**Assignment - 5**

**Name: Armaan UID: 22BCS12007**

**Aim:** To solve leetcode problems

1. Problem : Maximum depth of binary tree

Code:

class Solution {

    public int maxDepth(TreeNode root) {

        // Base case: if the node is null, return 0

        if (root == null) {

            return 0;

        }

        // Recursively find the depth of left and right subtrees

        int leftDepth = maxDepth(root.left);

        int rightDepth = maxDepth(root.right);

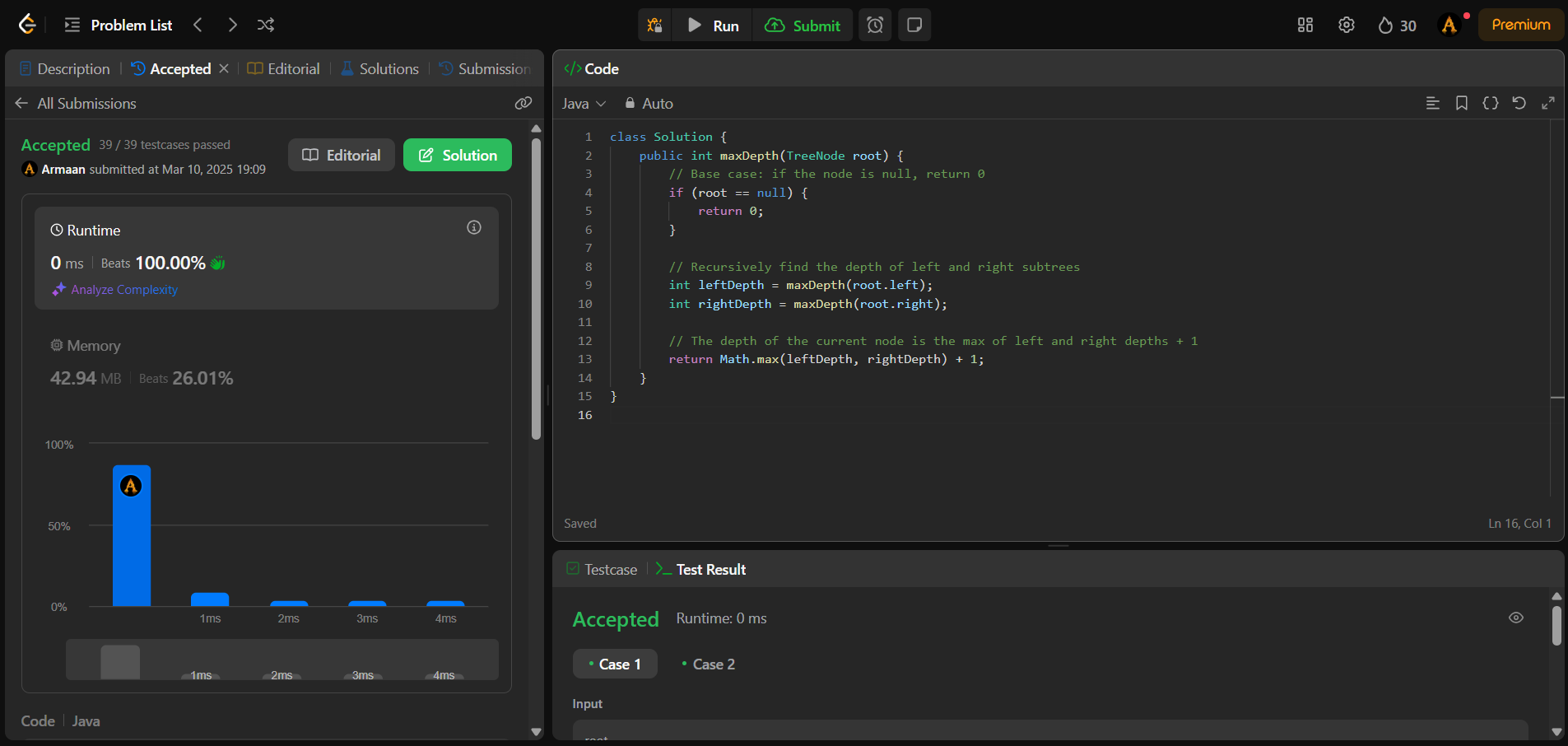
        // The depth of the current node is the max of left and right depths + 1

        return Math.max(leftDepth, rightDepth) + 1;

    }

}

Output :



1. Problem: Validate binary tree

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;

}

// Check if the current node's value is within the valid range

if (node.val <= min || node.val >= max) {

return false;

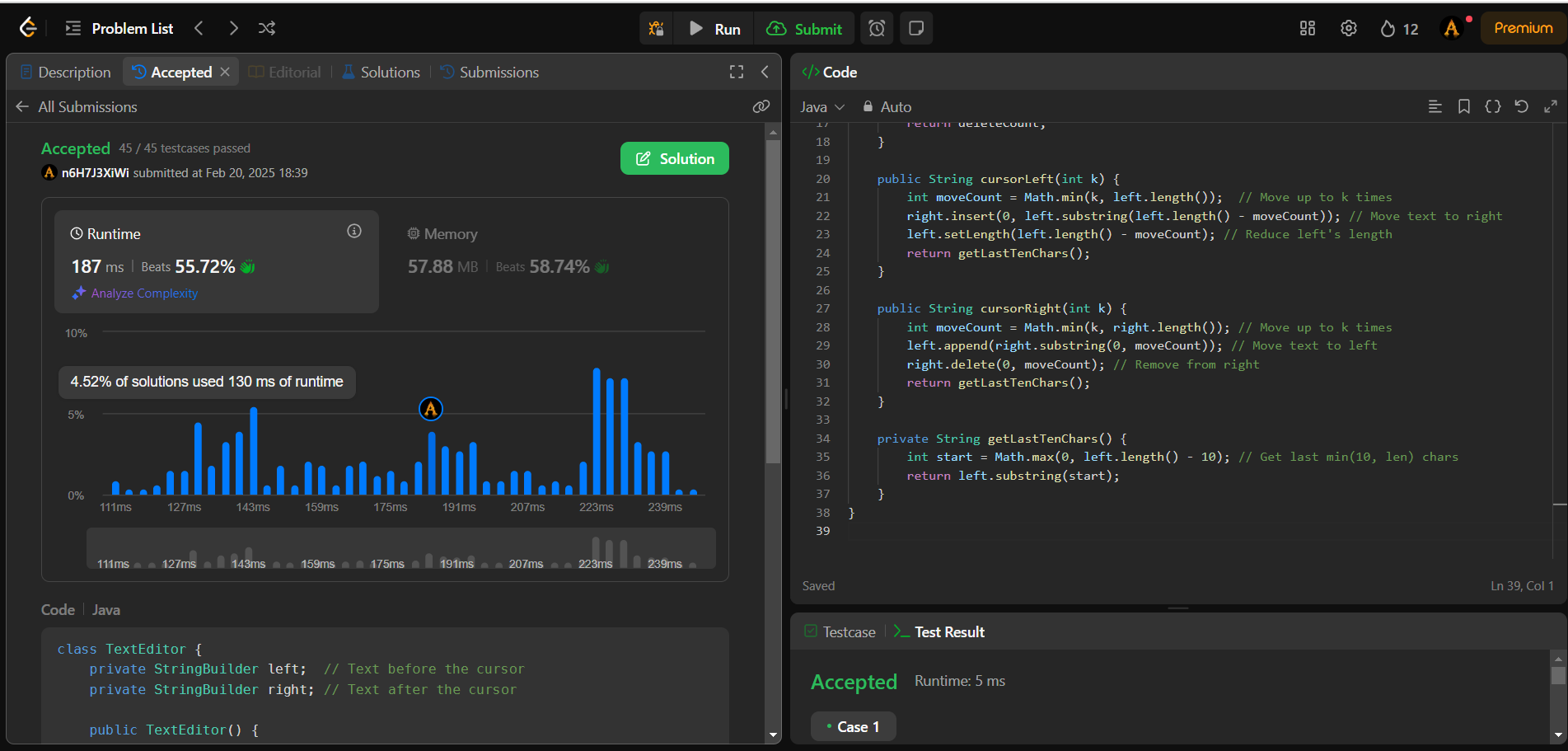
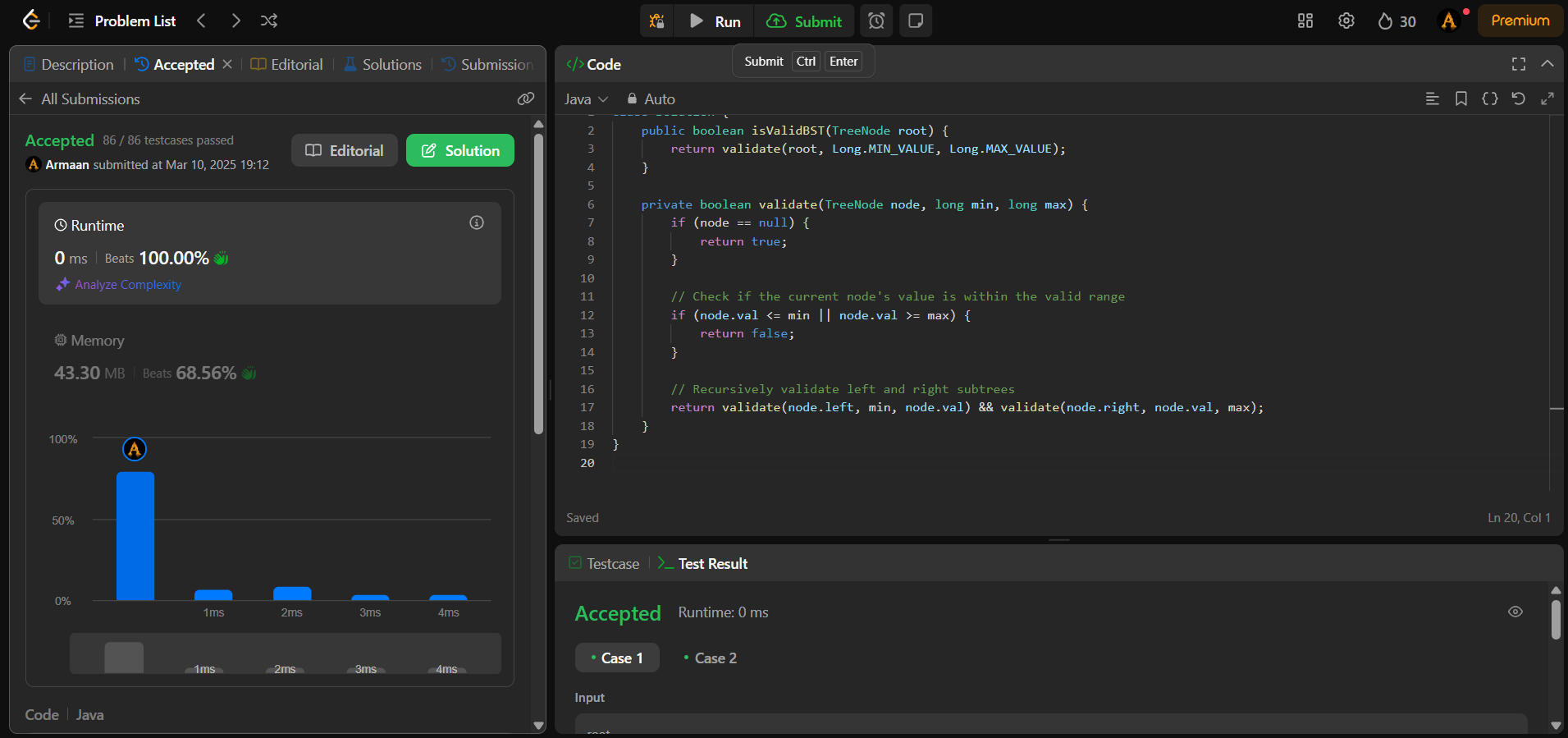
}

// Recursively validate left and right subtrees

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

}

}

Output:

1. Problem: Symmetric tree

Code:

class Solution {

public boolean isSymmetric(TreeNode root) {

if (root == null) {

return true;

}

return isMirror(root.left, root.right);

}

private boolean isMirror(TreeNode t1, TreeNode t2) {

if (t1 == null && t2 == null) {

return true;

}

if (t1 == null || t2 == null) {

return false;

}

return (t1.val == t2.val)

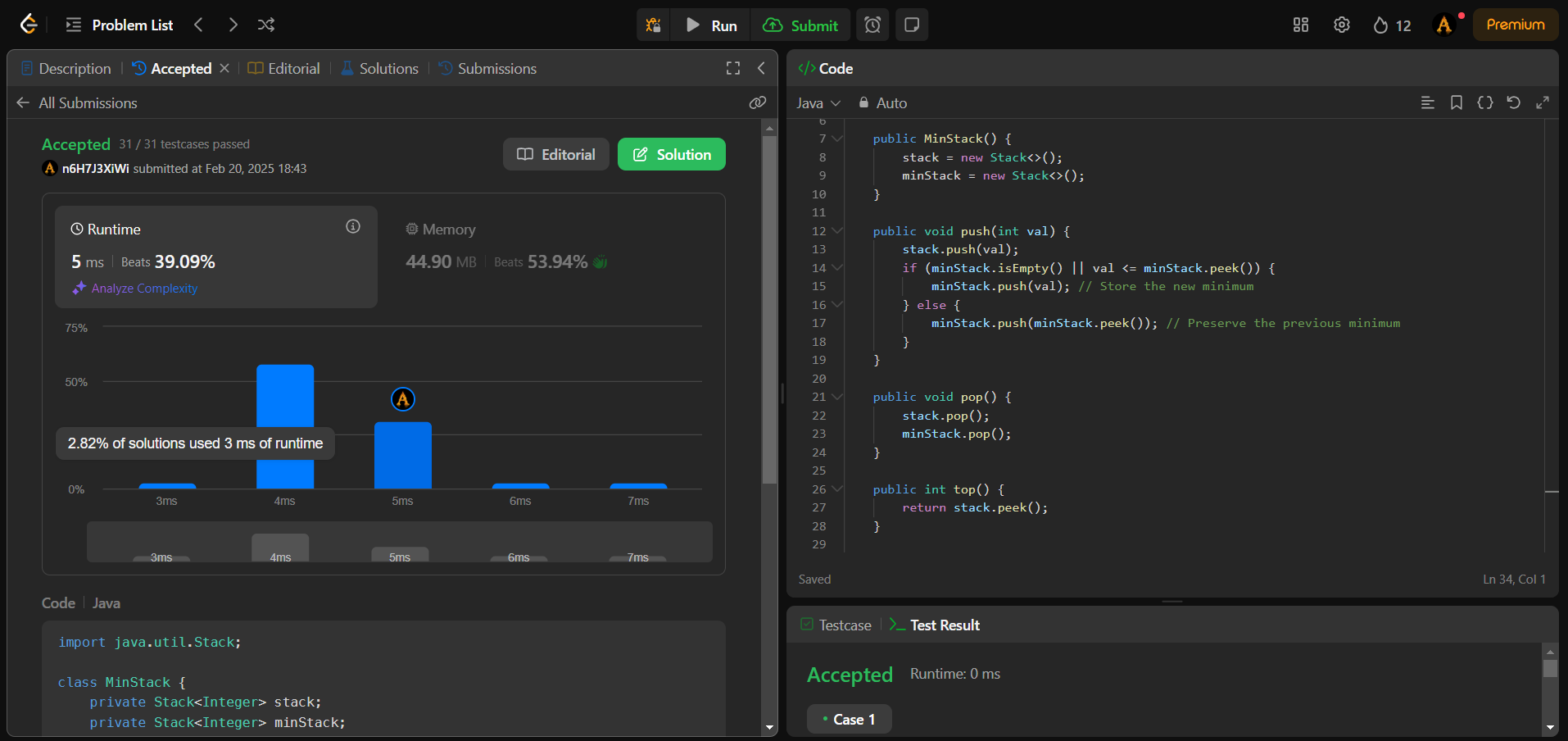
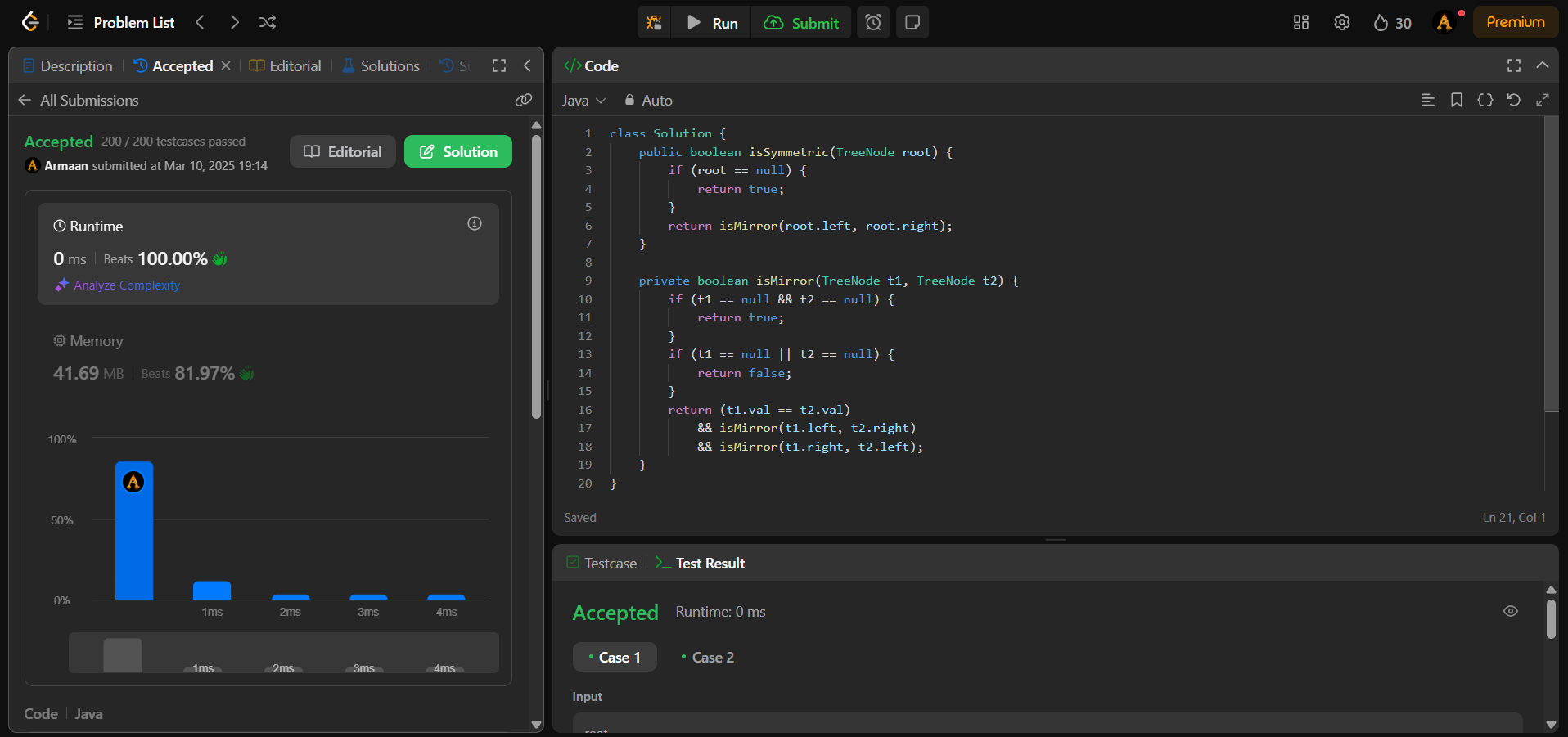
&& isMirror(t1.left, t2.right)

&& isMirror(t1.right, t2.left);

}

}

Output:



1. Problem: binary tree zig zag tree traversal

Code:

class Solution {

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

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

if (root == null) return result;

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

queue.offer(root);

boolean leftToRight = true;

while (!queue.isEmpty()) {

int size = queue.size();

Deque<Integer> level = new LinkedList<>()

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

TreeNode node = queue.poll();

if (leftToRight) {

level.offerLast(node.val);

} else {

level.offerFirst(node.val);

}

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

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

}

result.add(new ArrayList<>(level));

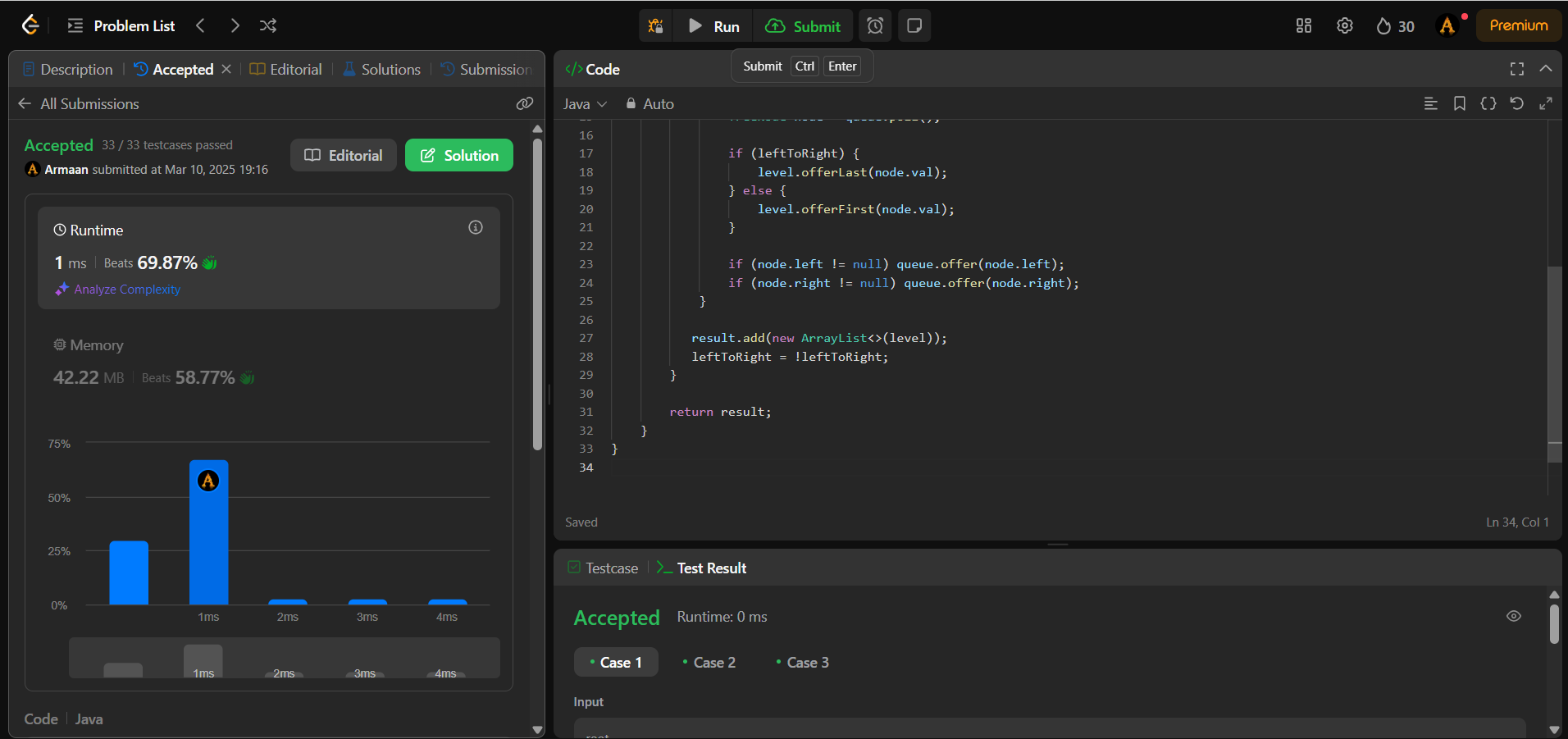
leftToRight = !leftToRight;

}

return result;

}

}

Output:

1. Problem: lowest common ancestor

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

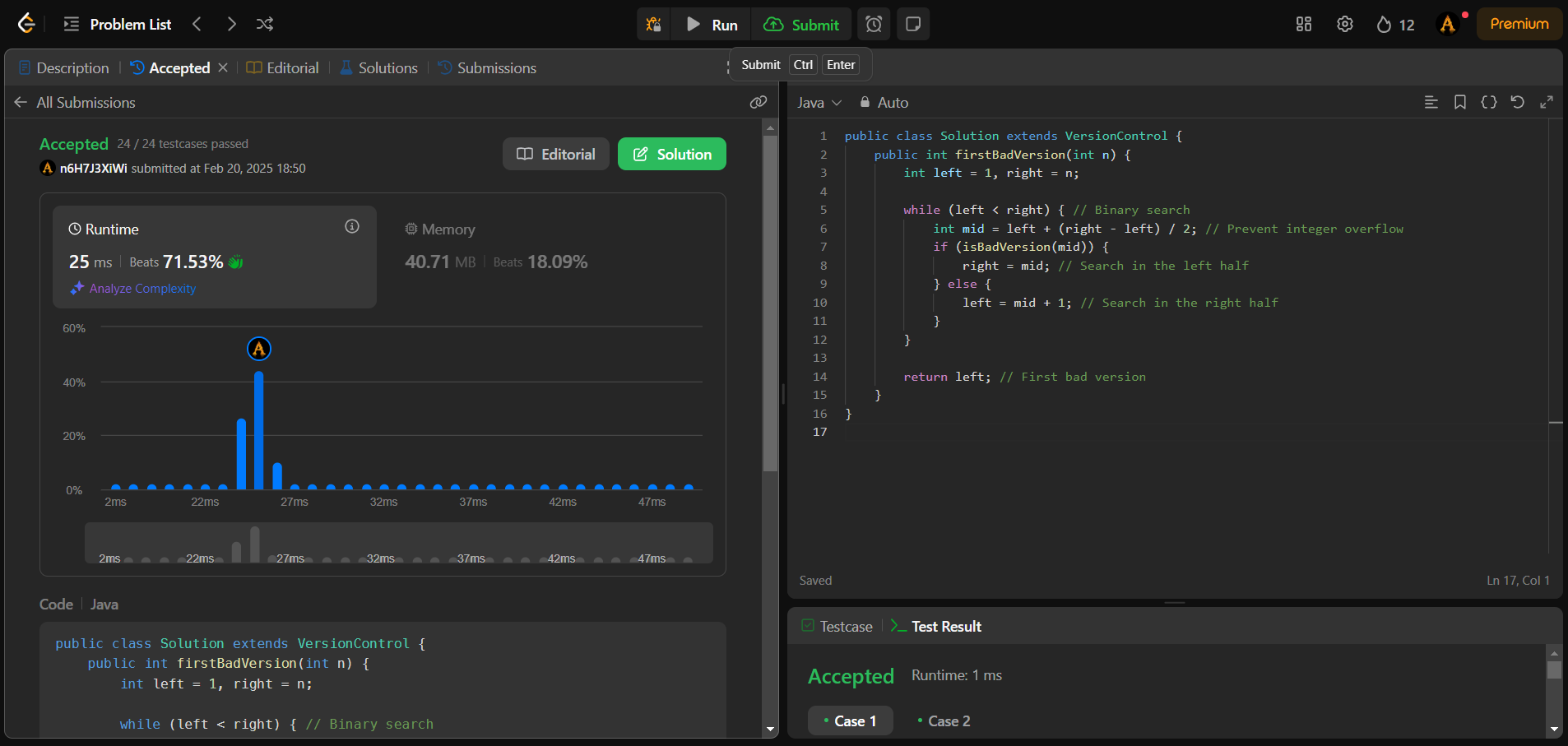
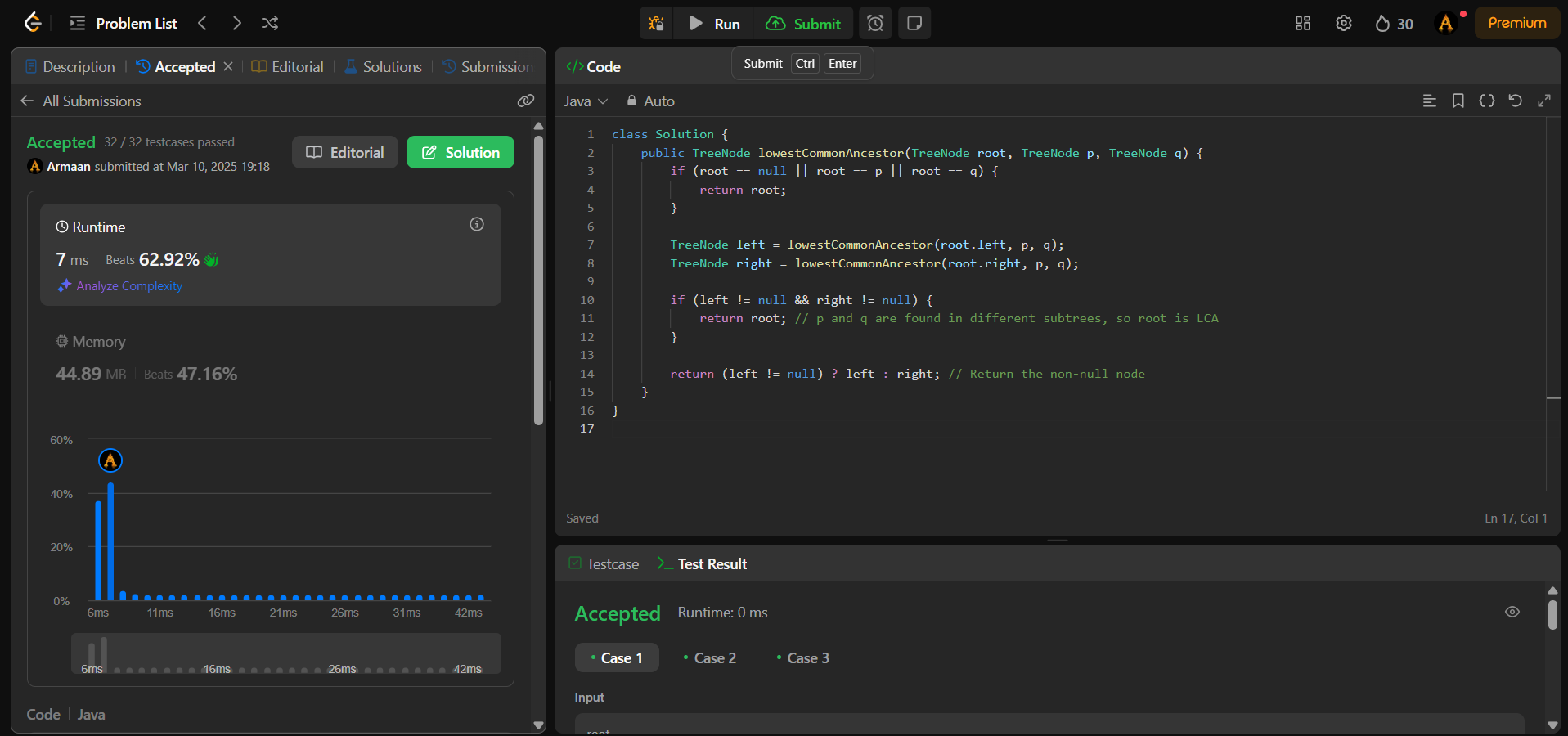
}

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

}

}

Output:



1. Problem: binary tree inorder traversal

Code:

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); // Visit left subtree

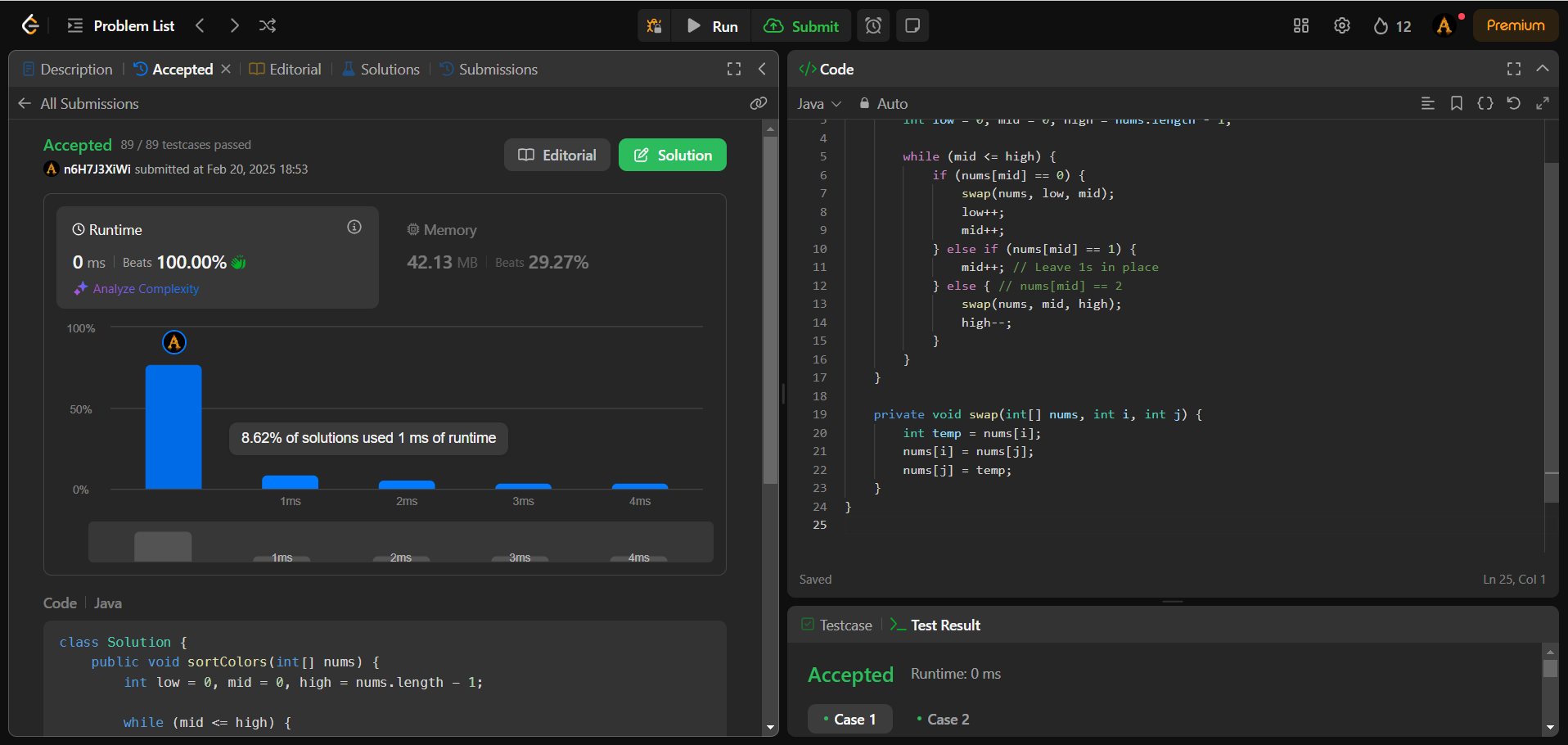
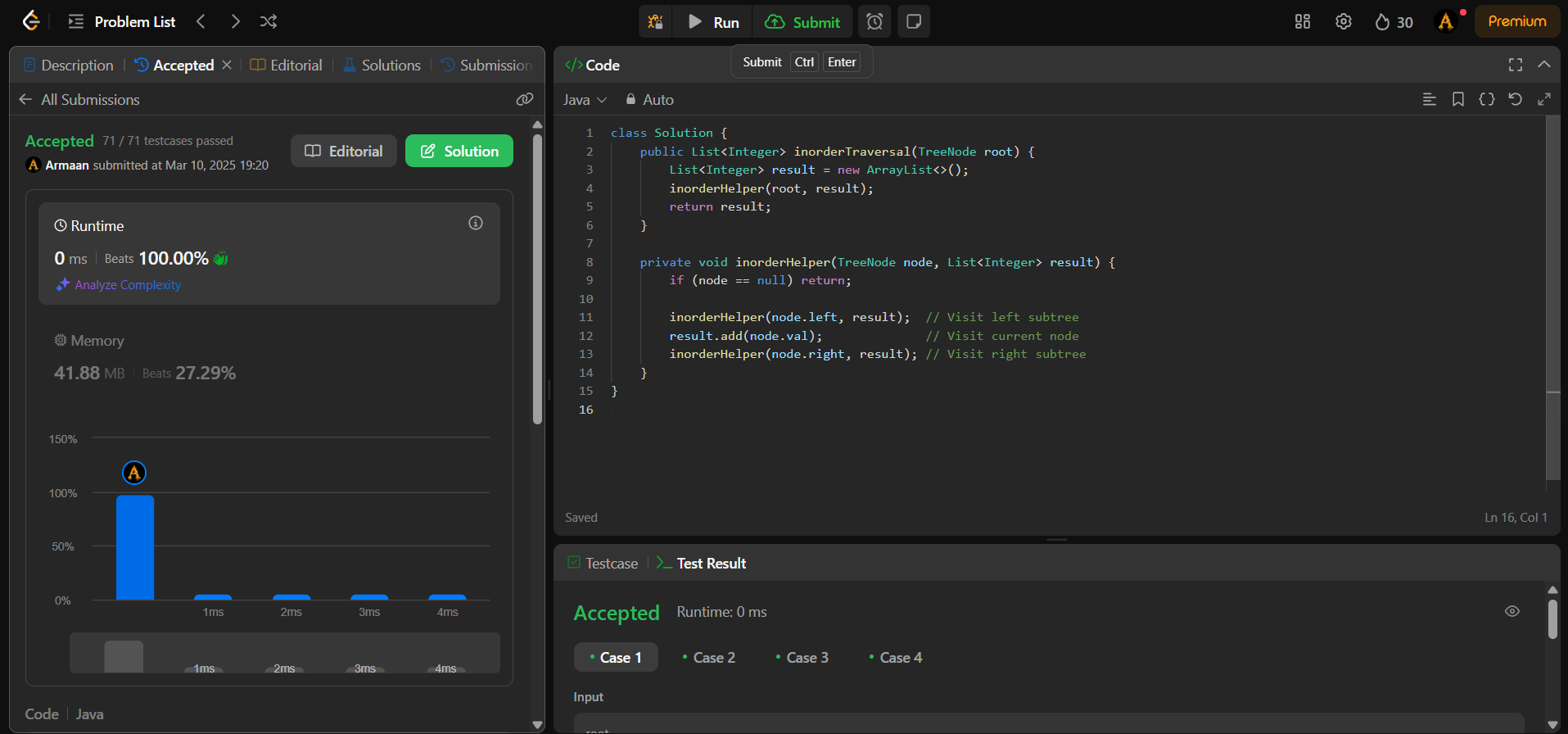
result.add(node.val); // Visit current node

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

}

}

Output:



1. Problem: binary tree level order traversal

Code:

import java.util.\*;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) { val = x; }

}

class Solution {

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

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

if (root == null) return result; // Edge case: empty tree

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 node = queue.poll();

currentLevel.add(node.val);

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

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

}

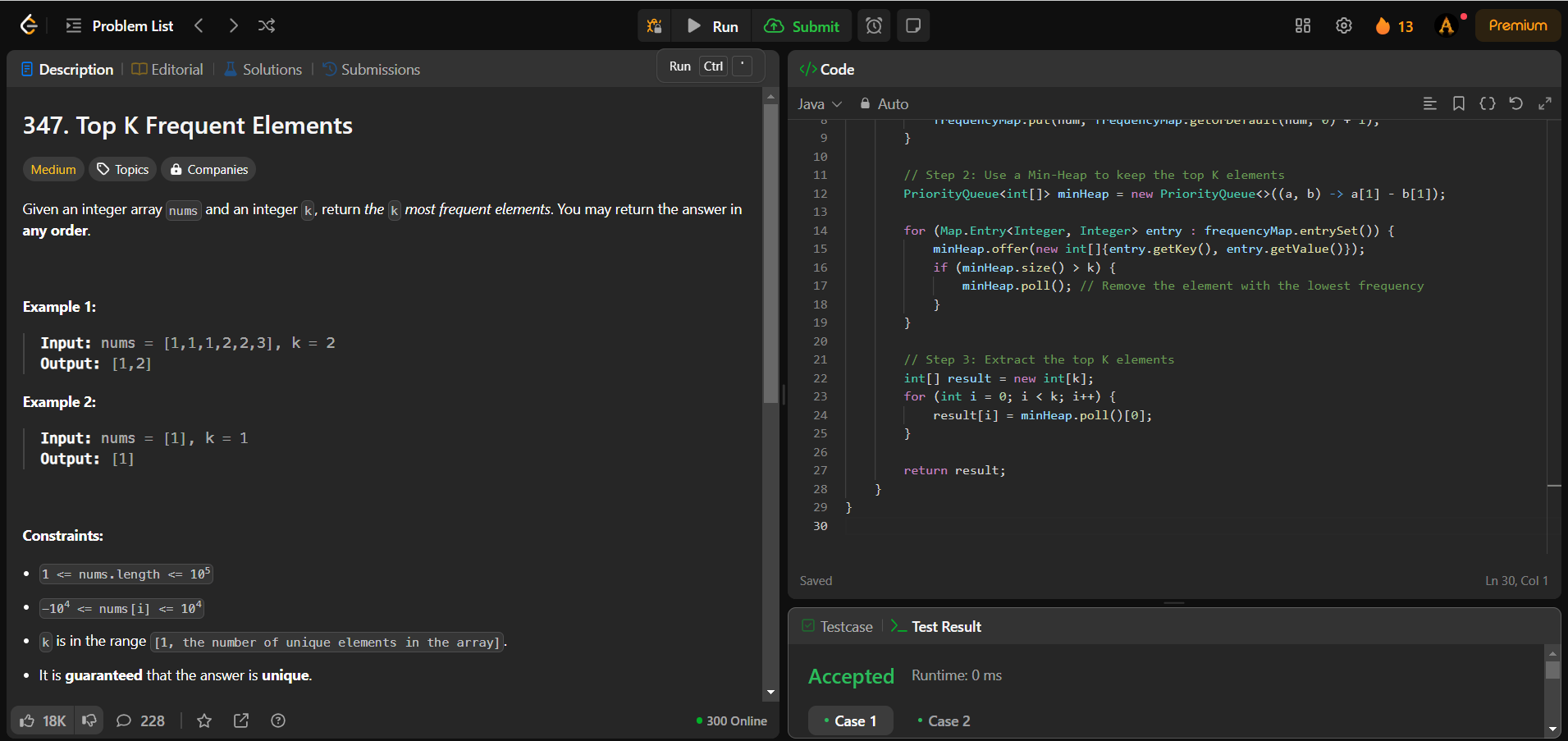
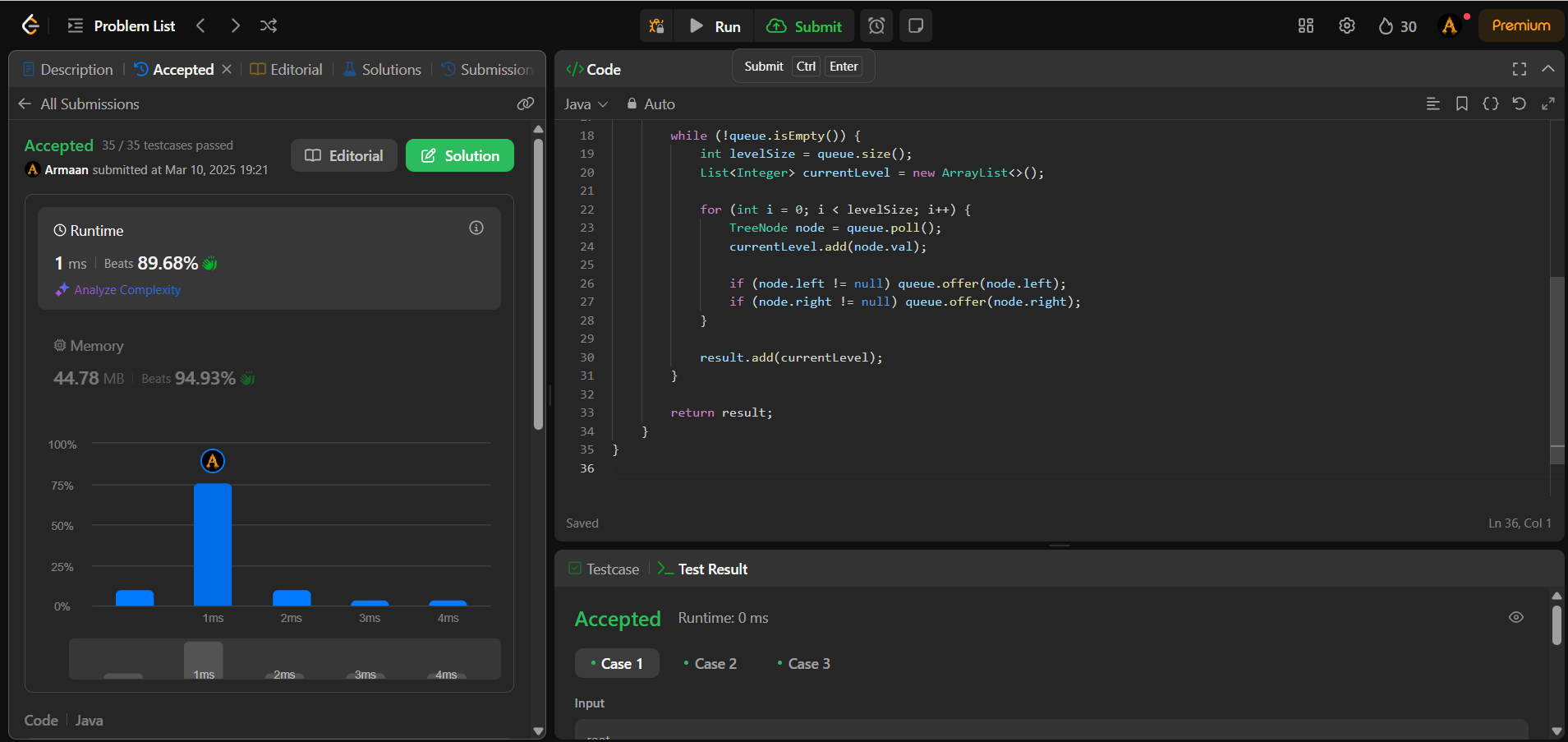
result.add(currentLevel);

}

return result;

}

}

Output:

1. Problem: Kth smallest element in a BST

Code:

import java.util.\*;

class Solution {

public int kthSmallest(TreeNode root, int k) {

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

inorderTraversal(root, inorderList);

return inorderList.get(k - 1); // 1-indexed, so k-1 in list

}

private void inorderTraversal(TreeNode node, List<Integer> list) {

if (node == null) return;

inorderTraversal(node.left, list); // Visit left subtree

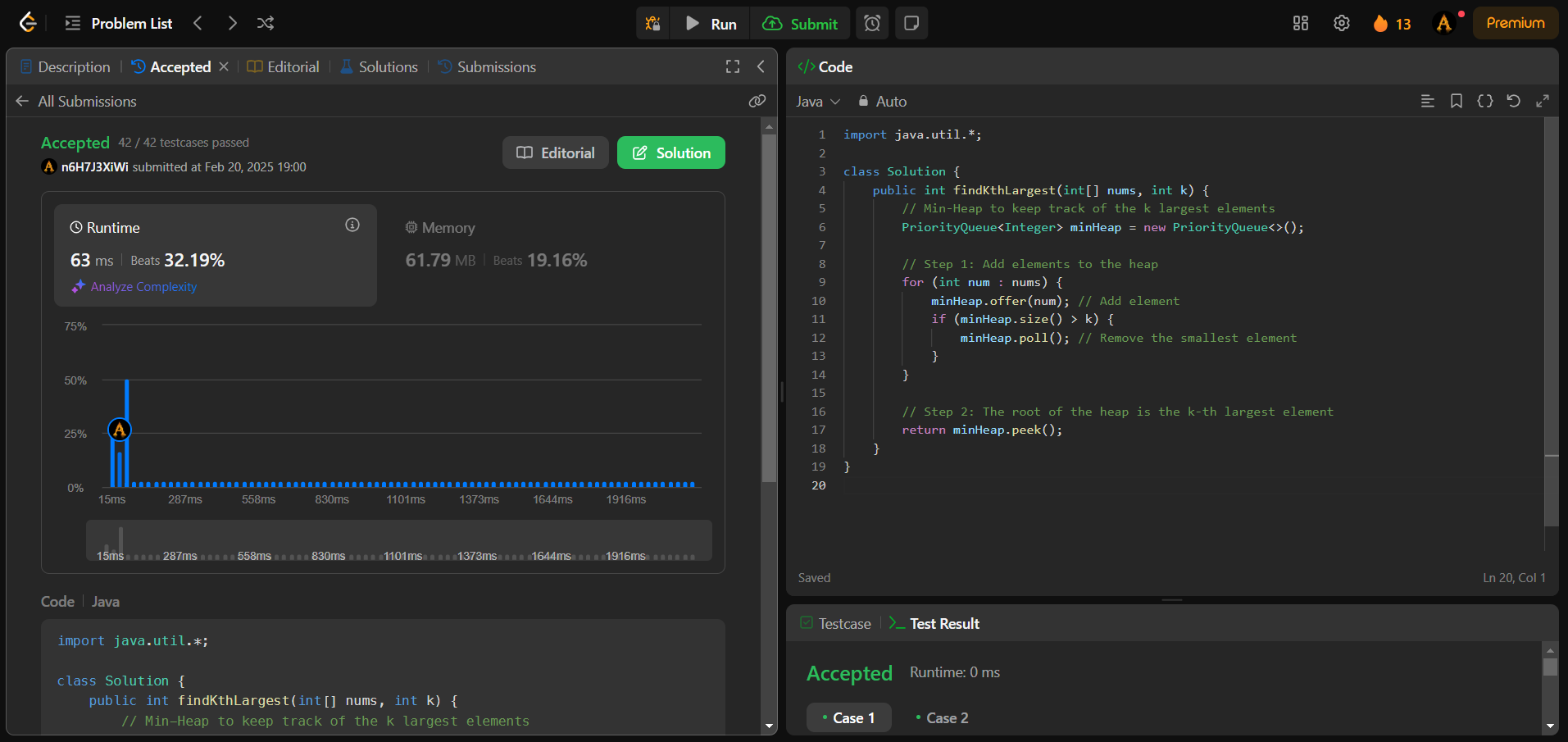
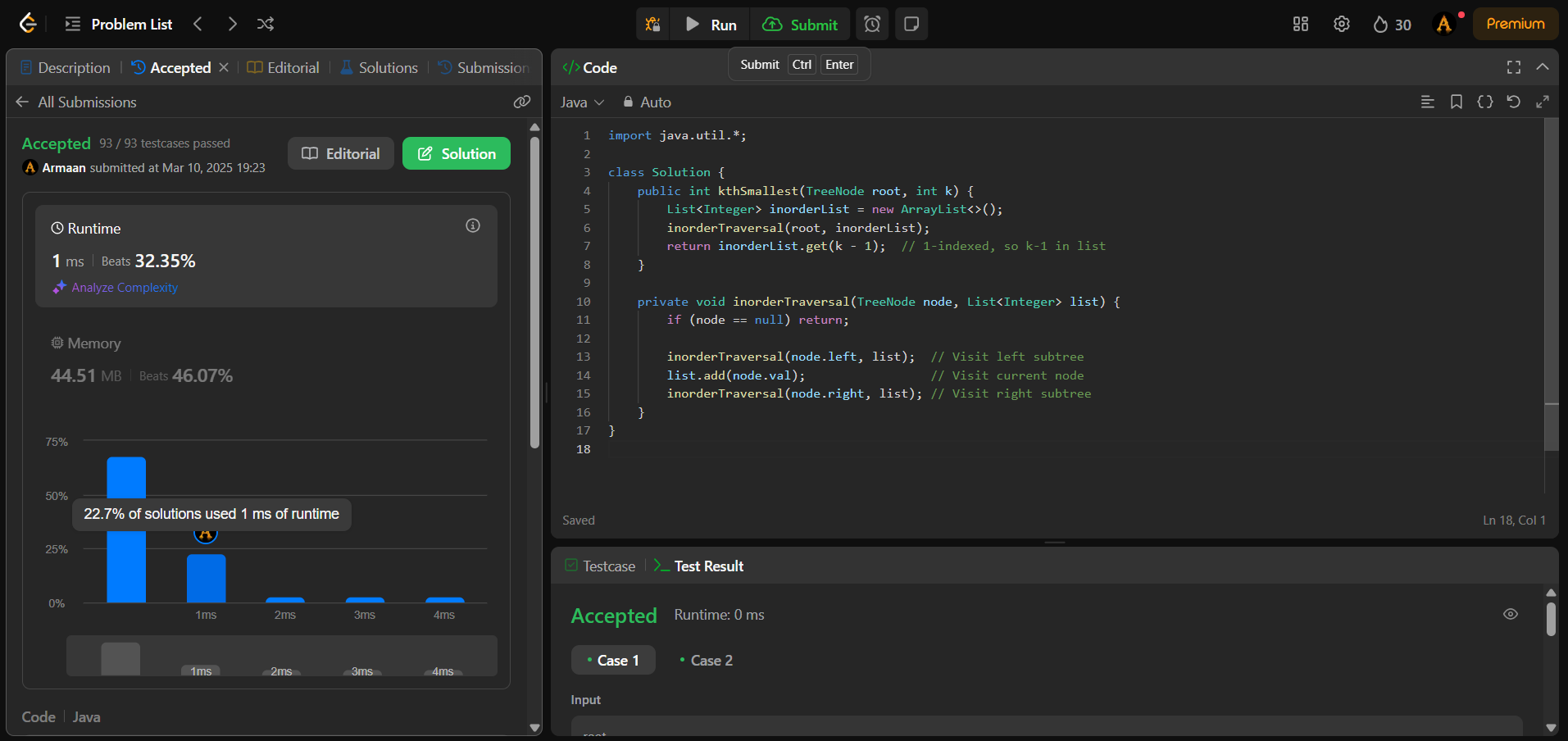
list.add(node.val); // Visit current node

inorderTraversal(node.right, list); // Visit right subtree

}

}

Output:



1. Problem: populating next right pointer in each node

Code:

class Solution {

    public Node connect(Node root) {

        if (root == null || root.left == null) return root;  // Base case

        root.left.next = root.right;  // Connect left → right

        if (root.next != null) {

            root.right.next = root.next.left;  // Connect right → next left

        }

        connect(root.left);  // Recursive call for left subtree

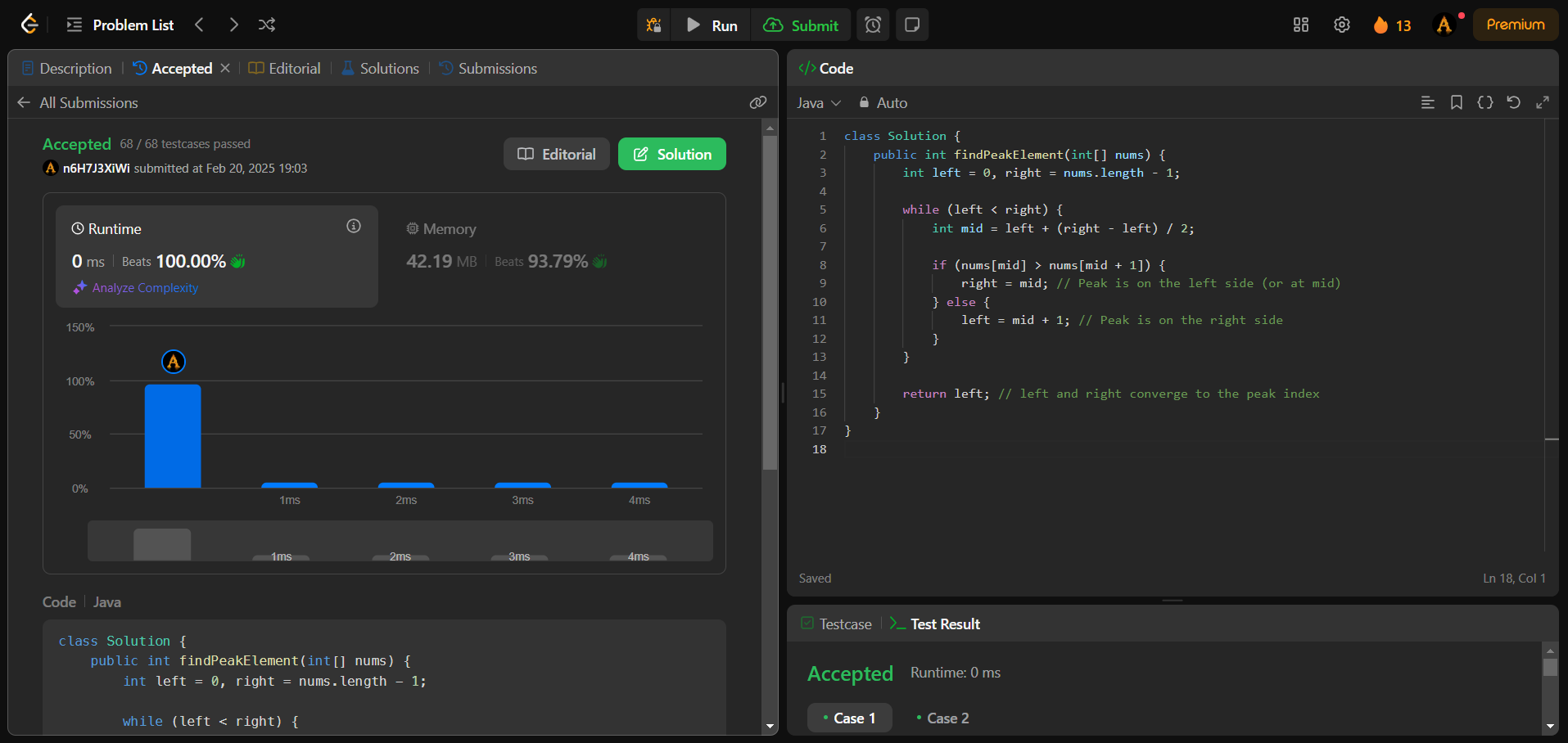
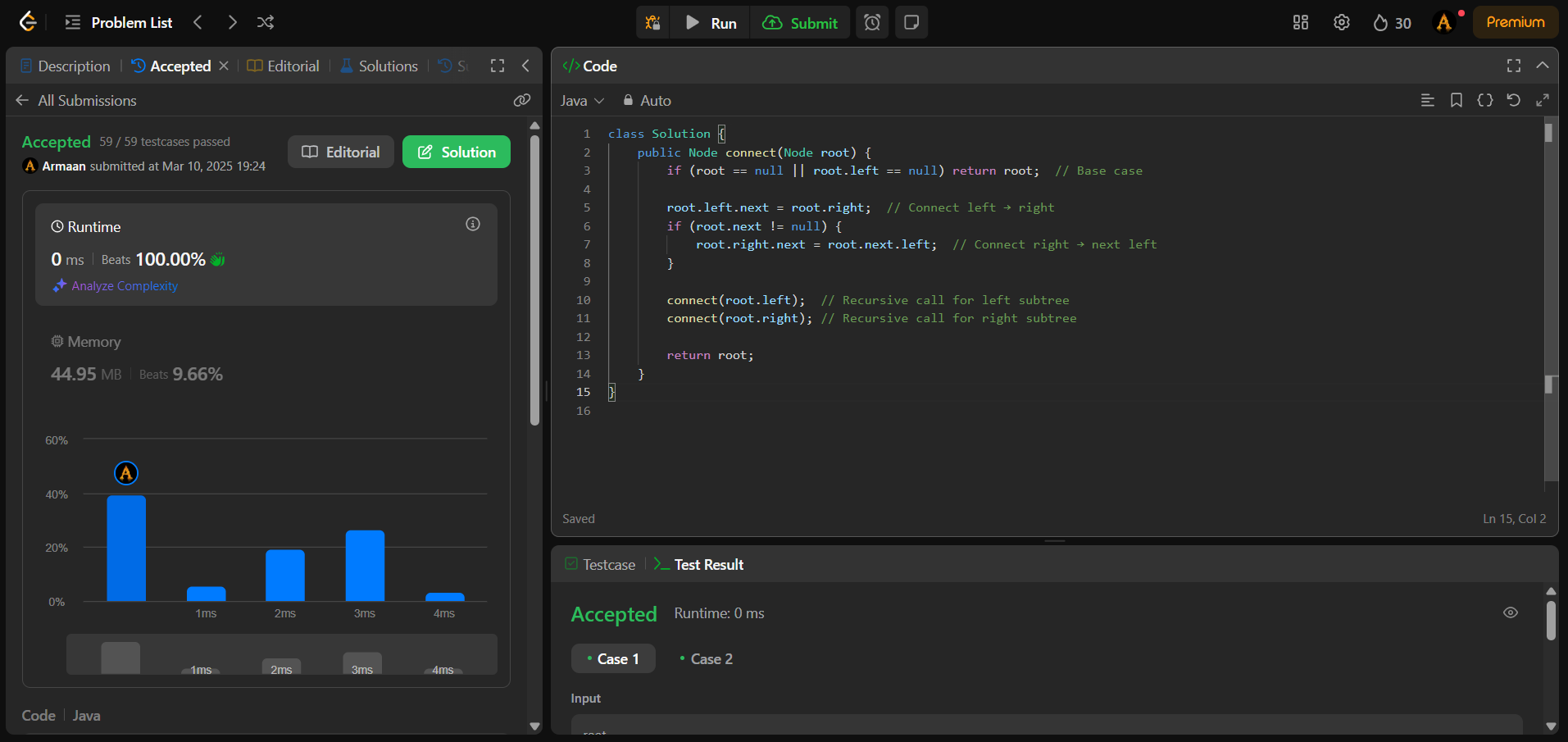
        connect(root.right); // Recursive call for right subtree

        return root;

    }

}

Output:



1. Problem: Sum of left leaves

Code:

class Solution {

public int sumOfLeftLeaves(TreeNode root) {

if (root == null) return 0;

int sum = 0;

// Check if left child is a leaf

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

sum += root.left.val;

}

// Recur for left and right subtrees

sum += sumOfLeftLeaves(root.left);

sum += sumOfLeftLeaves(root.right);

return sum;

}

}

Output:

