**Experiment 6**

**Student Name:** Keshav Mangla **UID:** 22BET10218

**Branch:** BE -IT **Section/Group:**22BET\_IOT-702/A

**Semester:** 6th **Date of Performance:**28/2/2025 **Subject Name:** Advanced Programming Lab-2 **Subject Code:** 22ITP-351

# Symmetric Tree Aim :

To determine whether a given binary tree is symmetric around its center.

# Objectives :

* Understand recursive and iterative methods to check tree symmetry.
* Explore the concept of mirroring subtrees.
* Analyze time and space complexity for symmetry checks.

**Implementation/Code :**

class Solution {

public boolean isSymmetric(TreeNode root) {

if (root == null) return true; // An empty tree is symmetric return isMirror(root.left, root.right);

}

private boolean isMirror(TreeNode t1, TreeNode t2) {

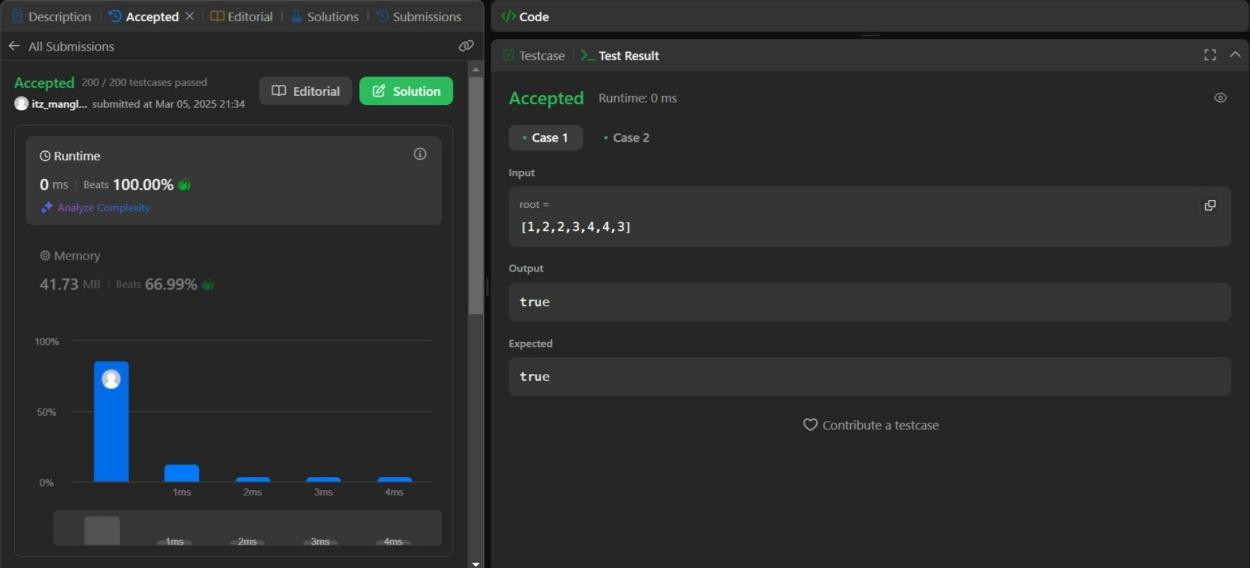
if (t1 == null && t2 == null) return true; // Both nodes are null, symmetric if (t1 == null || t2 == null) return false; // One node is null, not symmetric return (t1.val == t2.val) // Compare node values

&& isMirror(t1.left, t2.right) // Compare outer pairs && isMirror(t1.right, t2.left); // Compare inner pairs

}

}

**Output :**



# Kth Smallest Element in a BST Aim :

To find the k-th smallest element in a Binary Search Tree (BST).

# Objective :

* Understand in-order traversal of a BST.
* Learn how BST properties help in ordered node retrieval.
* Implement both recursive and iterative solutions.

