**Experiment - 9**

**Name: Armaan UID: 22BCS12007**

**Aim:** To solve leet code problems

1. Problem : Number of Islands

Code:

class Solution {

public int numIslands(char[][] grid) {

int islands = 0;

int rows = grid.length;

int cols = grid[0].length;

Set<String> visited = new HashSet<>();

int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

for (int r = 0; r < rows; r++) {

for (int c = 0; c < cols; c++) {

if (grid[r][c] == '1' && !visited.contains(r + "," + c)) {

islands++;

bfs(grid, r, c, visited, directions, rows, cols);

}

}

}

return islands;

}

private void bfs(char[][] grid, int r, int c, Set<String> visited, int[][] directions, int rows, int cols) {

Queue<int[]> q = new LinkedList<>();

visited.add(r + "," + c);

q.add(new int[]{r, c});

while (!q.isEmpty()) {

int[] point = q.poll();

int row = point[0], col = point[1];

for (int[] direction : directions) {

int nr = row + direction[0], nc = col + direction[1];

if (nr >= 0 && nr < rows && nc >= 0 && nc < cols && grid[nr][nc] == '1' && !visited.contains(nr + "," + nc)) {

q.add(new int[]{nr, nc});

visited.add(nr + "," + nc);

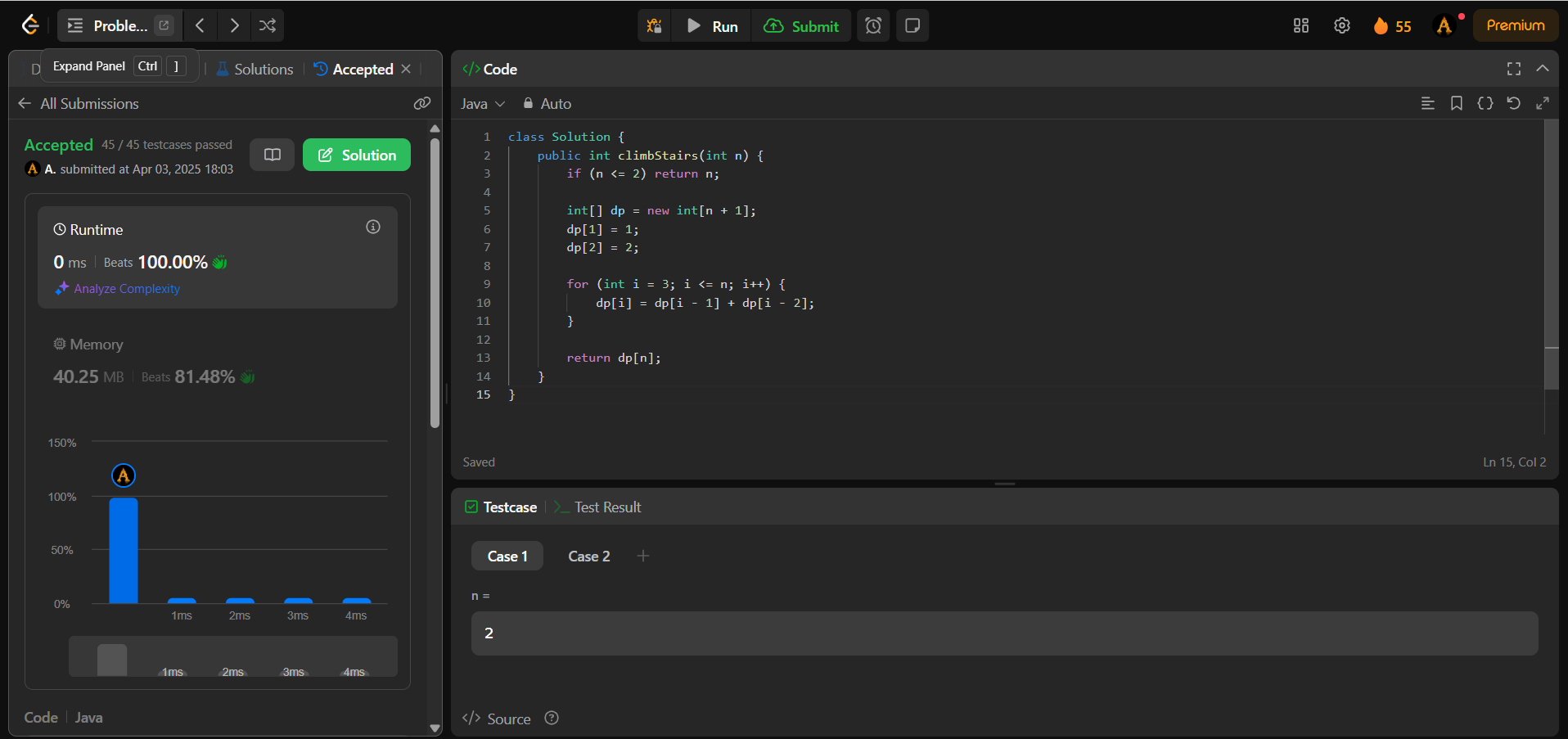
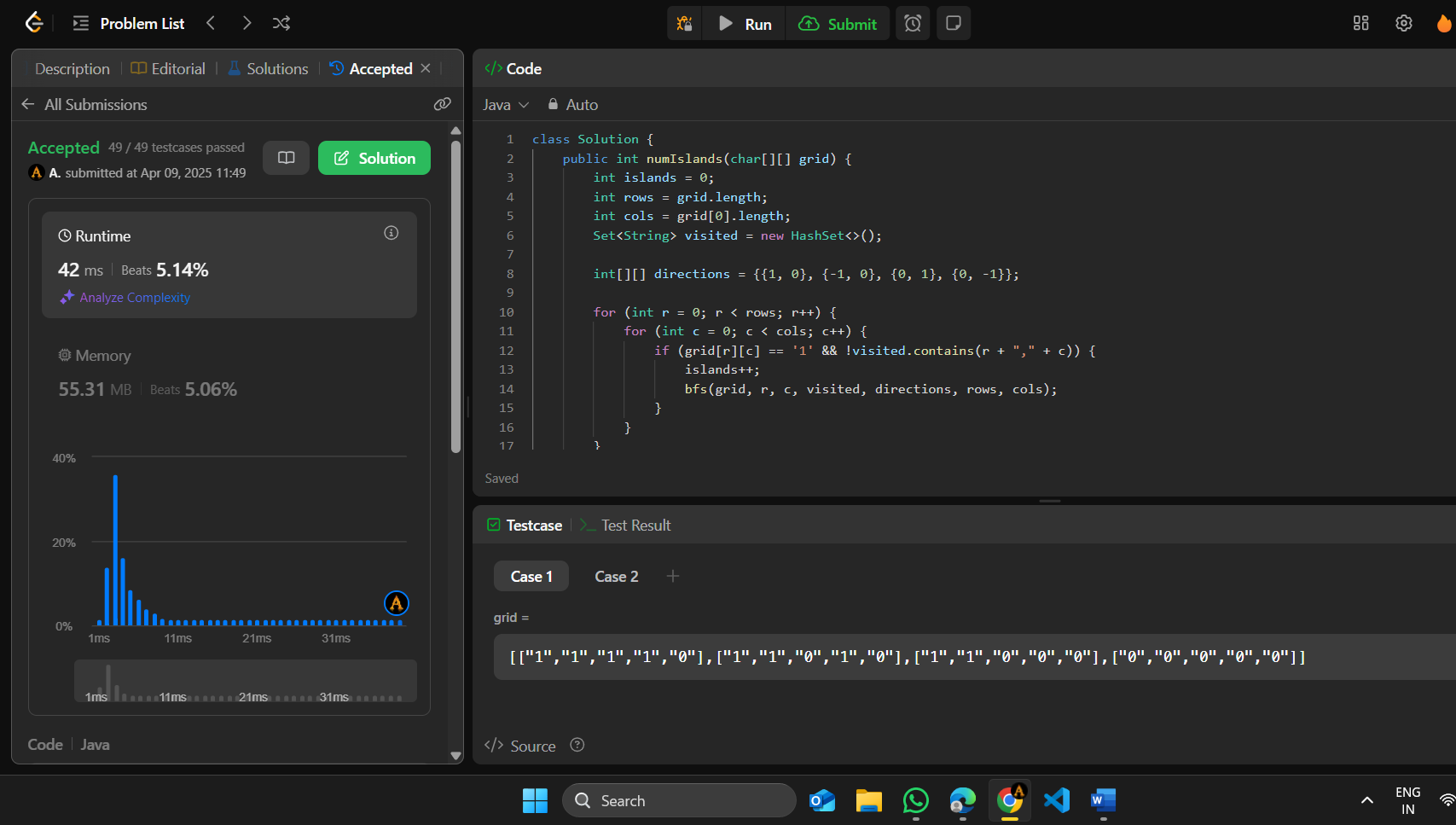
}

}

}

}

}

Output :

1. Problem: Word Ladder

Code:

class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

Set<String> set = new HashSet<>(wordList);

if(!set.contains(endWord)) return 0;

Queue<String> queue = new LinkedList<>();

queue.add(beginWord);

Set<String> visited = new HashSet<>();

queue.add(beginWord);

int changes = 1;

while(!queue.isEmpty()){

int size = queue.size();

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

String word = queue.poll();

if(word.equals(endWord)) return changes;

for(int j = 0; j < word.length(); j++){

for(int k = 'a'; k <= 'z'; k++){

char arr[] = word.toCharArray();

arr[j] = (char) k;

