**ASSIGNMENT 9 (Graphs)**

**AP LAB**

**Lavanya Saini – 22BCS15497**

**22BCS IOT 614-B**

1. **Number of Islands**

class Solution {

public:

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

int m = grid.size();

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

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

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();

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

int count = 0;

for (int i = 0; i < m; i++) {

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

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

count++;

dfs(grid, i, j);

}

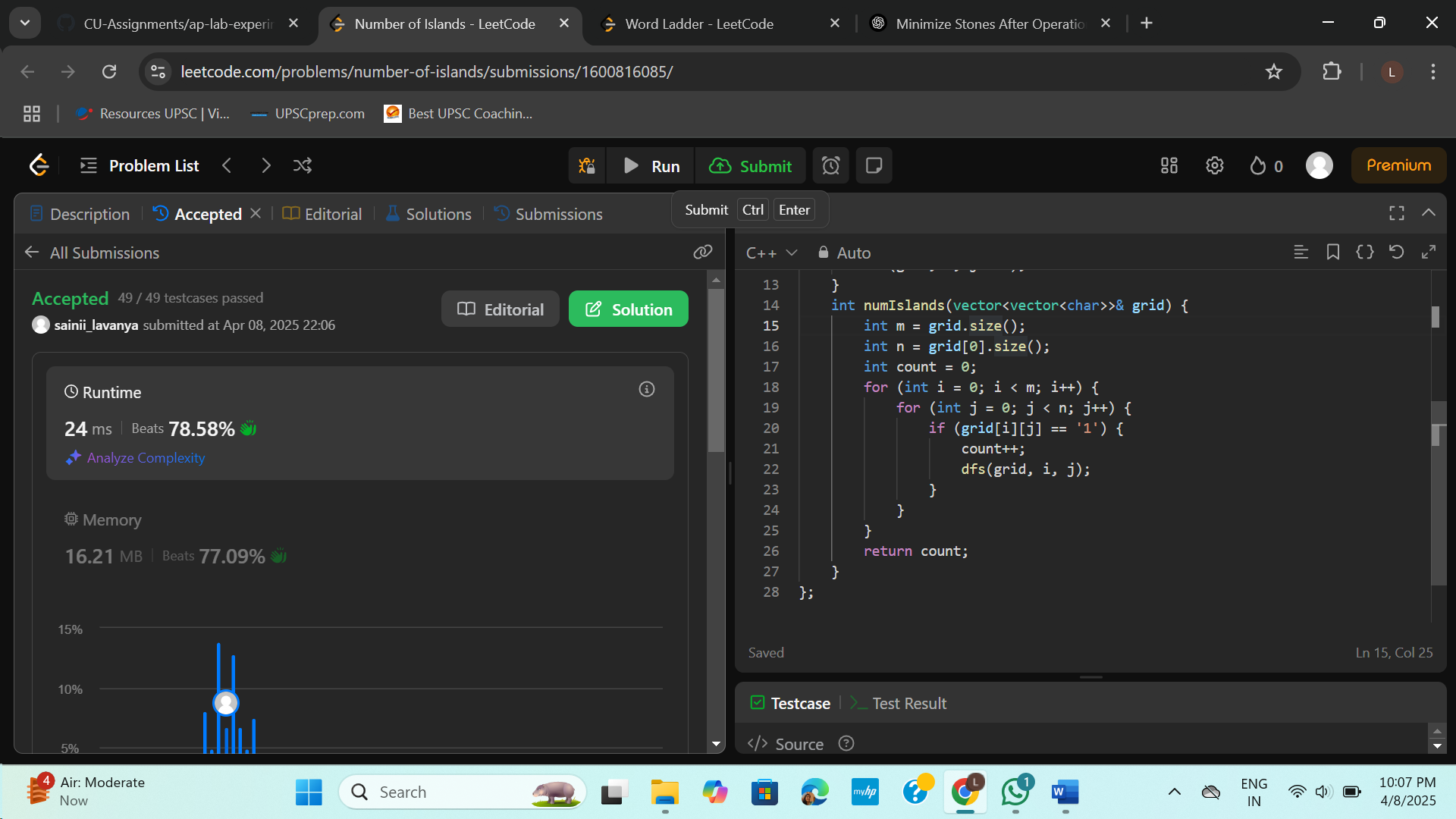
}

}

return count;

}

};



1. **World Ladder**

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, level] = q.front();

q.pop();

if (word == endWord)

return level;

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

string temp = word;

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

temp[i] = ch;

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

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

wordSet.erase(temp);

}

}

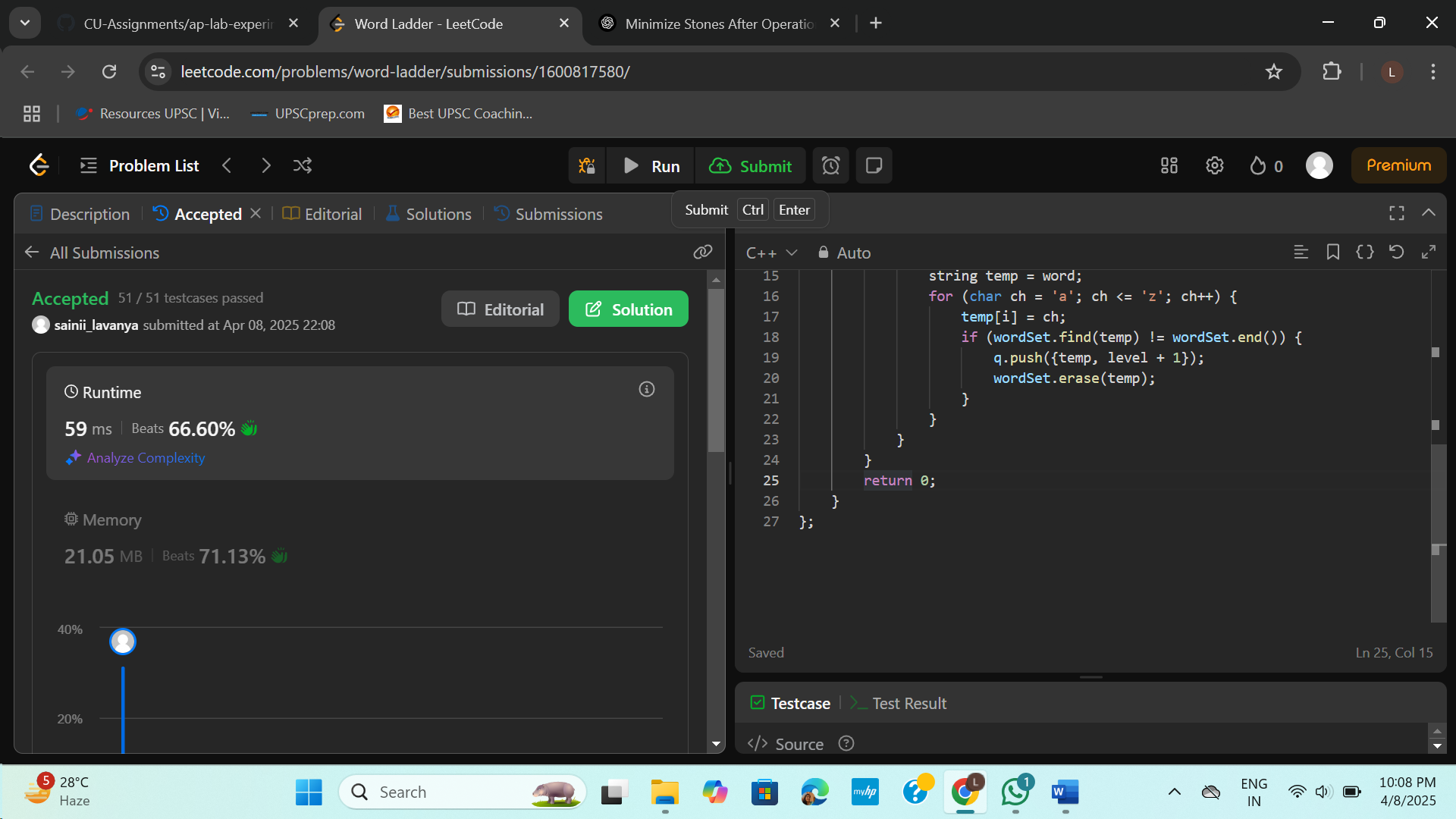
}

}

return 0;

}

};



1. **Surrounded Regions**

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 || j < 0 || i >= m || 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();

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

for (int i = 0; i < m; i++) {

dfs(board, i, 0);

dfs(board, i, n - 1);

}

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

dfs(board, 0, j);

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';

if (board[i][j] == '#')

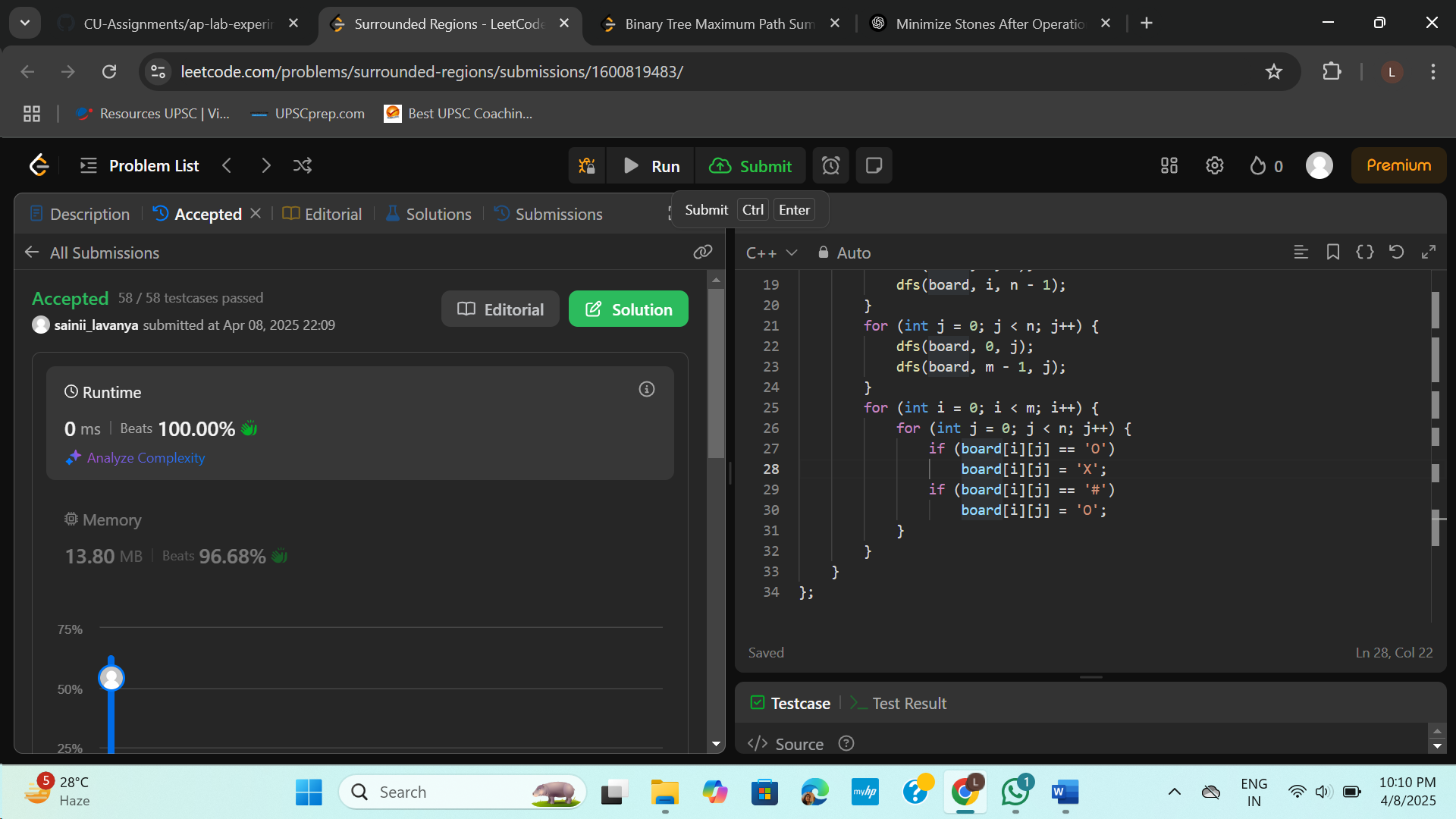
board[i][j] = 'O';

}

}

}

};



1. **Binary Tree Maximum Path Sum**

struct TreeNode {

int val;

TreeNode \*left;

TreeNode \*right;

TreeNode(int x) : val(x), left(NULL), right(NULL) {}

};

class Solution {

public:

int maxSum = INT\_MIN;

int dfs(TreeNode\* root) {

if (!root) return 0;

int left = max(0, dfs(root->left));

int right = max(0, dfs(root->right));

maxSum = max(maxSum, left + right + root->val);

return root->val + max(left, right);

}

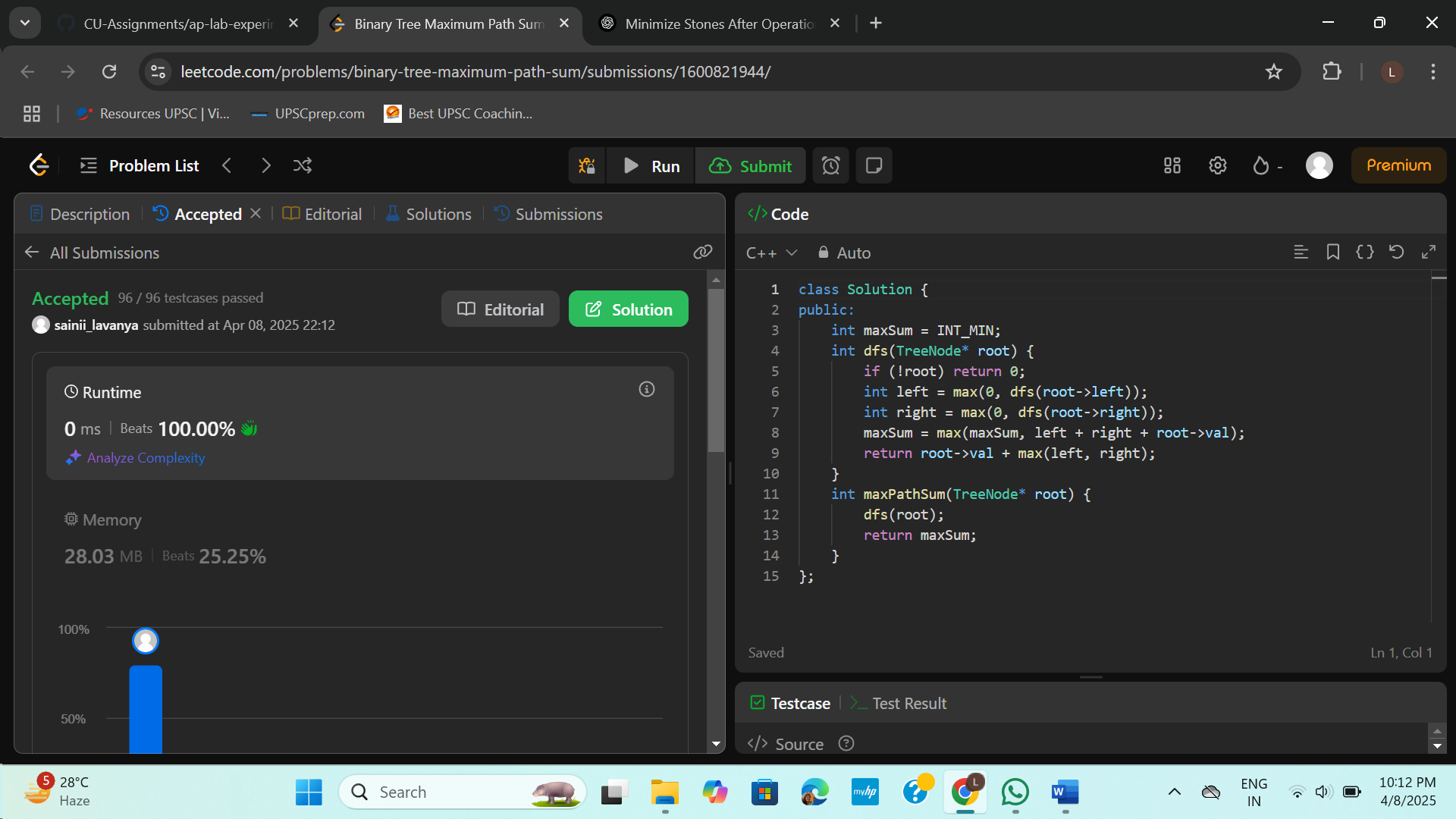
int maxPathSum(TreeNode\* root) {

dfs(root);

return maxSum;

}

};



1. **Number of Provinces**

class Solution {

public:

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

visited[i] = true;

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

if (isConnected[i][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]) {

provinces++;

dfs(isConnected, visited, i);