# Implementation/Code :

class Solution {

private int count = 0; private int result = -1;

public int kthSmallest(TreeNode root, int k) { inorderTraversal(root, k);

return result;

}

private void inorderTraversal(TreeNode node, int k) { if (node == null) {

return;

}

// Traverse the left subtree

inorderTraversal(node.left, k);

// Visit the node count++;

if (count == k) { result = node.val; return;

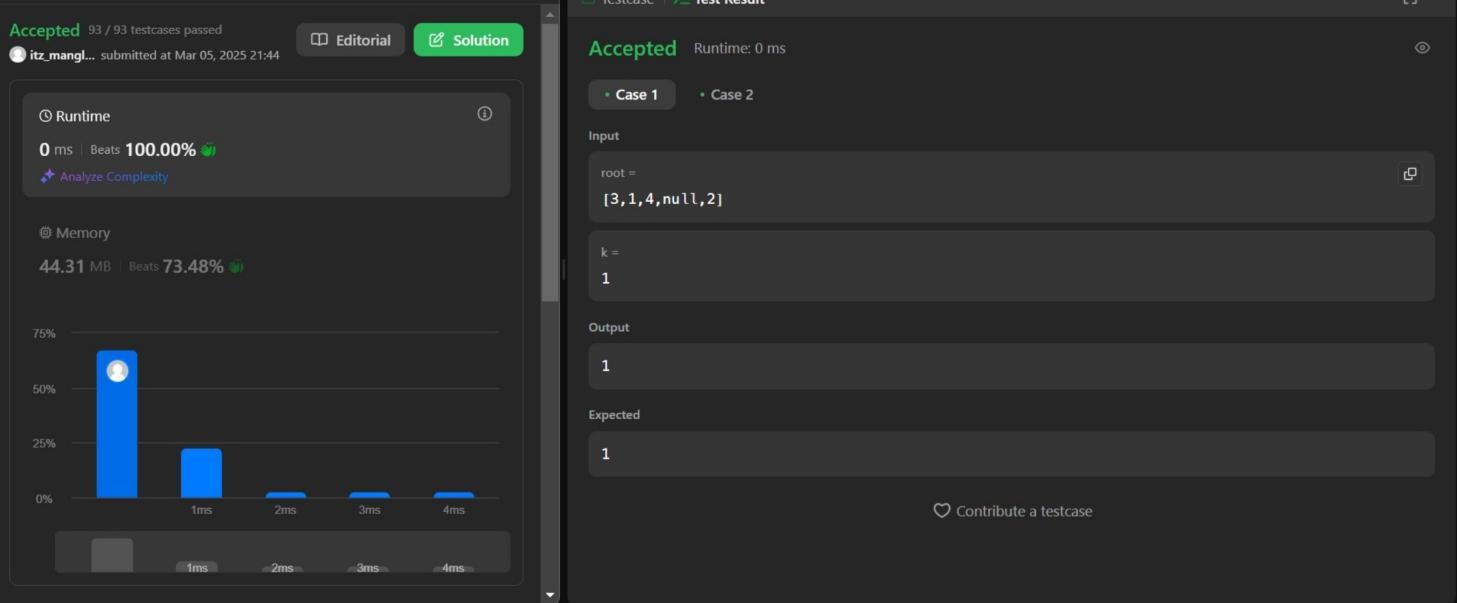
}

// Traverse the right subtree inorderTraversal(node.right, k);

}

}

# Output :

****

**Populating Next Right Pointers in Each Node Aim :**

To link each node in a perfect binary tree to its next right node.

# Objective :

* Understand level-order traversal and its applications.
* Explore pointer manipulation techniques for tree traversal.
* Implement both recursive and iterative approaches.

# Implementation/Code :

class Solution {

public Node connect(Node root) { if (root == null) {

return null;

}

Node leftmost = root; // start with the root node while (leftmost.left != null) { // move level by level

Node current = leftmost;

while (current != null) {

// connect left child to right child current.left.next = current.right;

// connect right child to next node's left child if (current.next != null) {

current.right.next = current.next.left;

}

// move to next node on the same level current = current.next;

}

// move to the next level leftmost = leftmost.left;

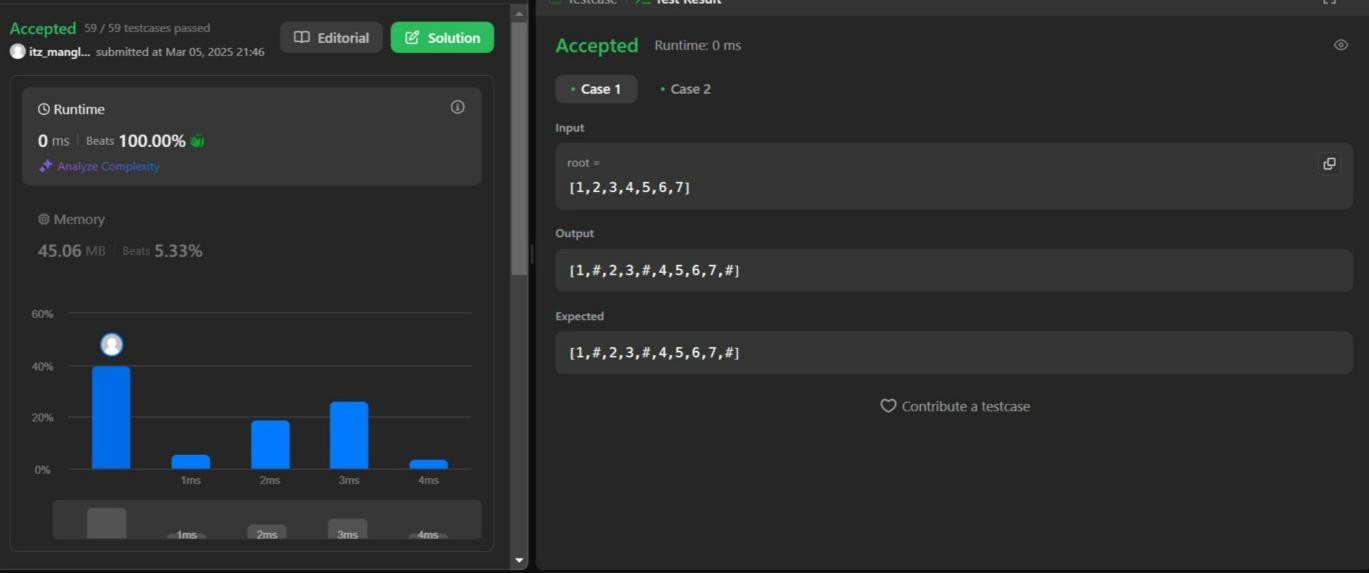
}

return root;

}

}

# Output :

****

**Maximum Depth of a Binary Tree Aim :**

To calculate the maximum depth of a binary tree.

# Objective :

* Understand the concept of tree height and depth.
* Explore recursive depth-first search (DFS) and breadth-first search (BFS).
* Implement algorithms to compute tree depth efficiently.

# Implementation/Code :

class Solution {

public int maxDepth(TreeNode root) { if (root == null) {

return 0; // base case: if the tree is empty, depth is 0

}

int leftDepth = maxDepth(root.left); // find depth of left subtree

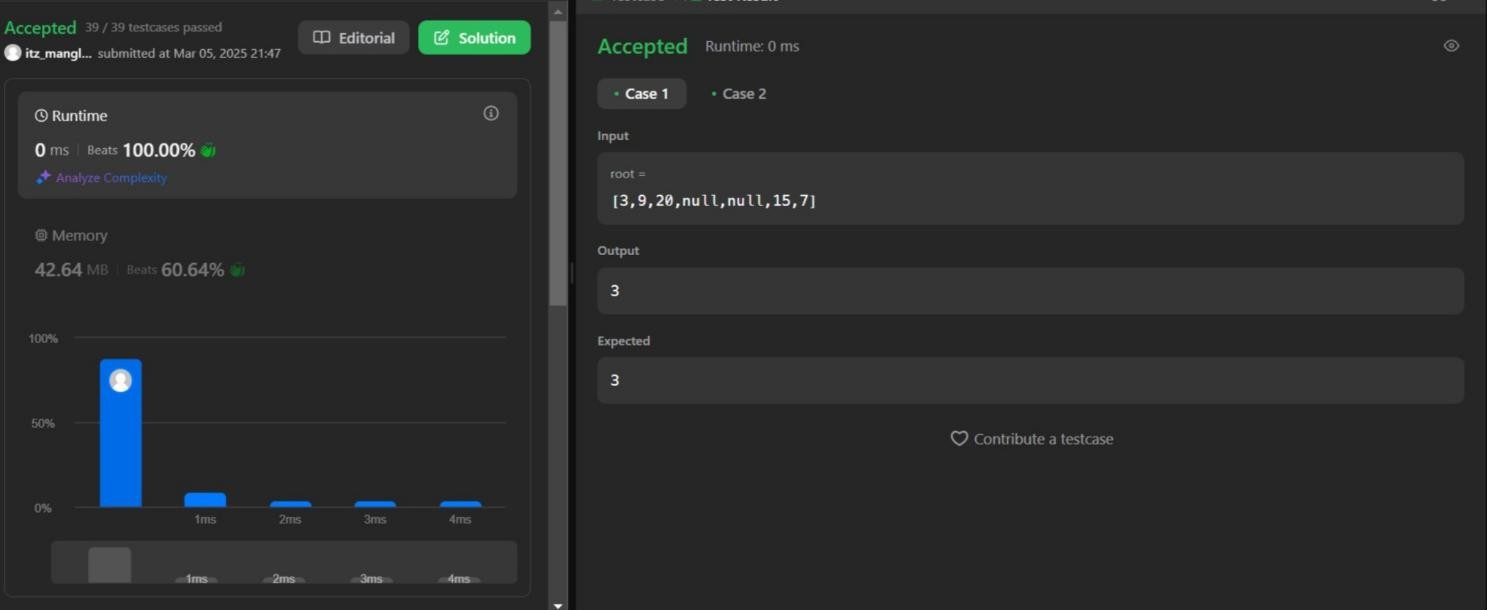
int rightDepth = maxDepth(root.right); // find depth of right subtree