String str = new String(arr);

if(set.contains(str) && !visited.contains(str)){

queue.add(str);

visited.add(str);

}

}

}

}

++changes;

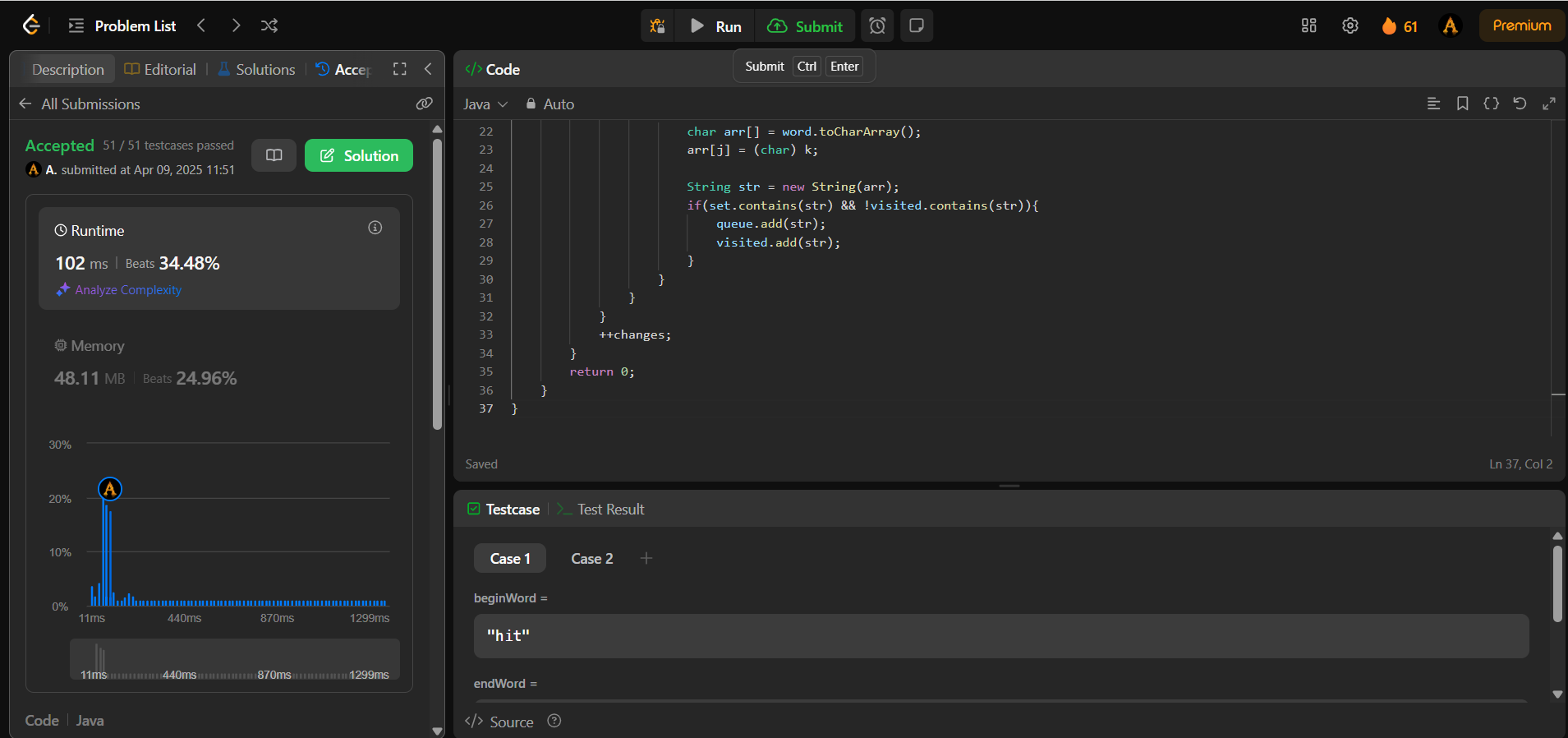
}

return 0;

}

}

Output:



1. Problem: Surrounded Regions

Code:

class Solution {

class Pair {

int first;

int second;

Pair(int first, int second) {

this.first = first;

this.second = second;

}

}

public void solve(char[][] board) {

int n = board[0].length;

int m = board.length;

int visited[][] = new int[m][n];

// i would like to start dfs traversal from boundry if getting O and mark

// visited, with this tech be visited all vertex of graph which can not be crossed(X).

