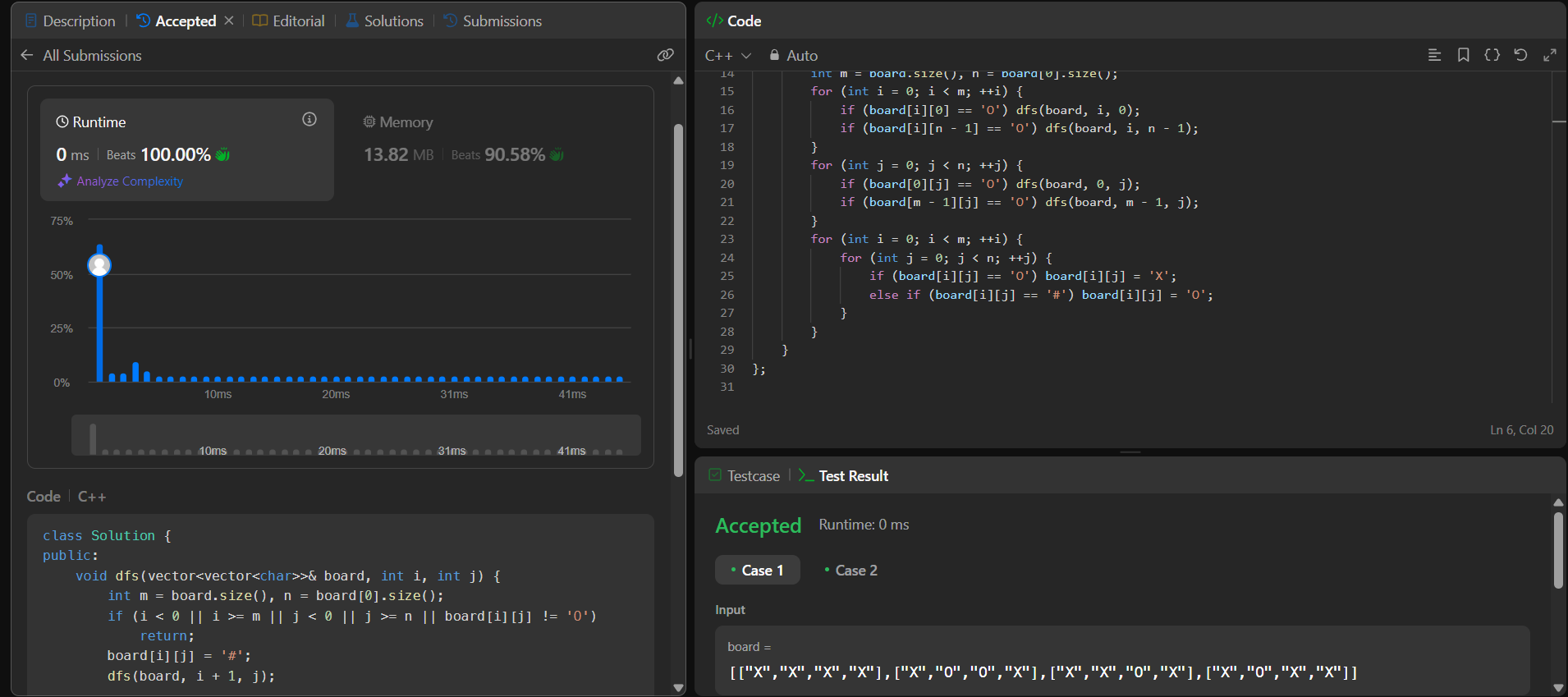
            }

        }

    }

};

**Screenshot:**



**Q4.Friend Circles:**

**Code:**

class Solution {

public:

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

        visited[city] = true;

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

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

                dfs(i, isConnected, visited);

            }

        }

    }

    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(i, isConnected, visited);

                provinces++;

            }

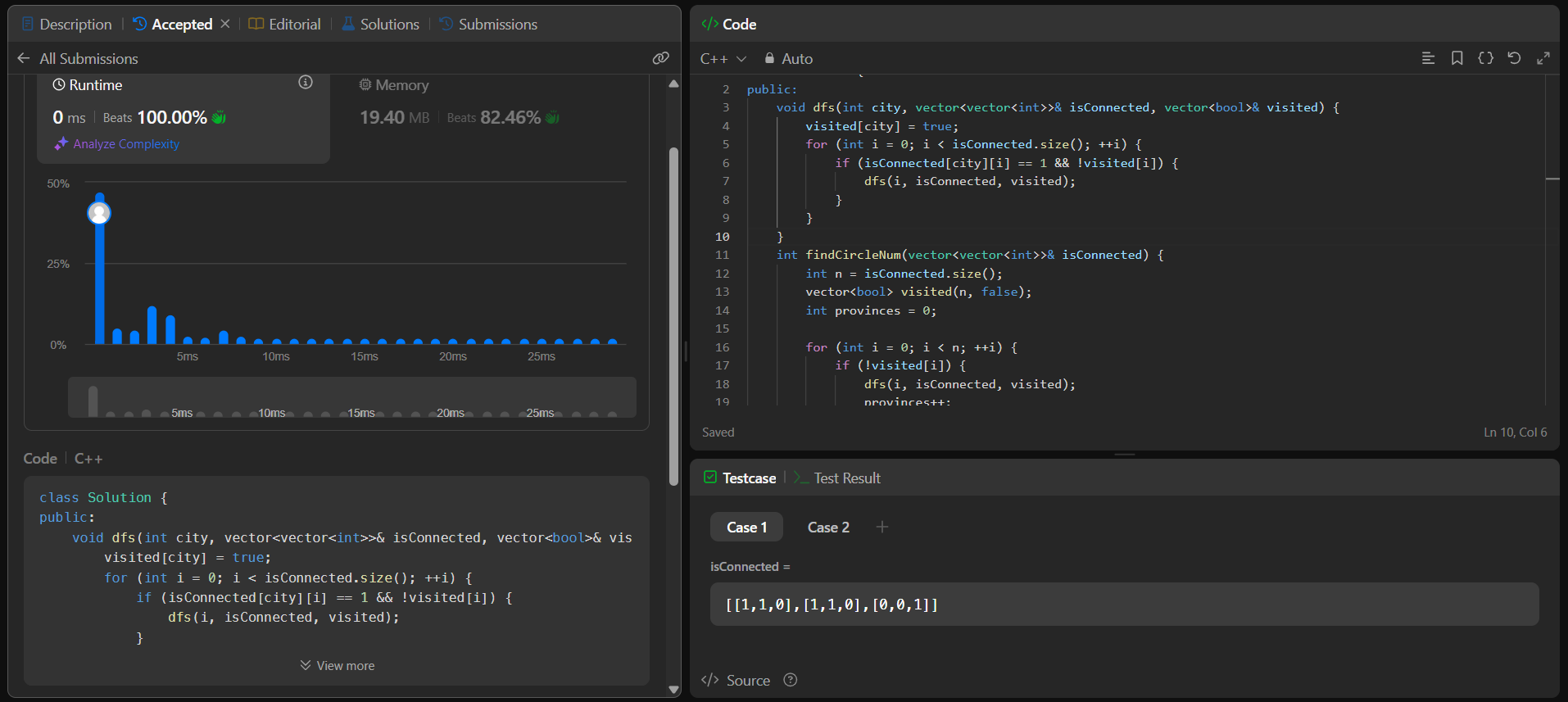
        }

        return provinces;

    }

};

**Screenshot:**



**Q5.Lowest Common Ancestor of a Binary Tree:**

**Code:**

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 && right)

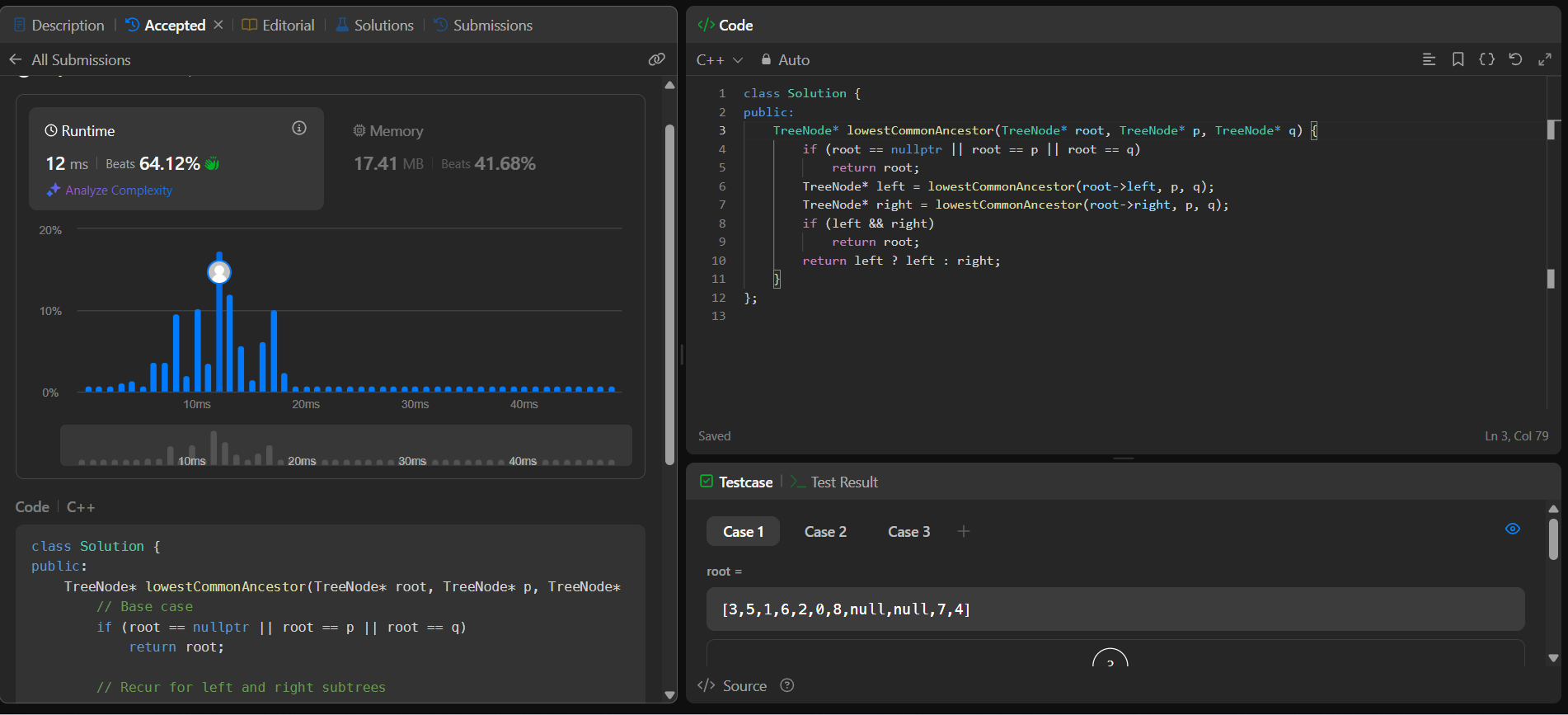
            return root;

        return left ? left : right;

    }

};

**Screenshot:**



**Q6.Course Schedule:**

**Code:**

class Solution {

public:

    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {

        vector<vector<int>> graph(numCourses);

        vector<int> indegree(numCourses, 0);

        for (auto& pre : prerequisites) {

            graph[pre[1]].push\_back(pre[0]);

            indegree[pre[0]]++;

        }    }

        int completed = 0;

        while (!q.empty()) {

            int course = q.front(); q.pop();

            completed++;

            for (int neighbor : graph[course]) {

                indegree[neighbor]--;

                if (indegree[neighbor] == 0)

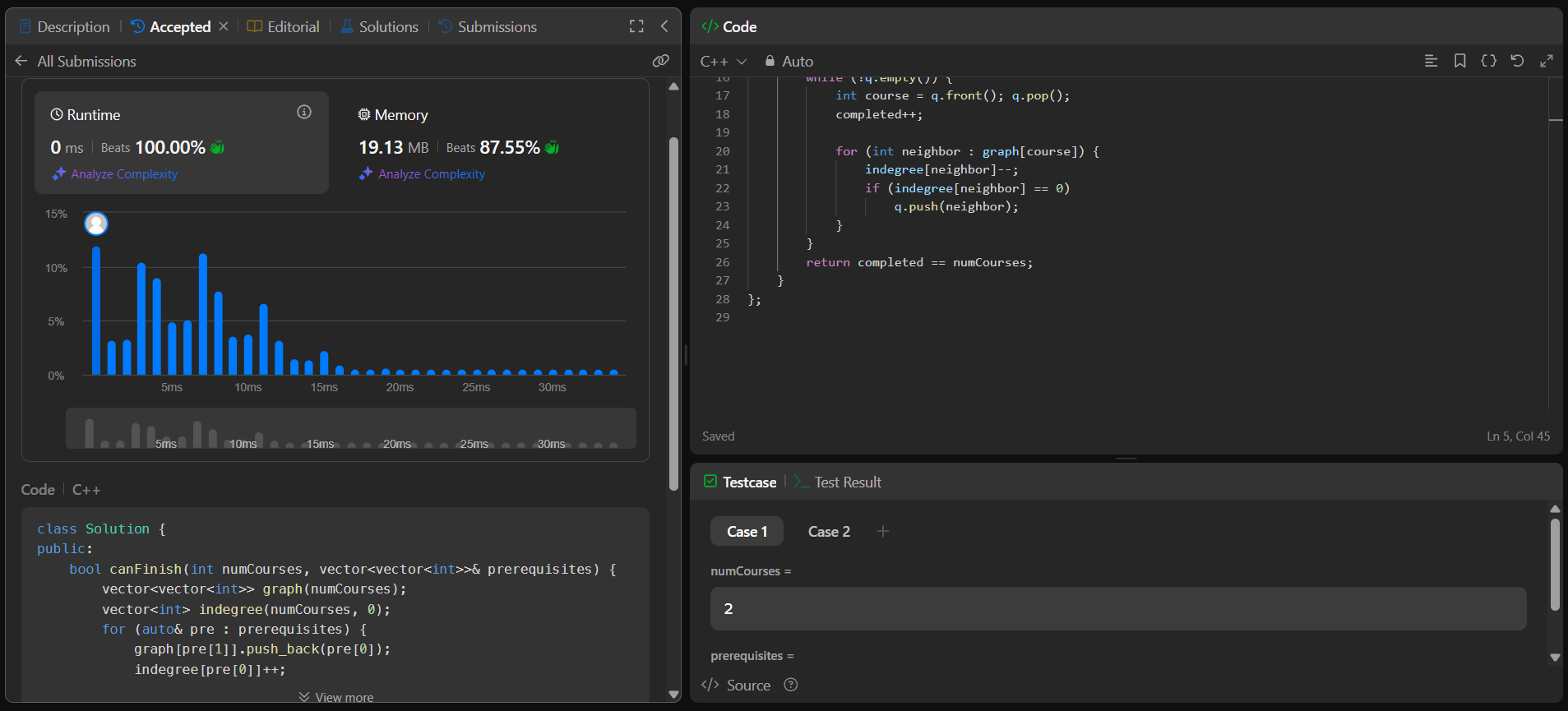
                    q.push(neighbor);

            }  }

        return completed == numCourses;

    }};

**Screenshot:**



**Q7. Longest Increasing Path in a Matrix:**

**Code:**

class Solution {

public:

    int m, n;

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

    int dfs(int i, int j, vector<vector<int>>& matrix, vector<vector<int>>& dp) {

        if (dp[i][j] != 0) return dp[i][j];

        int maxLen = 1;

        for (auto& d : dirs) {

            int x = i + d[0], y = j + d[1];

            if (x >= 0 && x < m && y >= 0 && y < n && matrix[x][y] > matrix[i][j]) {

                maxLen = max(maxLen, 1 + dfs(x, y, matrix, dp));

            }

        }

        return dp[i][j] = maxLen;

    }

    int longestIncreasingPath(vector<vector<int>>& matrix) {

        m = matrix.size();

        n = matrix[0].size();

        vector<vector<int>> dp(m, vector<int>(n, 0));

        int result = 0;

        for (int i = 0; i < m; ++i) {

            for (int j = 0; j < n; ++j) {

                result = max(result, dfs(i, j, matrix, dp));

            }

        }

        return result;

    }

};

**Screenshot:**