return Math.max(leftDepth, rightDepth) + 1; // add 1 for the current node

}

}

# Output :

****

**Validate Binary Search Tree (BST) Aim :**

To check whether a given binary tree is a valid BST.

# Objectives :

* Understand the properties of binary search trees.
* Implement recursive algorithms with boundary conditions for BST validation.
* Explore in-order traversal for BST property checks.

# Implementation/Code :

class Solution {

public boolean isValidBST(TreeNode root) {

return validate(root, Long.MIN\_VALUE, Long.MAX\_VALUE);

}

private boolean validate(TreeNode node, long min, long max) { if (node == null) {

return true; // An empty tree is a valid BST

}

if (node.val <= min || node.val >= max) { return false; // violates BST rules

}

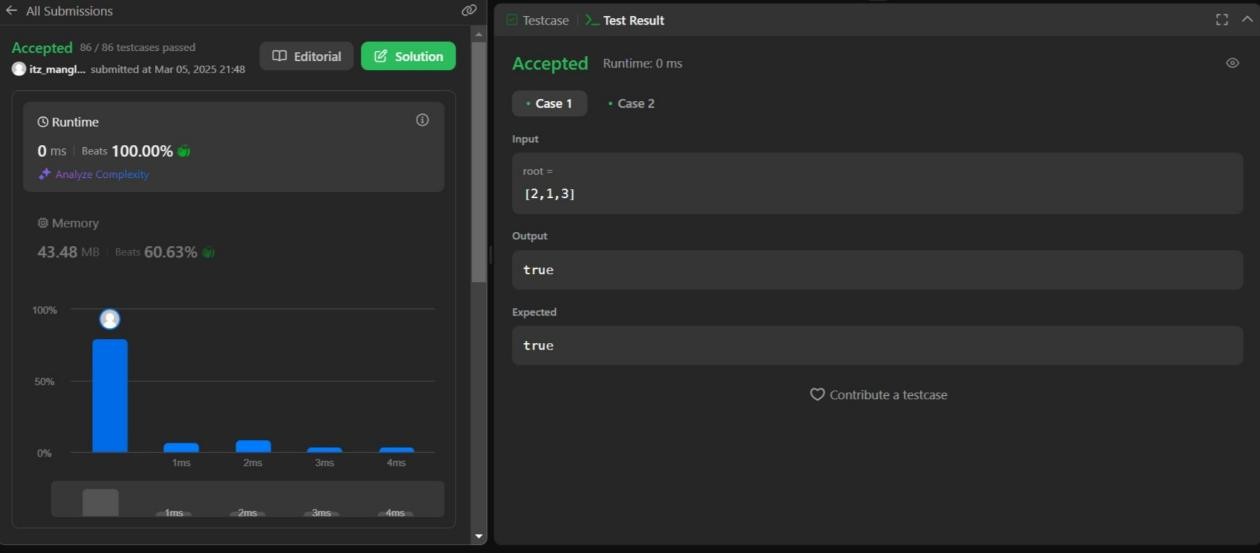
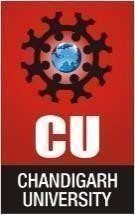
// Recursively check the left and right subtree with updated boundaries

return validate(node.left, min, node.val) && validate(node.right, node.val, max);

}

}

# Output :



& ENGINEERING



COMPUTER SCIENCE

**Binary Tree Level Order Traversal**

**Aim :**

To implement a program that performs level order traversal of a binary tree.

**Objectives :**

* Understand the concept of binary tree traversal.
* Implement level order traversal using queues.
* Analyze the time and space complexity of the traversal algorithm.

**Code :**

import java.util.\*;

// Definition for a binary tree node.

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) { this.val = val; }

TreeNode(int val, TreeNode left, TreeNode right) {

this.val = val;

this.left = left;

this.right = right;

}

}

public class Solution {

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> result = new ArrayList<>();

if (root == null) return result;

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

queue.offer(root);

while (!queue.isEmpty()) {

int levelSize = queue.size();

List<Integer> currentLevel = new ArrayList<>();

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

TreeNode currentNode = queue.poll();

currentLevel.add(currentNode.val);

if (currentNode.left != null) queue.offer(currentNode.left);

if (currentNode.right != null) queue.offer(currentNode.right);

}

result.add(currentLevel);

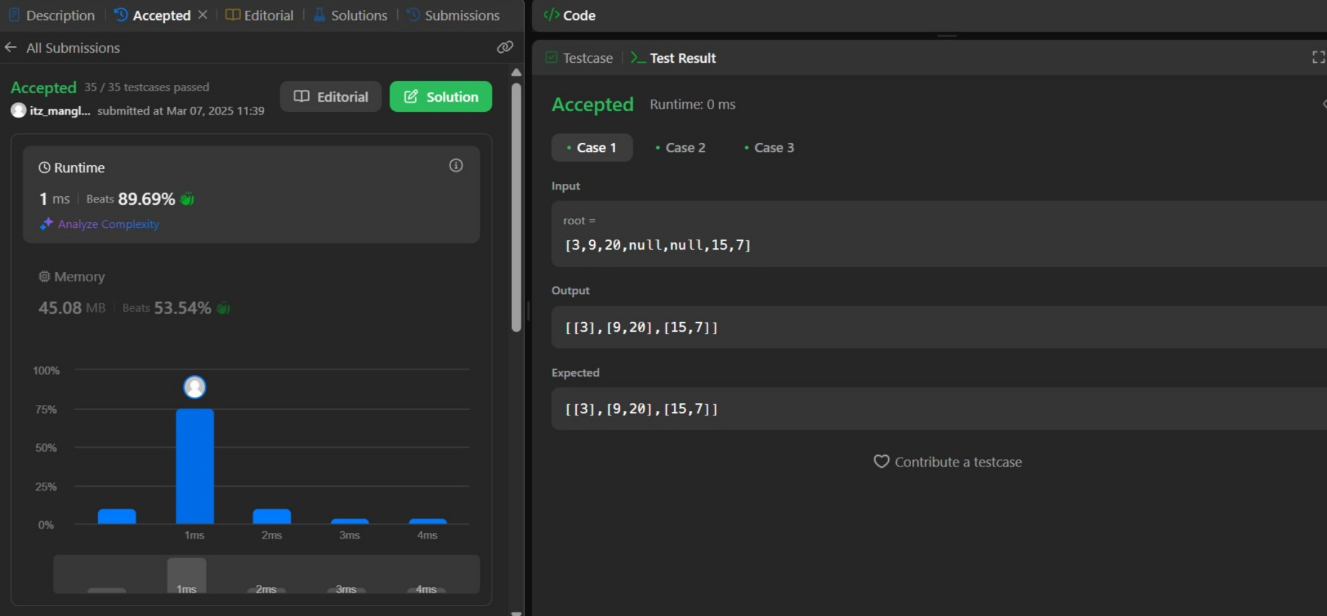
}

return result;

}

}

**Output:**

****

**Convert Sorted Array to Binary Search Tree**

**Aim :**

To implement a program that converts a sorted array into a height-balanced binary search tree (BST).

**Objectives :**

* Understand the properties of a BST.
* Implement a recursive approach to convert a sorted array into a BST.
* Analyze the time and space complexity of the conversion.

**Code :**

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) { this.val = val; }

TreeNode(int val, TreeNode left, TreeNode right) {

this.val = val;

this.left = left;

this.right = right;

}

}

public class Solution {

public TreeNode sortedArrayToBST(int[] nums) {

if (nums == null || nums.length == 0) return null;

return helper(nums, 0, nums.length - 1);

}

private TreeNode helper(int[] nums, int left, int right) {

if (left > right) return null;

int mid = left + (right - left) / 2;

TreeNode node = new TreeNode(nums[mid]);

node.left = helper(nums, left, mid - 1);

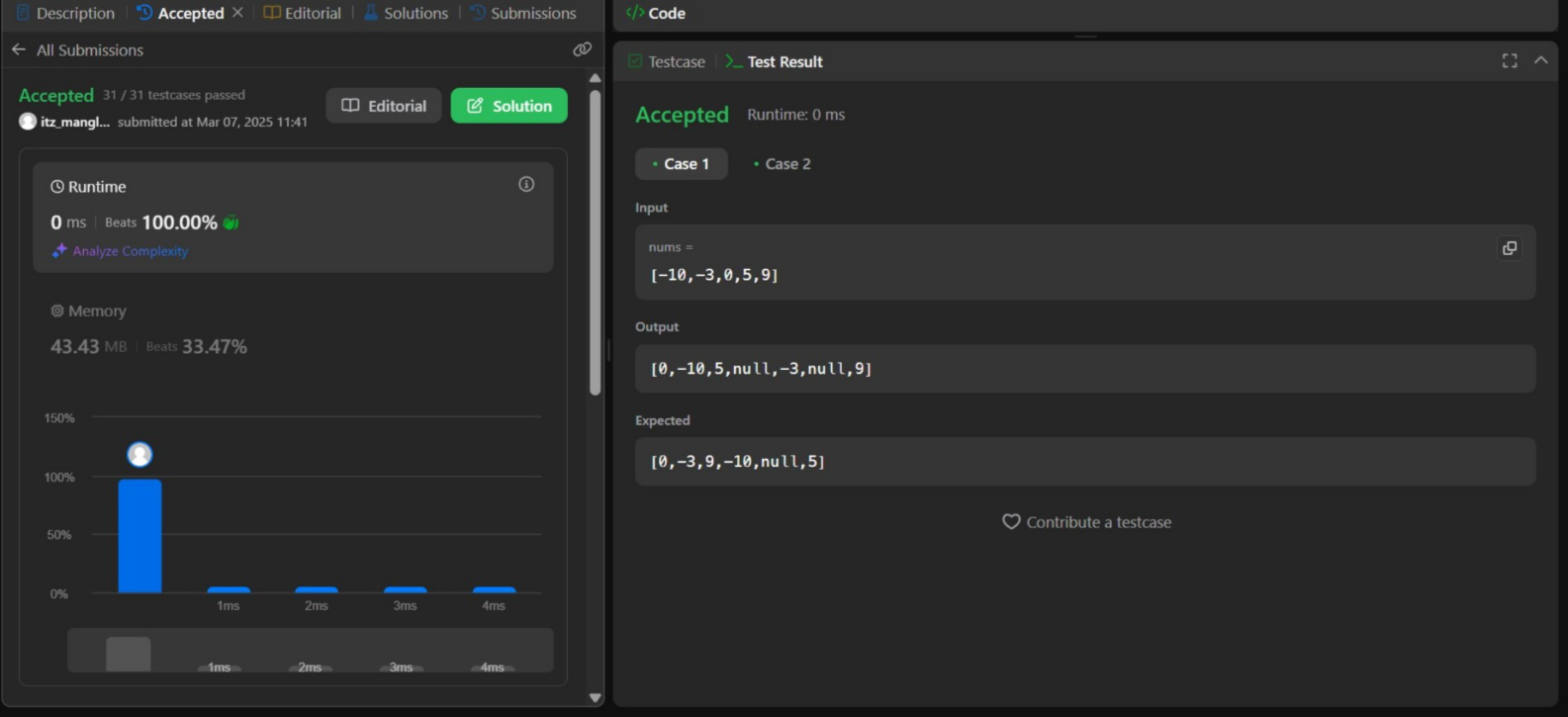
node.right = helper(nums, mid + 1, right);

return node;

}

}

**Output :**

****

**Binary Tree Inorder Traversal**

**Aim :**

To implement a program that performs inorder traversal of a binary tree.

**Objectives :**

* Understand the concept of binary tree traversal.
* Implement inorder traversal using recursion and iteration.
* Analyze the time and space complexity of inorder traversal.

**Code :**

import java.util.\*;

// Definition for a binary tree node.

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) { this.val = val; }

TreeNode(int val, TreeNode left, TreeNode right) {

this.val = val;

this.left = left;

this.right = right;

}

}

public class Solution {

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<>();

inorderHelper(root, result);

return result;

}

private void inorderHelper(TreeNode node, List<Integer> result) {

if (node == null) return;

inorderHelper(node.left, result); // Traverse left subtree

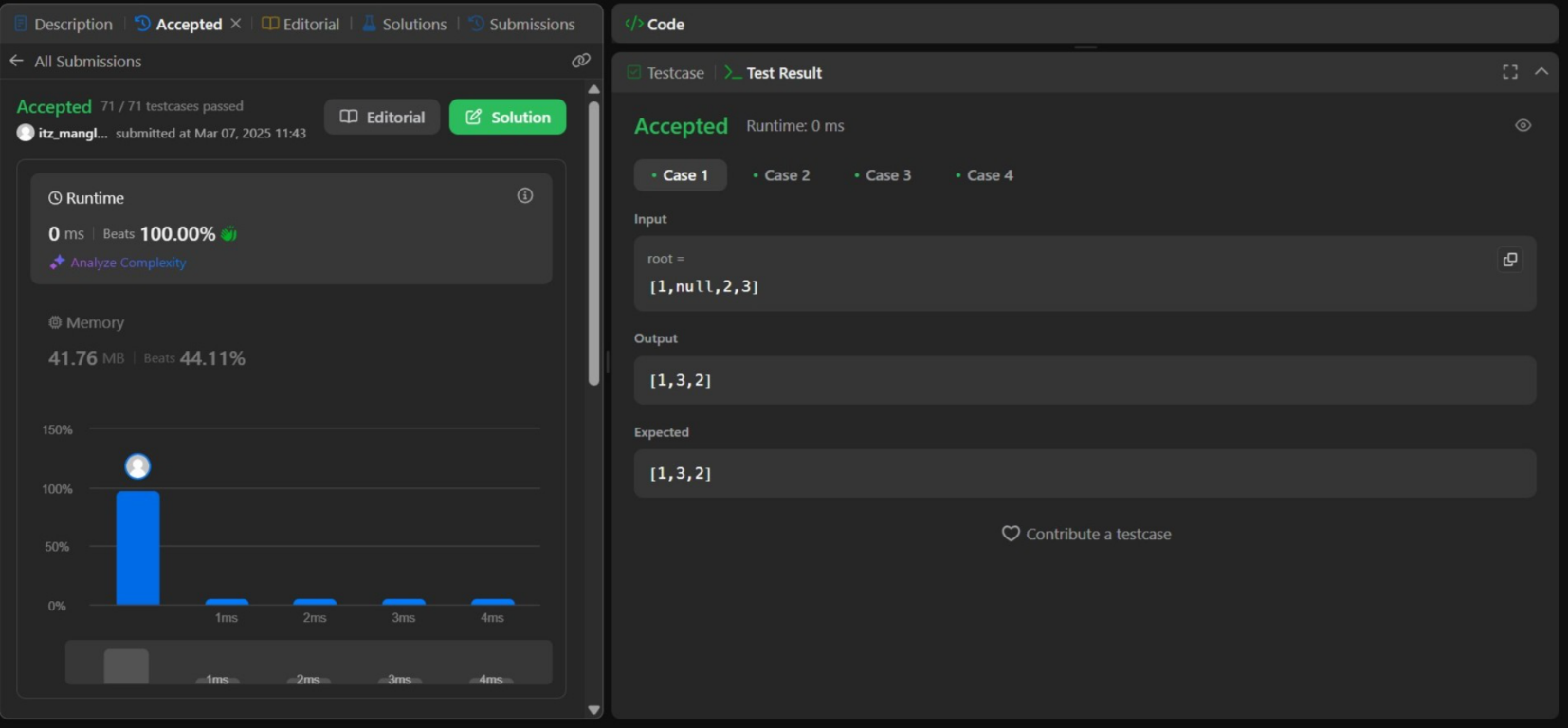
result.add(node.val); // Visit node

inorderHelper(node.right, result); // Traverse right subtree

}

}

**Output :**

****

**Construct Binary Tree from Inorder and Postorder Traversal**

**Aim :**

To construct a binary tree from given inorder and postorder traversal arrays.

**Objectives :**

* Understand the relationship between inorder and postorder traversals.
* Implement a recursive method to build a binary tree.
* Analyze time and space complexity of tree construction algorithms.

**Code :**

import java.util.\*;

// Definition for a binary tree node.