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

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

if (i == 0 && board[i][j] == 'O' && visited[i][j] == 0) {

DFS(board, i, j, visited);

}

if (i == m - 1 && board[i][j] == 'O' && visited[i][j] == 0) {

DFS(board, i, j, visited);

}

if (j == 0 && board[i][j] == 'O' && visited[i][j] == 0) {

DFS(board, i, j, visited);

}

if (j == n - 1 && board[i][j] == 'O' && visited[i][j] == 0) {

DFS(board, i, j, visited);

}

}

}

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

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

if (board[i][j] == 'O' && visited[i][j] == 0) {

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

}

}

}

}

public void DFS(char board[][], int i, int j, int visited[][]) {

int n = board[0].length;

int m = board.length;

Stack<Pair> st = new Stack<>();

st.push(new Pair(i, j));

while (!st.isEmpty()) {

Pair node = st.pop();

int first = node.first;

int second = node.second;

visited[first][second] = 1;

if (second < n - 1 && board[first][second + 1] == 'O' && visited[first][second + 1] == 0) {

st.push(new Pair(first, second + 1));

visited[first][second + 1] = 1;

}

// Left

if (second > 0 && board[first][second - 1] == 'O' && visited[first][second - 1] == 0) {

st.push(new Pair(first, second - 1));

visited[first][second - 1] = 1;

}

// Down

if (first < m - 1 && board[first + 1][second] == 'O' && visited[first + 1][second] == 0) {

st.push(new Pair(first + 1, second));

visited[first + 1][second] = 1;

}

// Up

if (first > 0 && board[first - 1][second] == 'O' && visited[first - 1][second] == 0) {

st.push(new Pair(first - 1, second));

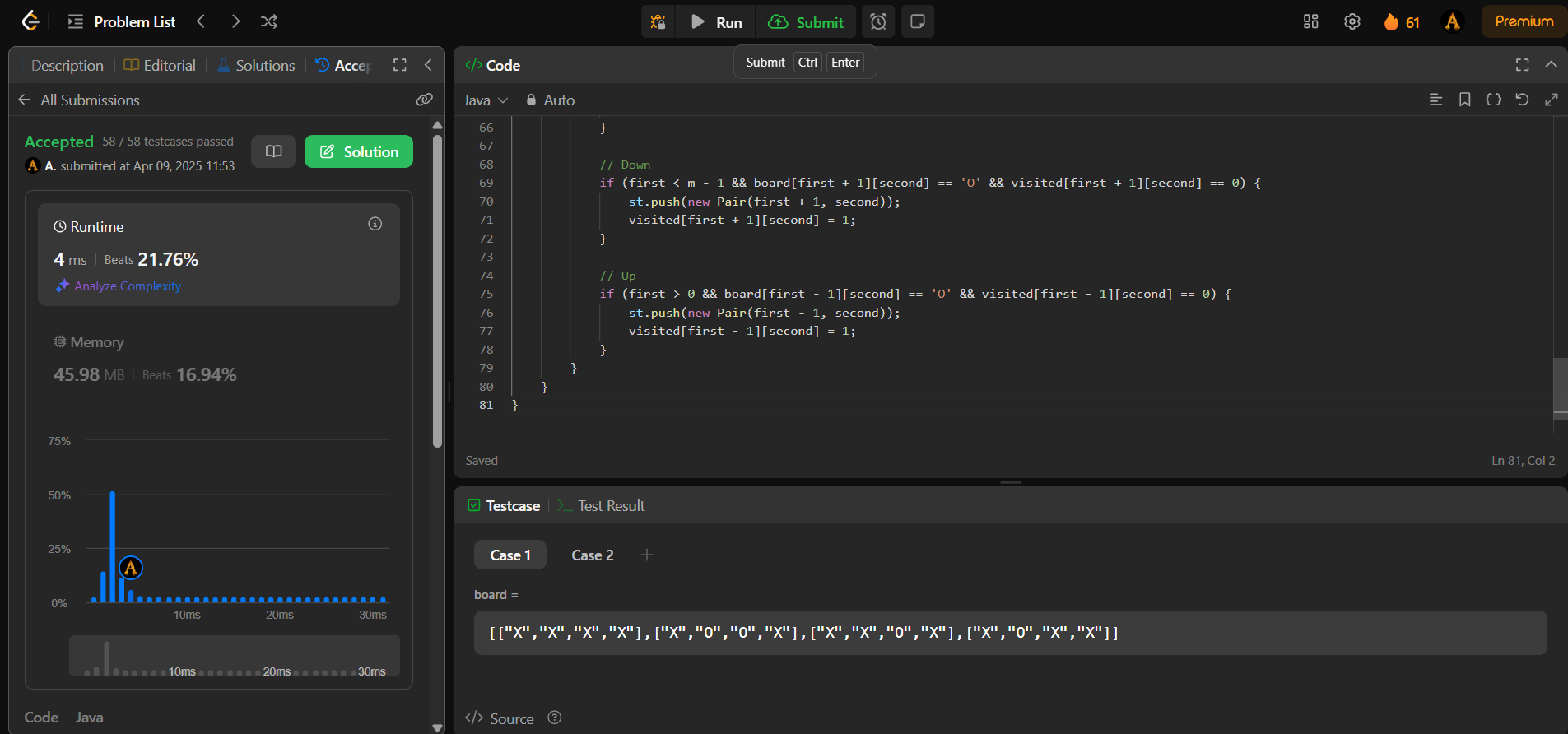
visited[first - 1][second] = 1;