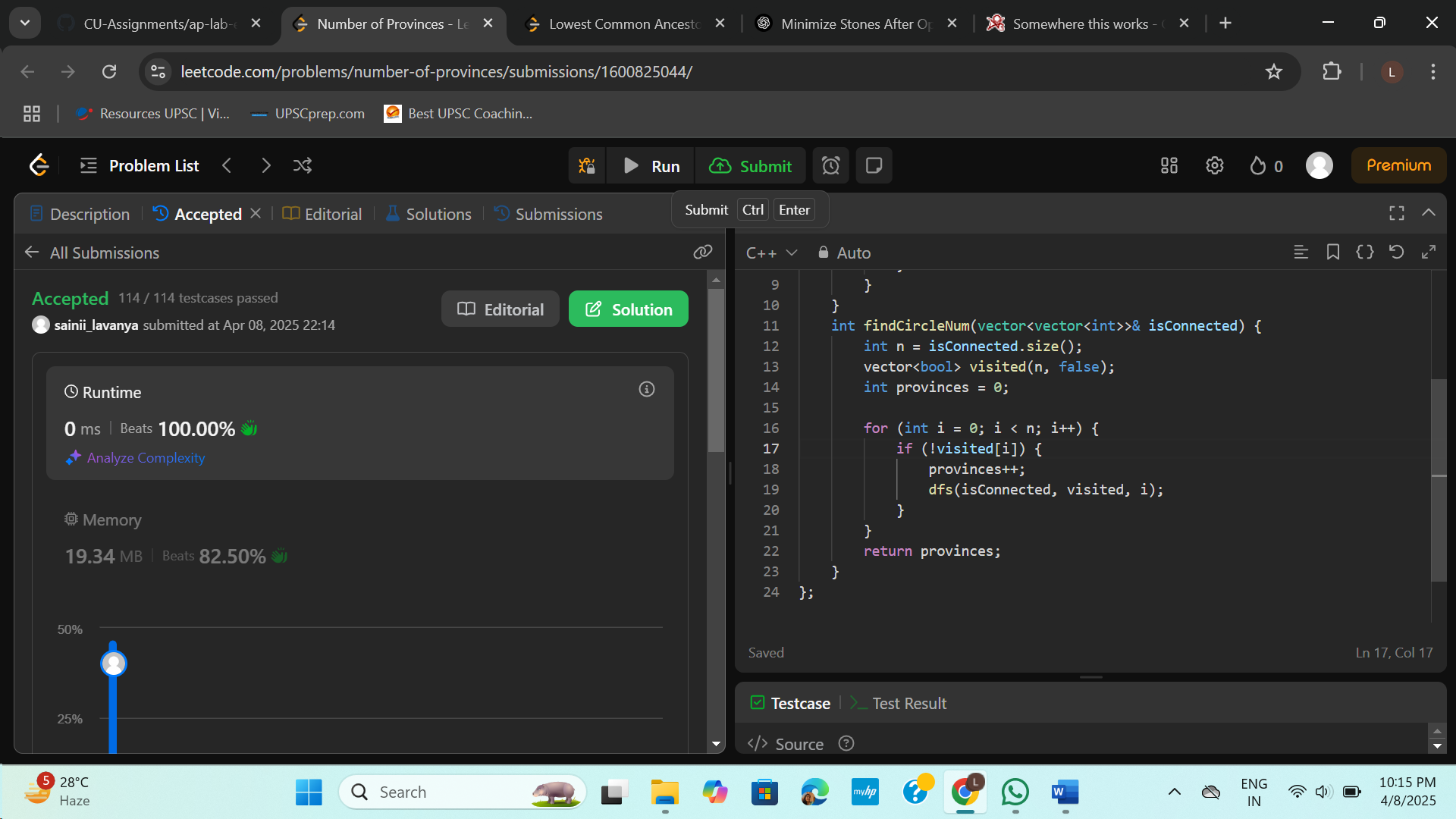
}

}

return provinces;

}

};



1. **Lowest Common Ancestor of a Binary Tree**

class Solution {

public:

TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

if (!root || root == p || root == q) return root;

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

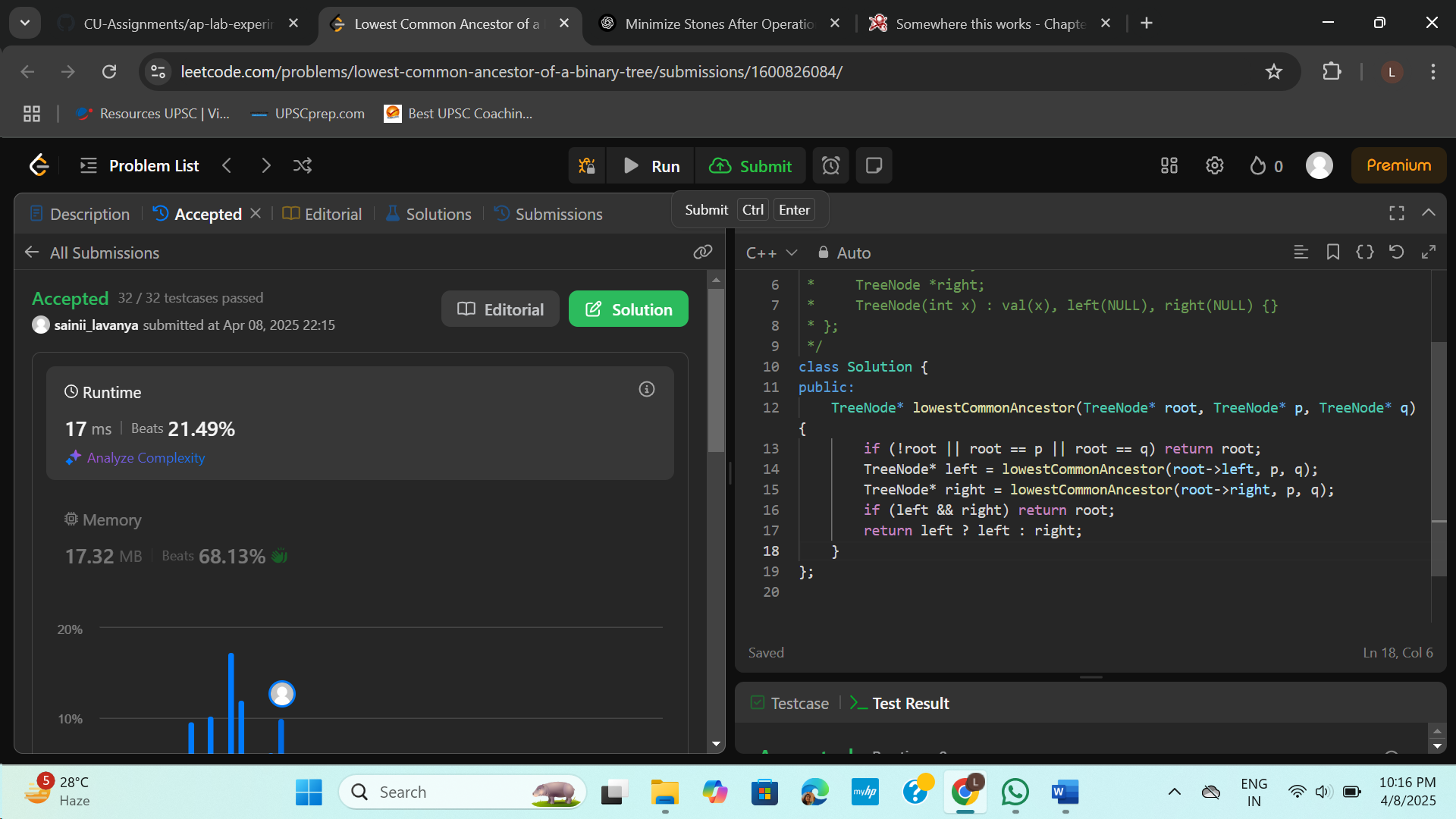
TreeNode\* right = lowestCommonAncestor(root->right, p, q);

if (left && right) return root;

return left ? left : right;

}

};



1. **Course Schedule**

class Solution {

public:

bool dfs(int node, vector<vector<int>>& graph, vector<int>& visited) {

if (visited[node] == 1) return true;

if (visited[node] == 2) return false;

visited[node] = 1;

for (int next : graph[node]) {

if (dfs(next, graph, visited)) return true;

}

visited[node] = 2;

return false;

}

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

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

for (auto& pre : prerequisites) {

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

}

vector<int> visited(numCourses, 0);

for (int i = 0; i < numCourses; i++) {

if (!visited[i]) {

if (dfs(i, graph, visited)) return false;

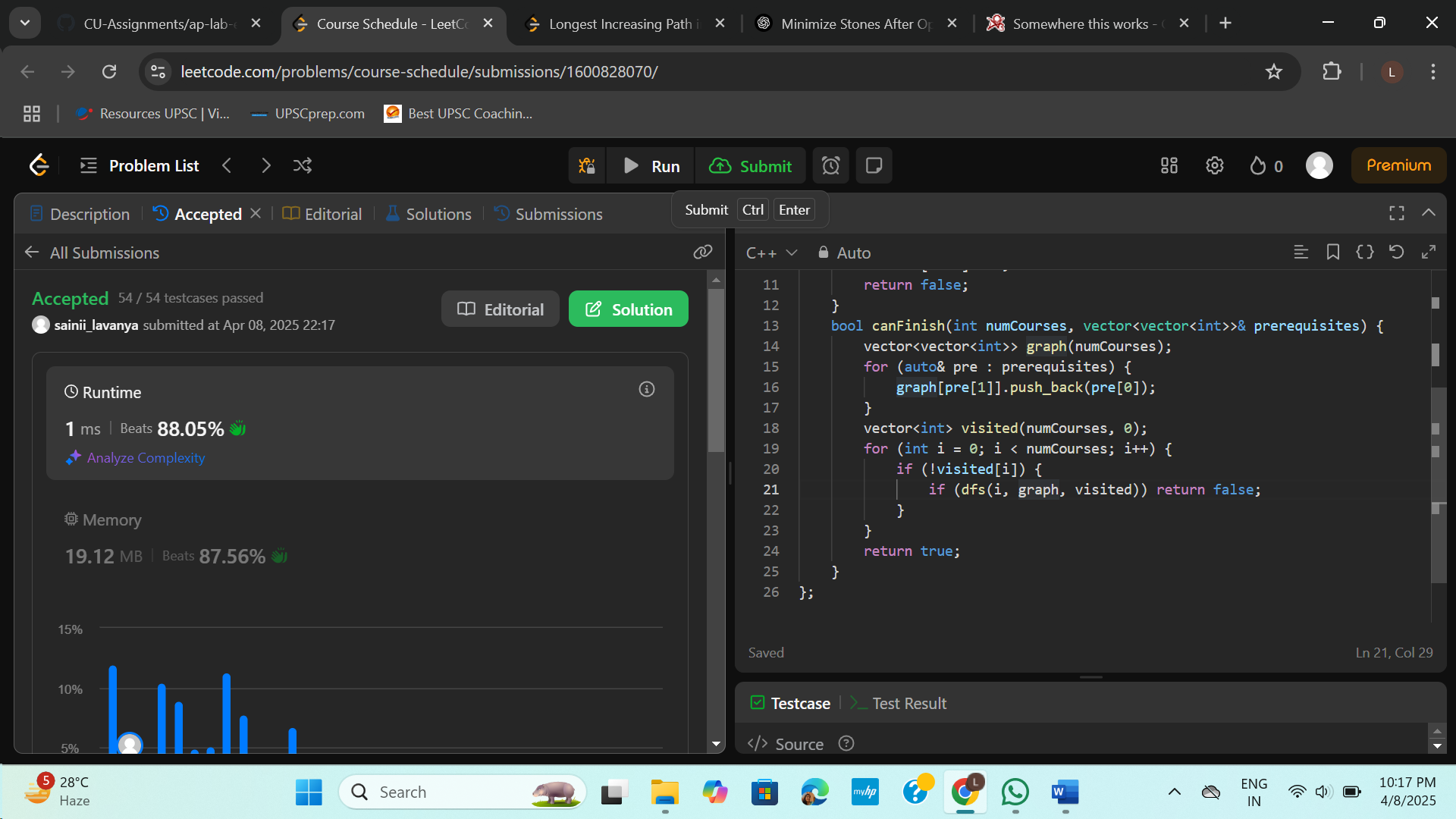
}

}

return true;

}

};



1. **Longest Increasing Path in a Matrix**

class Solution {

public:

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

if (matrix.empty()) return 0;

int m = matrix.size();

int n = matrix[0].size();

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

int maxLen = 0;

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

function<int(int, int)> dfs = [&](int i, int j) -> int {

if (dp[i][j]) return dp[i][j];

int maxPath = 1;

for (int d = 0; d < 4; ++d) {

int ni = i + dirs[d];

int nj = j + dirs[d + 1];

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

maxPath = max(maxPath, 1 + dfs(ni, nj));

}

}

return dp[i][j] = maxPath;

};

for (int i = 0; i < m; ++i) {

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

maxLen = max(maxLen, dfs(i, j));

}

}

return maxLen;

}

};