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) { this.val = val; }

TreeNode(int val, TreeNode left, TreeNode right) {

this.val = val;

this.left = left;

this.right = right;

}

}

public class Solution {

private Map<Integer, Integer> inorderMap;

private int postIndex;

public TreeNode buildTree(int[] inorder, int[] postorder) {

inorderMap = new HashMap<>();

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

inorderMap.put(inorder[i], i);

}

postIndex = postorder.length - 1;

return helper(postorder, 0, inorder.length - 1);

}

private TreeNode helper(int[] postorder, int left, int right) {

if (left > right) return null;

int rootVal = postorder[postIndex--];

TreeNode root = new TreeNode(rootVal);

int inorderIndex = inorderMap.get(rootVal);

root.right = helper(postorder, inorderIndex + 1, right);

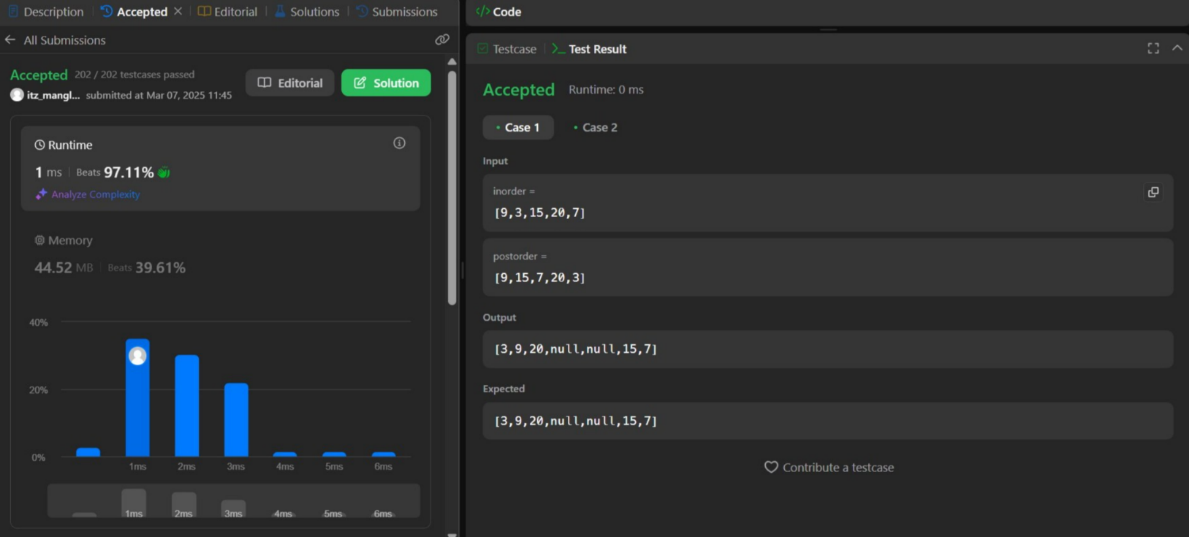
root.left = helper(postorder, left, inorderIndex - 1);

return root;

}

}

**Output :**

****

**Learning Outcomes :**

* Implement in-order traversal to confirm node ordering.
* Compare different validation techniques and their complexities.
* Construct algorithms to connect sibling nodes at each level.
* Optimize tree traversal using constant space techniques.
* Differentiate traversal strategies for perfect and non-perfect binary trees.