}

}

}

}

Output:

1. Problem: Binary Tree Maximum Path Sum

Code:

class Solution {

public int maxPathSum(TreeNode root) {

int[] res = { root.val };

dfs(root, res);

return res[0];

}

private int dfs(TreeNode node, int[] res) {

if (node == null) {

return 0;

}

// Recursively compute the maximum sum of the left and right subtree paths.

int leftSum = Math.max(0, dfs(node.left, res));

int rightSum = Math.max(0, dfs(node.right, res));

// Update the maximum path sum encountered so far (with split).

res[0] = Math.max(res[0], leftSum + rightSum + node.val);

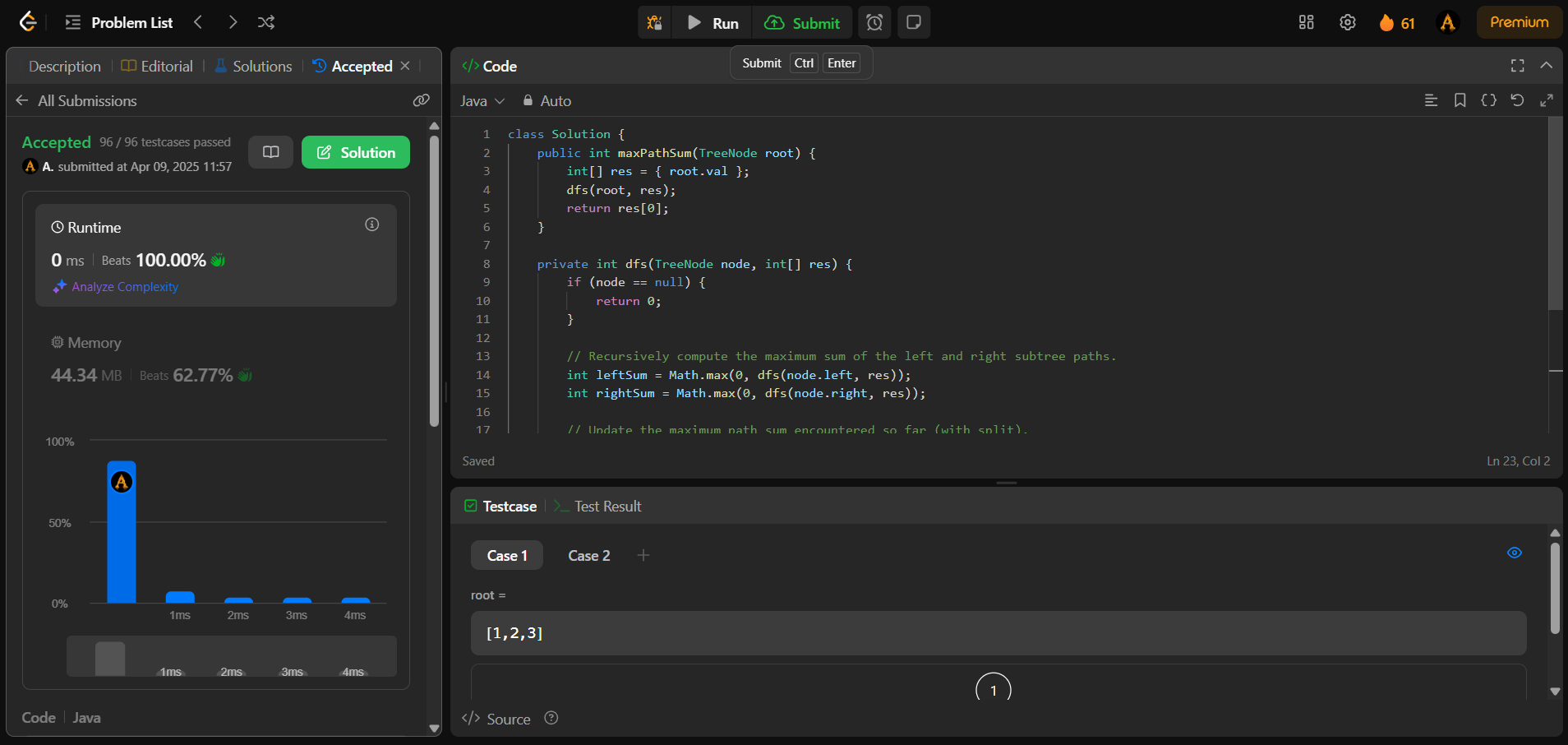
// Return the maximum sum of the path (without split).

return Math.max(leftSum, rightSum) + node.val;

}

}

Output:



1. Problem: Friend Circles

Code:

class Solution {

boolean visited[];

public int findCircleNum(int[][] isConnected) {

visited=new boolean[isConnected.length];

int cnt=0;

for(int i=0;i<visited.length;i++)

{

if(!visited[i])

{

dfs(isConnected, i);

cnt++;

}

}

return cnt;

}

private void dfs(int[][] isConnected, int curr)

{

visited[curr]=true;

for(int i=0;i<isConnected[curr].length;i++)

{

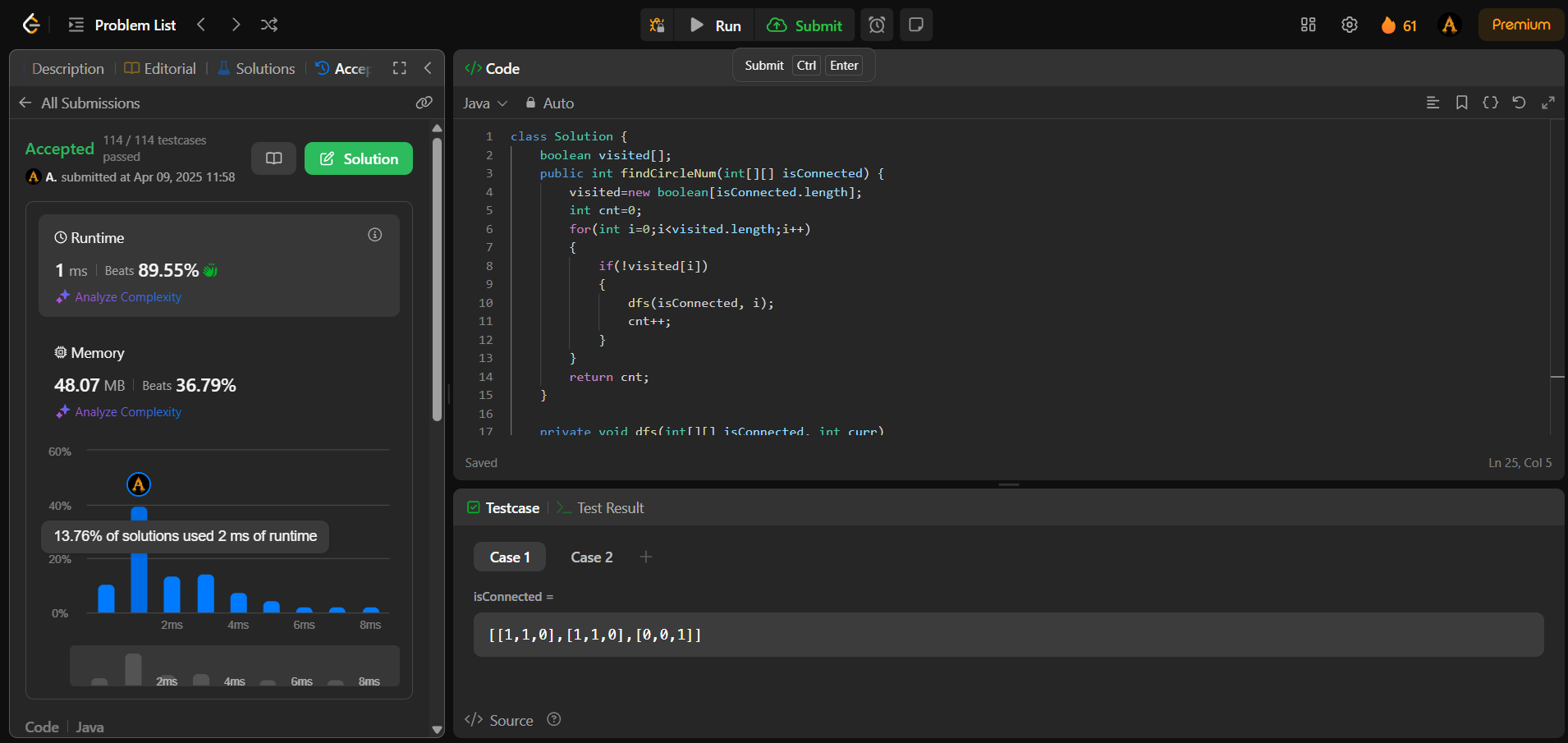
if(isConnected[curr][i]==1 && !visited[i]) dfs(isConnected, i);

}

}

}

Output:



1. Problem : Lowest common ancestor of a binary tree

Code:

class Solution {

    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

        if (root == null || root == p || root == q) {

            return root;

        }

        TreeNode left = lowestCommonAncestor(root.left, p, q);

        TreeNode right = lowestCommonAncestor(root.right, p, q);

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

            return root; // p and q are found in different subtrees, so root is LCA

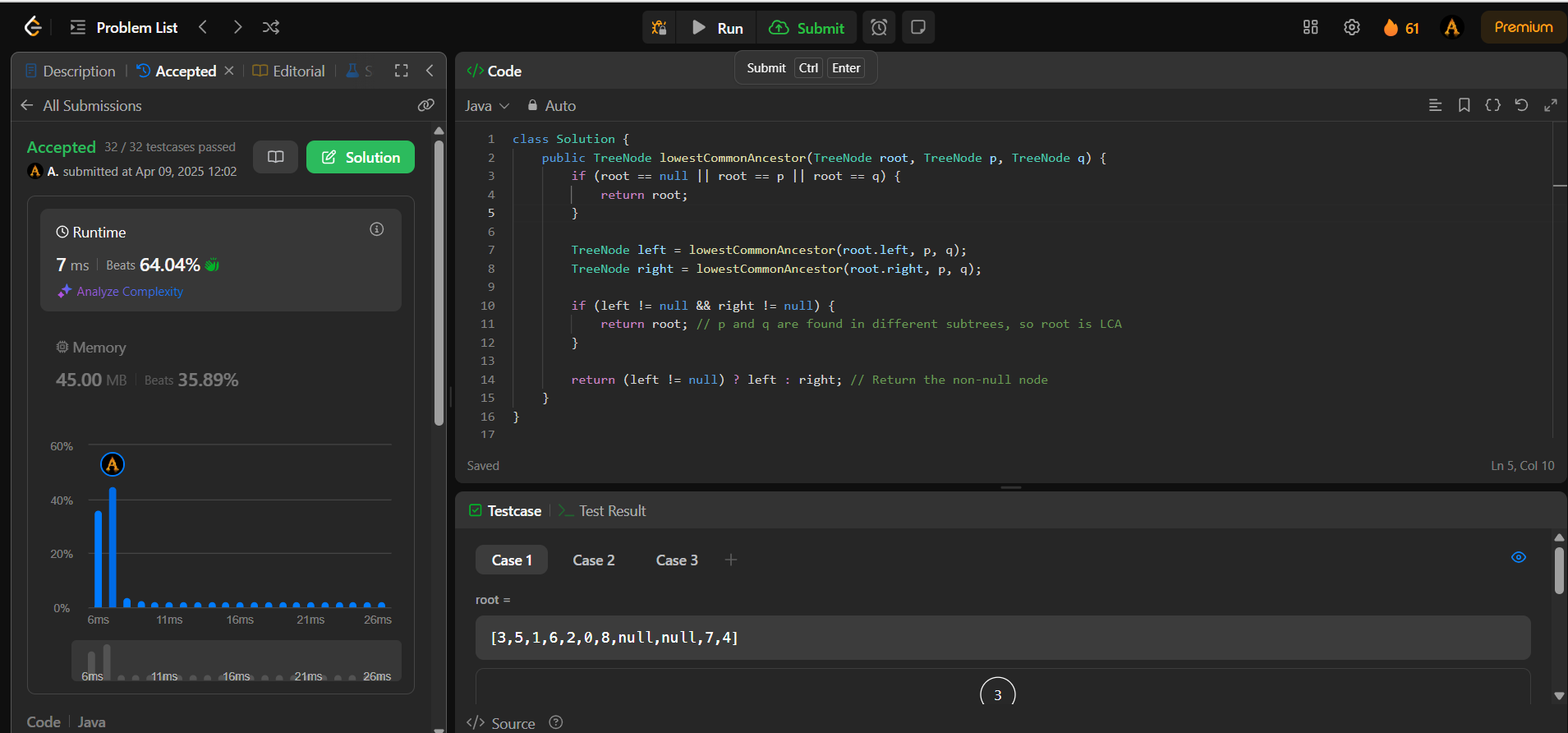
        }

        return (left != null) ? left : right; // Return the non-null node

    }

}

Output:



1. Problem: Course Schedule

Code:

class Solution {

public boolean dfs(ArrayList<Integer>[] edges, int[] state, int in) {

if (state[in] == 1) return false;

if (state[in] == 2) return true;

state[in] = 1;

for (int i : edges[in]) {

if (!dfs(edges, state, i)) {

return false;

}

}

state[in] = 2;

return true;

}

public boolean canFinish(int n, int[][] pr) {

ArrayList<Integer>[] edges = new ArrayList[n];

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

edges[i] = new ArrayList<>();

}

for (int[] i : pr) {

edges[i[1]].add(i[0]);

}

int[] state = new int[n];

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

if (state[i] == 0) {

if (!dfs(edges, state, i)) {

return false;

}

}

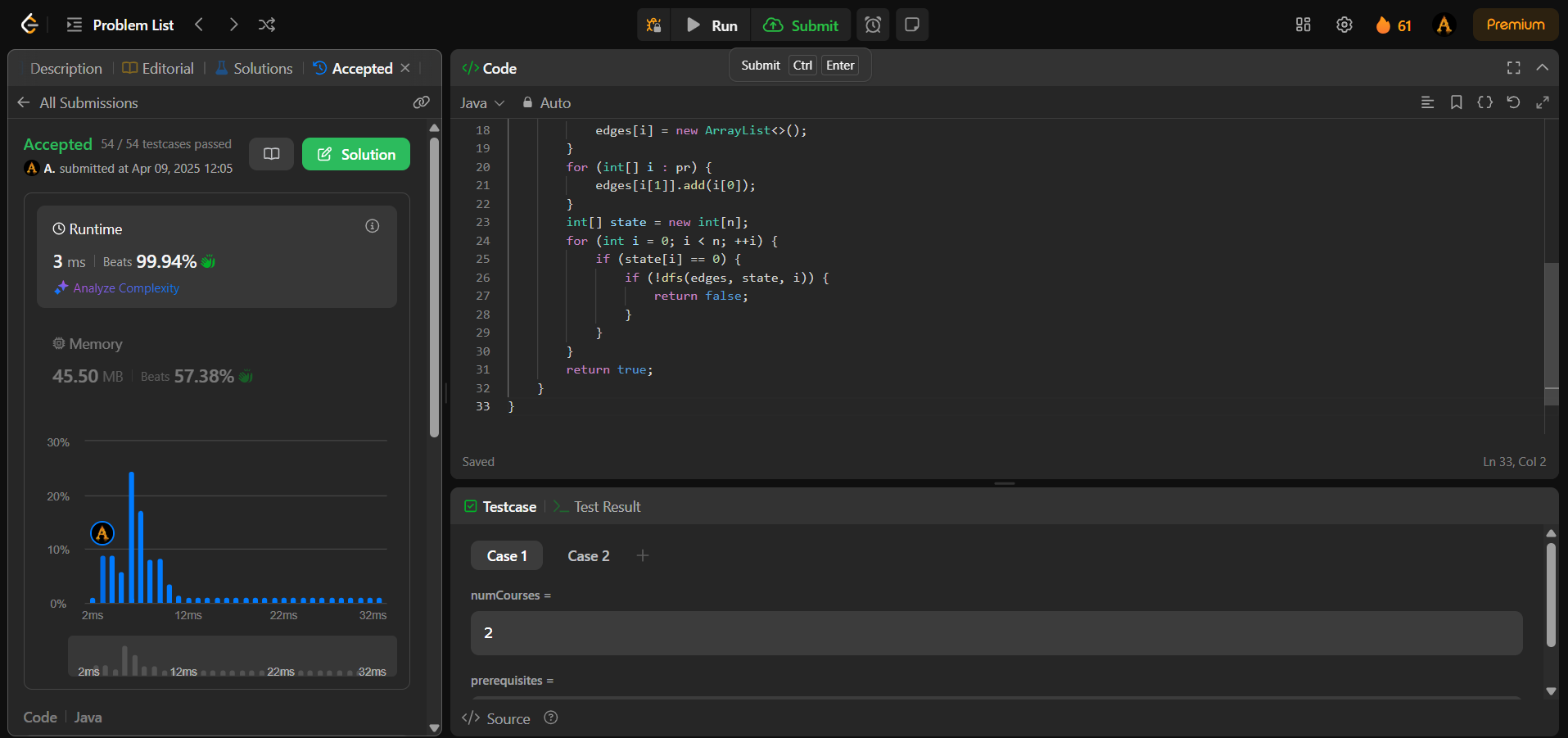
}

return true;

}

}

Output:



1. Problem: Longest Increasing Path in a Matrix

Code:

class Solution {

public int longestIncreasingPath(int[][] M) {

int ylen = M.length, xlen = M[0].length, ans = 0;

int[][] memo = new int[ylen][xlen];

for (int i = 0; i < ylen; i++)

for (int j = 0; j < xlen; j++)

ans = Math.max(ans, dfs(i,j,M,memo));

return ans;

}

public int dfs(int y, int x, int[][] M, int[][] memo) {

if (memo[y][x] > 0) return memo[y][x];

int val = M[y][x];

memo[y][x] = 1 + Math.max(

Math.max(y < M.length - 1 && M[y+1][x] < val ? dfs(y+1,x,M,memo) : 0,

y > 0 && M[y-1][x] < val ? dfs(y-1,x,M,memo) : 0),

Math.max(x < M[0].length - 1 && M[y][x+1] < val ? dfs(y,x+1,M,memo) : 0,

x > 0 && M[y][x-1] < val ? dfs(y,x-1,M,memo) : 0));

return memo[y][x];

}

}

Output:

