Assignment - 7

Name: Aanchal Negi **UID: 22BCS14969**

Branch: BE-CSE **Section/Group: IOT-609/B**

Semester: 6th **Date :02/04/25**

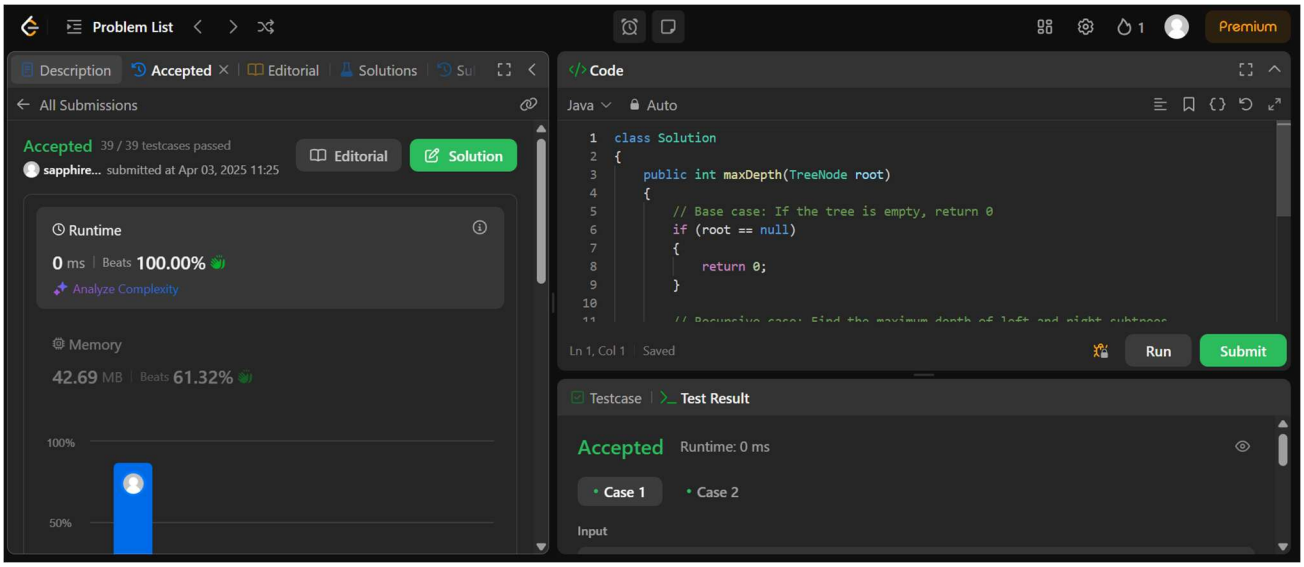
Subject Name: AP Lab-2 **Subject Code: 22CSP-351**

➢ Maximum Depth of Binary Tree

Code:

```
class Solution
{
    public int maxDepth(TreeNode root)
    {
        // Base case: If the tree is empty, return 0
        if (root == null)
        {
            return 0;
        }

        // Recursive case: Find the maximum depth of left and right subtrees
        int leftDepth = maxDepth(root.left);
        int rightDepth = maxDepth(root.right);

        // Return the maximum depth plus one for the current node
        return Math.max(leftDepth, rightDepth) + 1;
    }
}
```

Output:

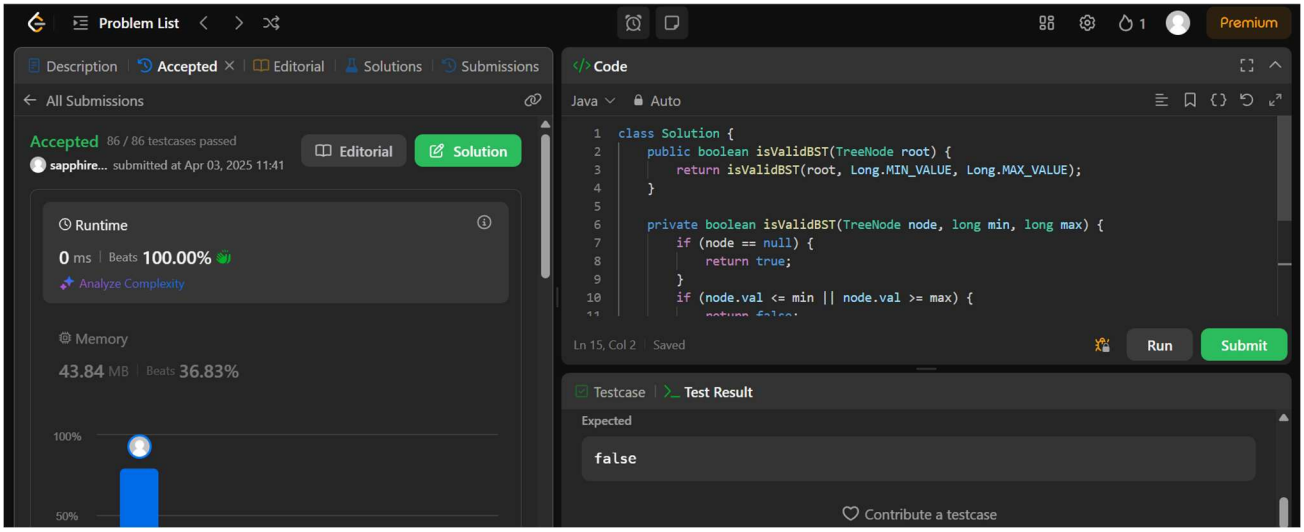


➢ Validate Binary Search Tree

Code:

```
class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }

    private boolean isValidBST(TreeNode node, long min, long max) {
        if (node == null) {
            return true;
        }
        if (node.val <= min || node.val >= max) {
            return false;
        }
        return isValidBST(node.left, min, node.val) && isValidBST(node.right, node.val, max);
    }
}
```

Output:

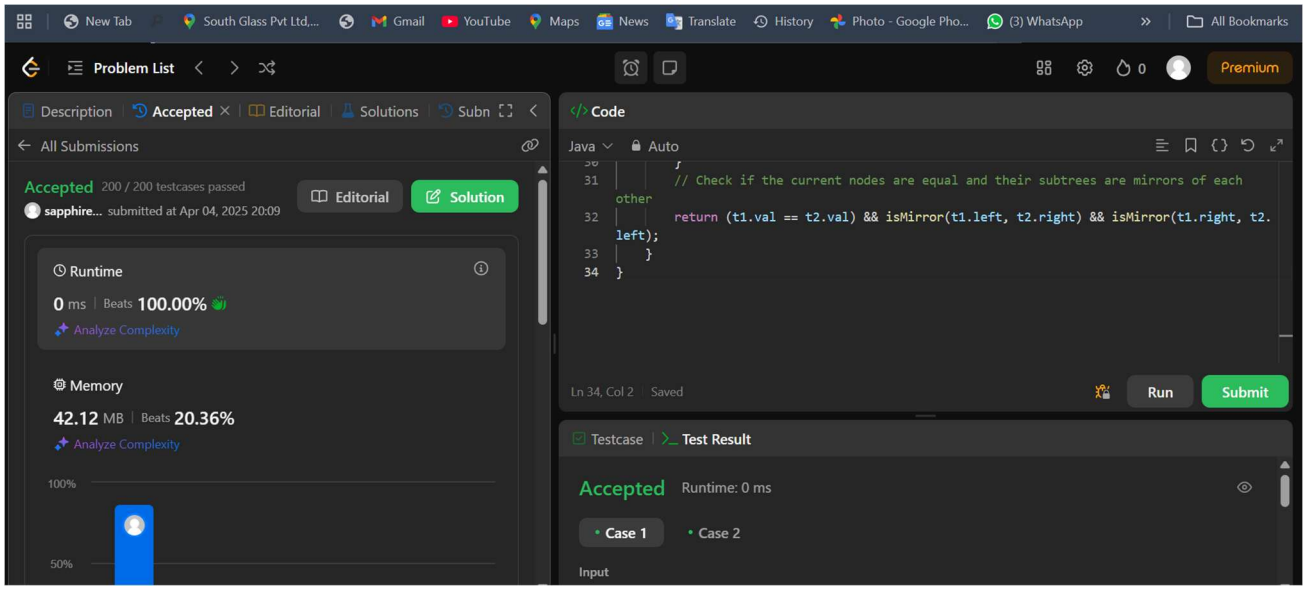


➢ Symmetric Tree

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 subtrees are null, so they're symmetric
        }
        if (t1 == null || t2 == null) {
            return false; // One of the subtrees is null, so not symmetric
        }
        // Check if the current nodes are equal and their subtrees are mirrors of each other
        return (t1.val == t2.val) && isMirror(t1.left, t2.right) && isMirror(t1.right, t2.left);
    }
}
```

Output:

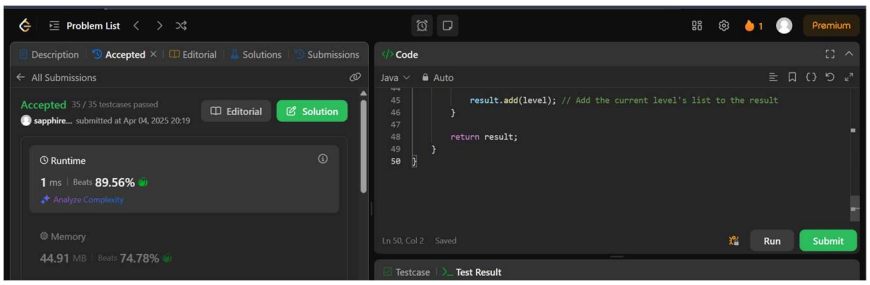


➢ Binary Tree Level Order Traversal

Code:

```
import java.util.*;

class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) {
            return result; // Return an empty list if the tree is empty
        }

        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root); // Start with the root node

        while (!queue.isEmpty()) {
            int levelSize = queue.size(); // Number of nodes in the current level
            List<Integer> level = new ArrayList<>();

            for (int i = 0; i < levelSize; i++) {
                TreeNode currentNode = queue.poll(); // Remove the front node from the queue
                level.add(currentNode.val); // Add its value to the current level's list

                // Add left and right children to the queue if they exist
                if (currentNode.left != null) {
                    queue.add(currentNode.left);
                }
                if (currentNode.right != null) {
                    queue.add(currentNode.right);
                }
            }

            result.add(level); // Add the current level's list to the result
        }

        return result;
    }
}
```

Output:

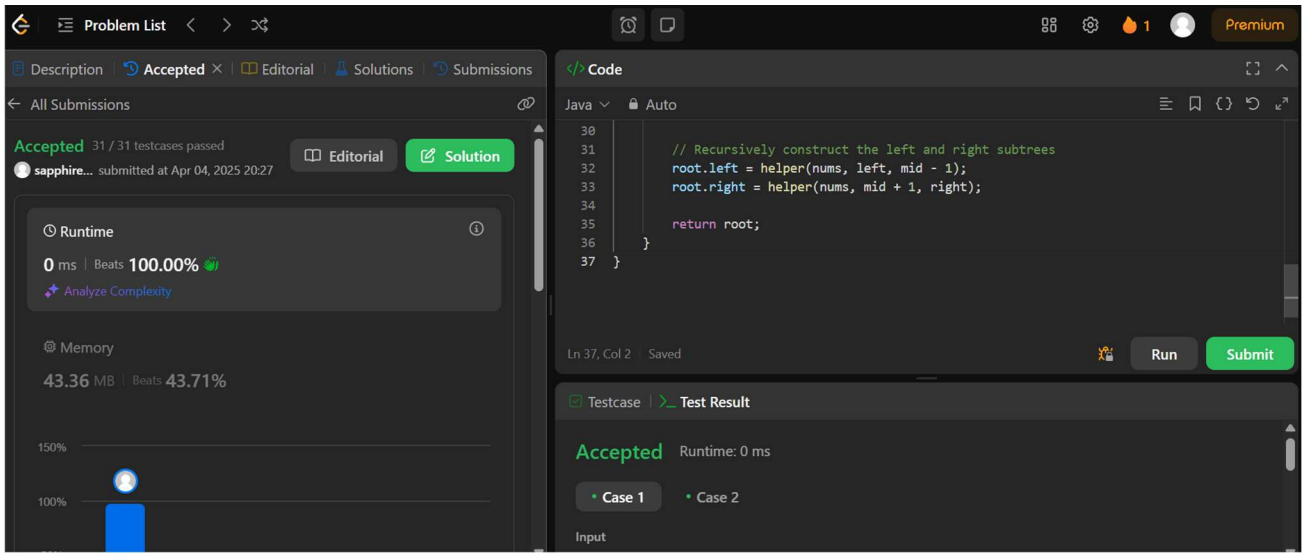


➢ Convert Sorted Array to Binary Search Tree

Code:

```
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        return helper(nums, 0, nums.length - 1);
    }

    private TreeNode helper(int[] nums, int left, int right) {
        // Base case: If the range is invalid
        if (left > right) {
            return null;
        }

        // Find the middle element to make it the root
        int mid = left + (right - left) / 2;
        TreeNode root = new TreeNode(nums[mid]);

        // Recursively construct the left and right subtrees
        root.left = helper(nums, left, mid - 1);
        root.right = helper(nums, mid + 1, right);

        return root;
    }
}
```

Output:

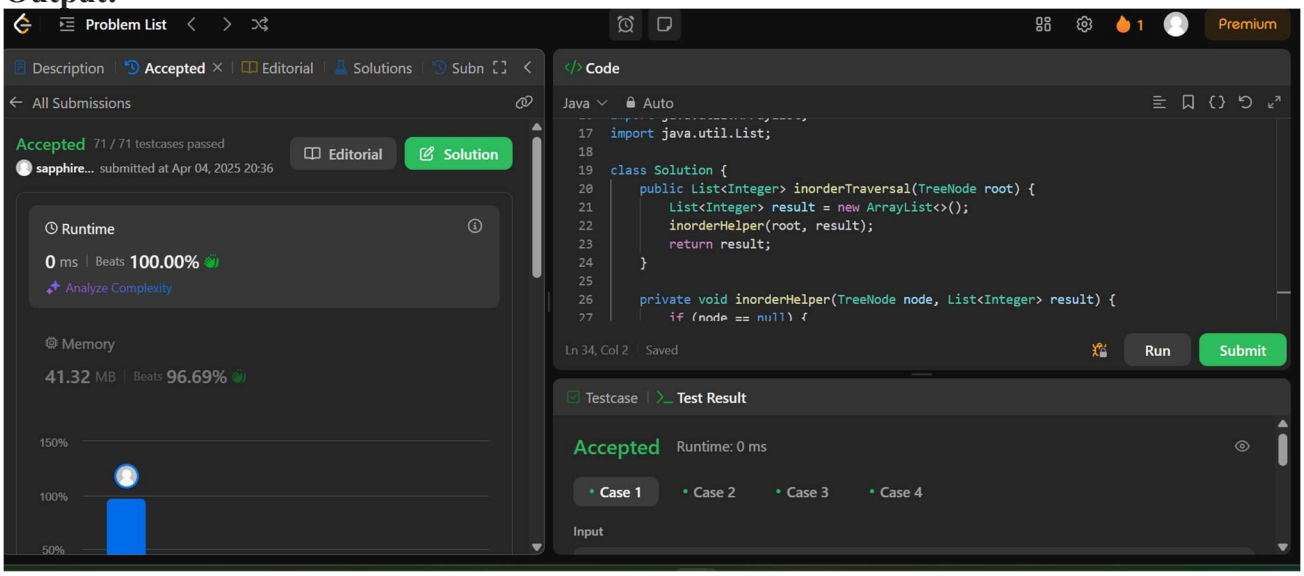


➢ Binary Tree Inorder Traversal

Code:

```
import java.util.ArrayList;
import java.util.List;

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; // Base case: if node is null, do nothing
        }
        inorderHelper(node.left, result); // Visit left subtree
        result.add(node.val); // Visit current node
        inorderHelper(node.right, result); // Visit right subtree
    }
}
```

Output:



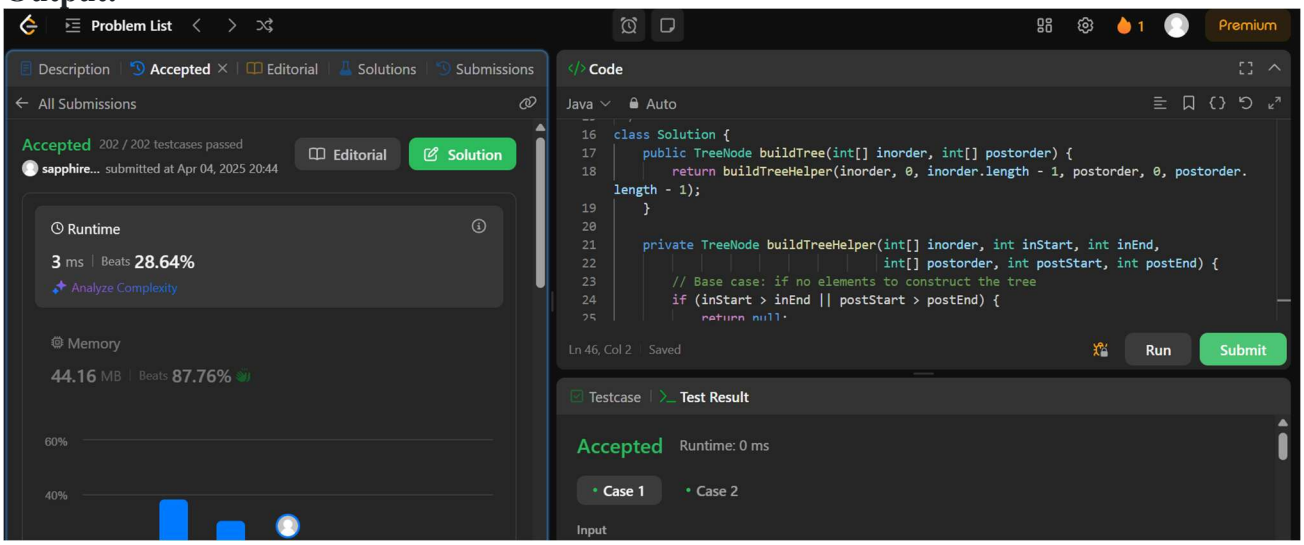
➢ Construct Binary Tree from Inorder and Postorder Traversal

Code:

Code:

```
class Solution {
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        return buildTreeHelper(inorder, 0, inorder.length - 1, postorder, 0, postorder.length - 1);
    }

    private TreeNode buildTreeHelper(int[] inorder, int inStart, int inEnd,
                                     int[] postorder, int postStart, int postEnd) {
        // Base case: if no elements to construct the tree
        if (inStart > inEnd || postStart > postEnd) {
            return null;
        }

        // The root is the last element of the postorder traversal
        TreeNode root = new TreeNode(postorder[postEnd]);

        // Find the index of the root in the inorder array
        int rootIndex = inStart;
        while (rootIndex <= inEnd && inorder[rootIndex] != root.val) {
            rootIndex++;
        }

        // Calculate the size of the left subtree
        int leftSize = rootIndex - inStart;

        // Recursively construct the left and right subtrees
        root.left = buildTreeHelper(inorder, inStart, rootIndex - 1, postorder, postStart, postStart + leftSize - 1);
        root.right = buildTreeHelper(inorder, rootIndex + 1, inEnd, postorder, postStart + leftSize, postEnd - 1);

        return root;
    }
}
```

Output:

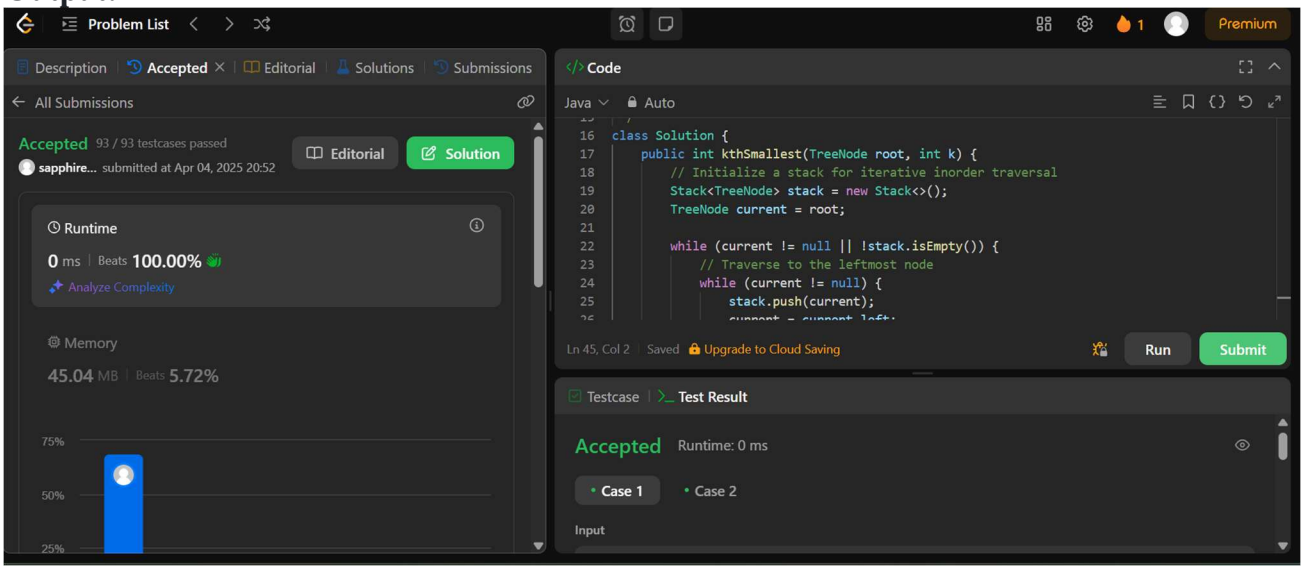


➢ Kth Smallest element in a BST

Code:

```
class Solution {
    public int kthSmallest(TreeNode root, int k) {
        // Initialize a stack for iterative inorder traversal
        Stack<TreeNode> stack = new Stack<>();
        TreeNode current = root;

        while (current != null || !stack.isEmpty()) {
            // Traverse to the leftmost node
            while (current != null) {
                stack.push(current);
                current = current.left;
            }

            // Process the node
            current = stack.pop();
            k--;

            // If k becomes 0, we have found the kth smallest element
            if (k == 0) {
                return current.val;
            }

            // Move to the right subtree
            current = current.right;
        }

        // If k is invalid, return -1 (though this shouldn't happen for valid input)
        return -1;
    }
}
```

Output:



➢ Populating Next Right Pointers in Each Node

Code:

```
class Solution {
    public Node connect(Node root) {
        if (root == null) {
            return null; // If the tree is empty
        }

        // Start with the root node
        Node levelStart = root;

        while (levelStart.left != null) { // Traverse levels until leaf nodes
            Node current = levelStart;

            while (current != null) {
                // Connect left child to right child
                current.left.next = current.right;

                // Connect right child to the left child of the next node
                if (current.next != null) {
                    current.right.next = current.next.left;
                }

                // Move to the next node in the level
                current = current.next;
            }

            // Move to the next level
            levelStart = levelStart.left;
        }

        return root;
    }
}
```

Output: