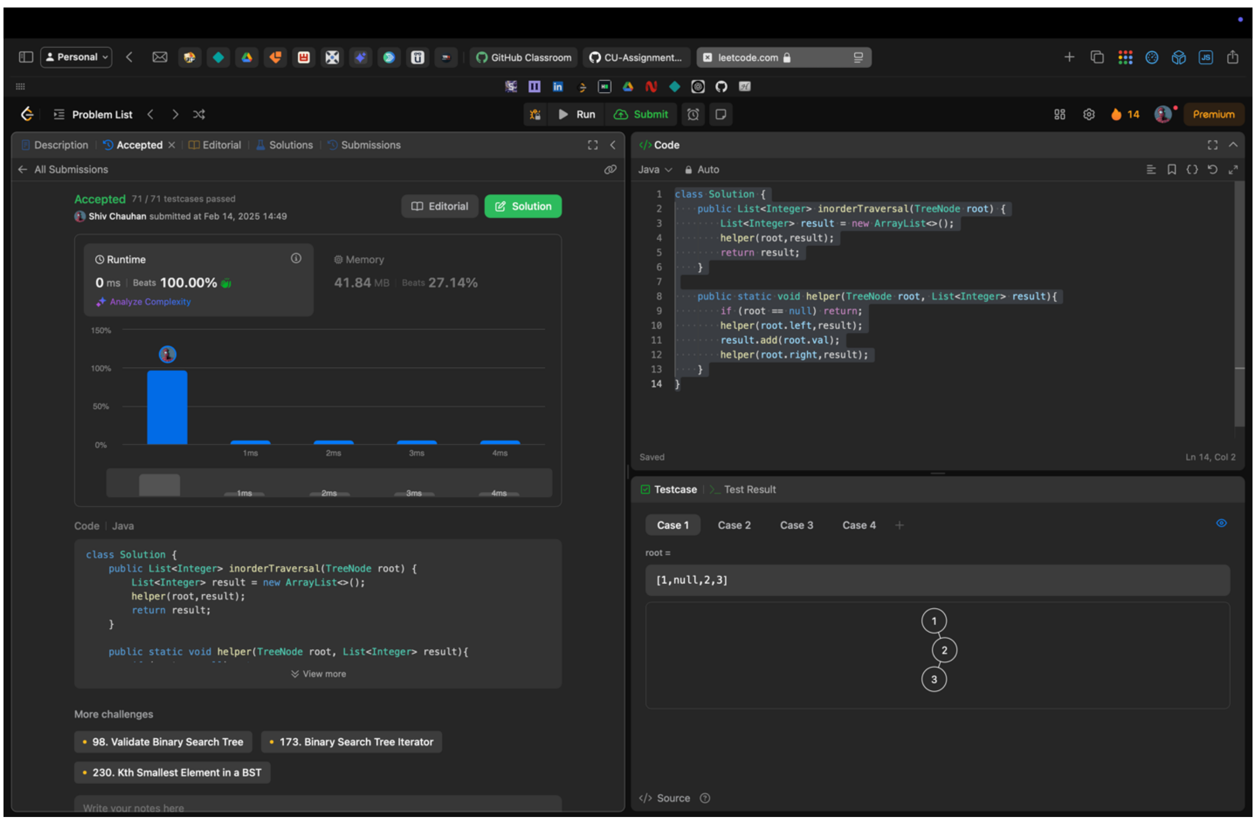
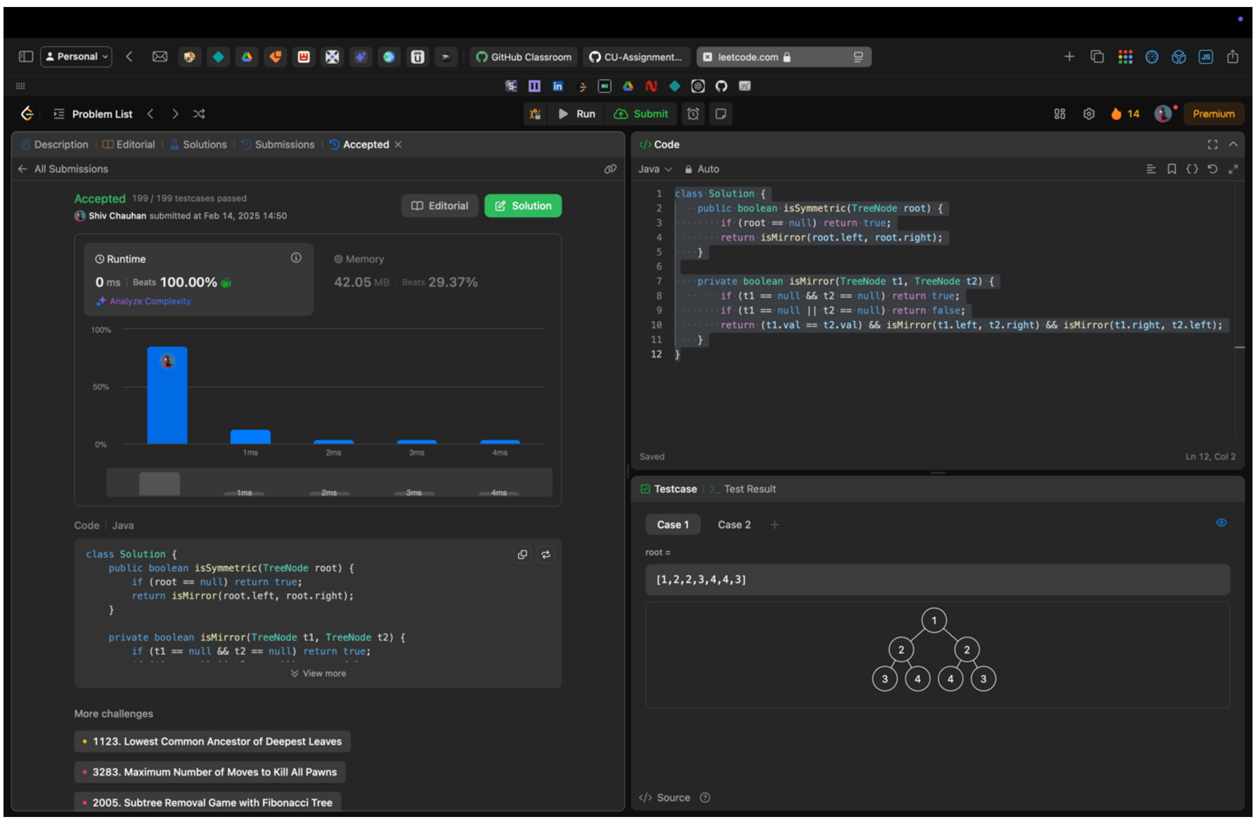
Binary Tree Inorder Traversal

```java
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        helper(root,result);
        return result;
    }

    public static void helper(TreeNode root, List<Integer> result){
        if (root == null) return;
        helper(root.left,result);
        result.add(root.val);
        helper(root.right,result);
    }
}
```



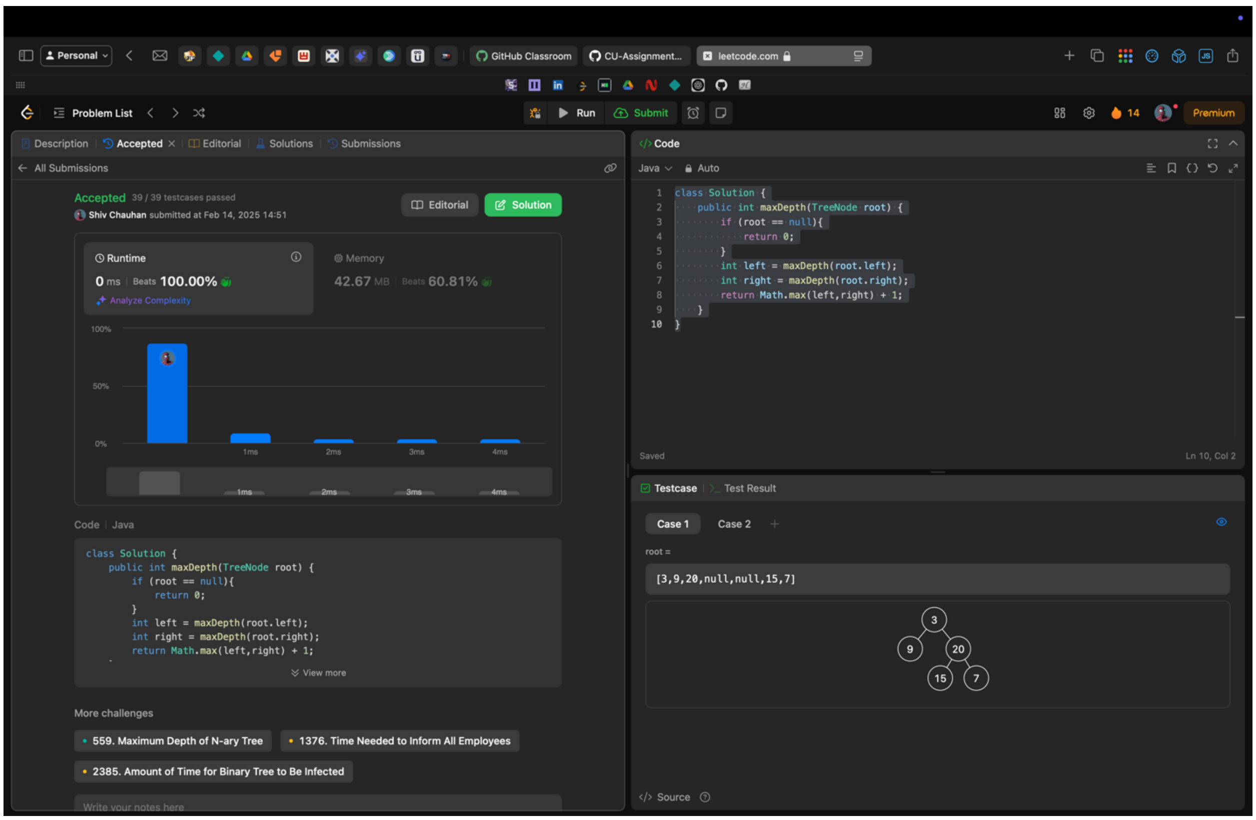
Symmetric Tree

```java
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);
    }
}
```



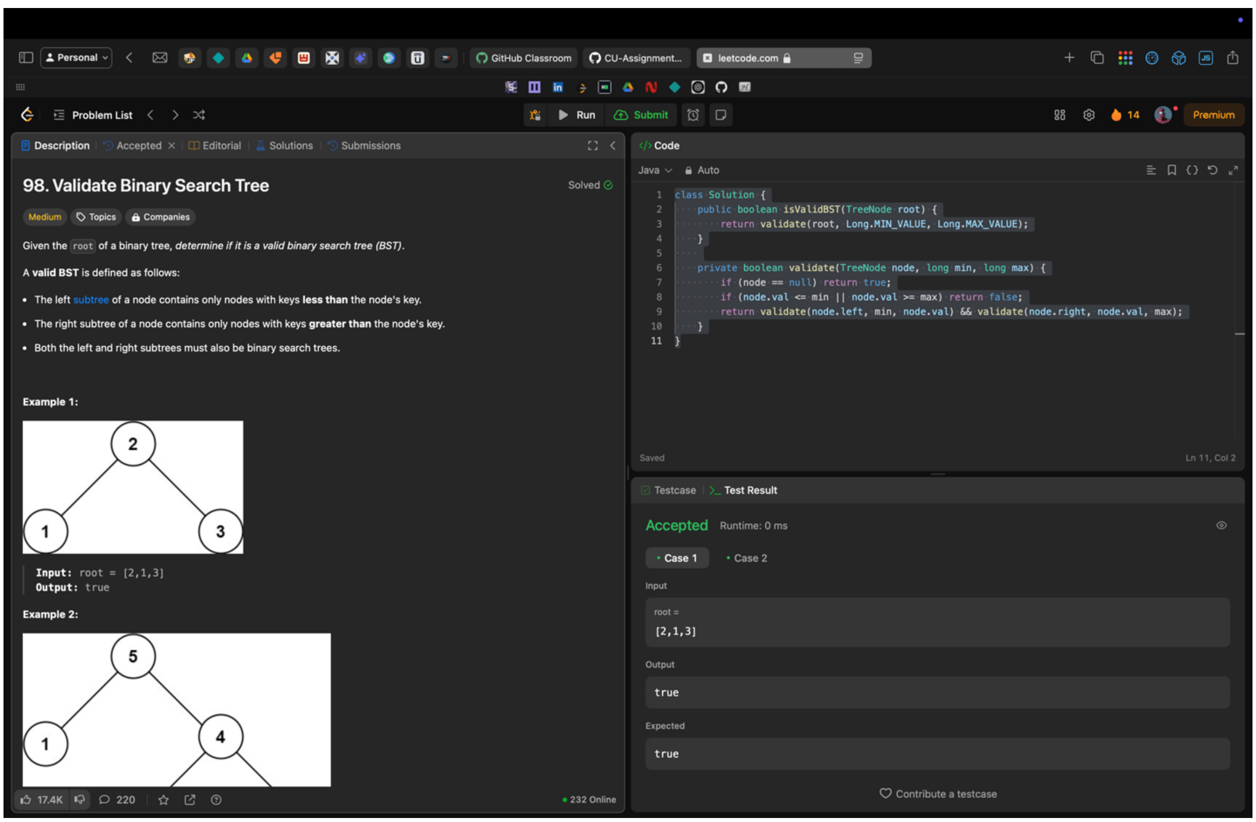
Maximum Depth of Binary Tree

```java
class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null){
            return 0;
        }
        int left = maxDepth(root.left);
        int right = maxDepth(root.right);
        return Math.max(left,right) + 1;
    }
}
```



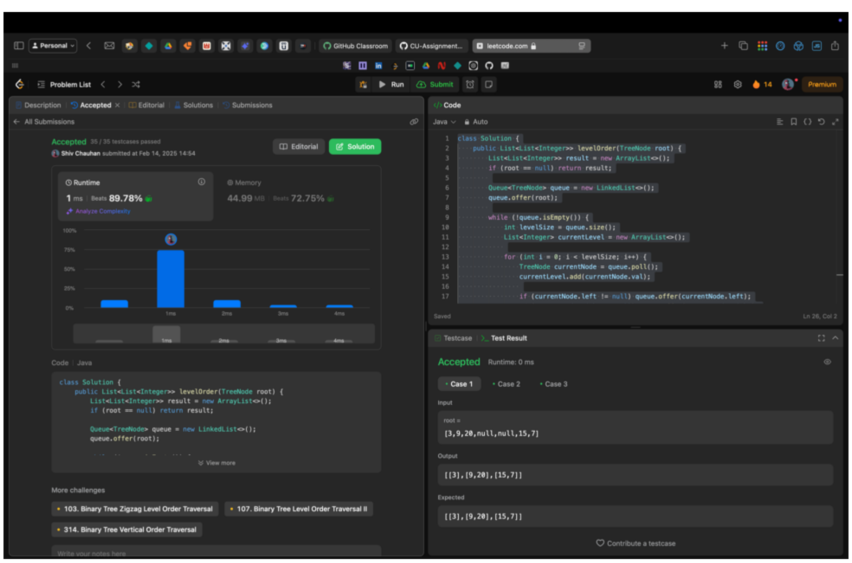
Validate Binary Search Tree

```java
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;
        if (node.val <= min || node.val >= max) return false;
        return validate(node.left, min, node.val) && validate(node.right, 
node.val, max);
    }
}
```



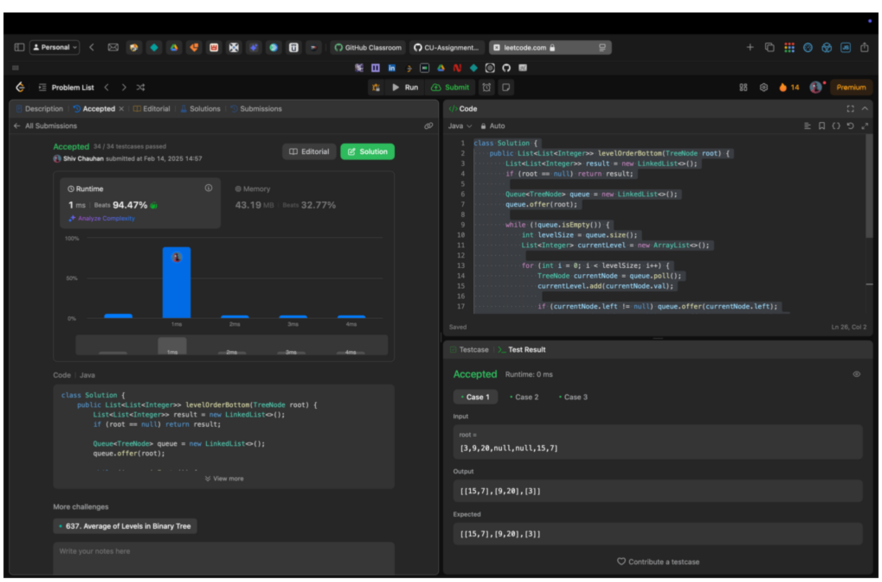
Binary Tree Level Order Traversal

```java
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;
    }
}
```



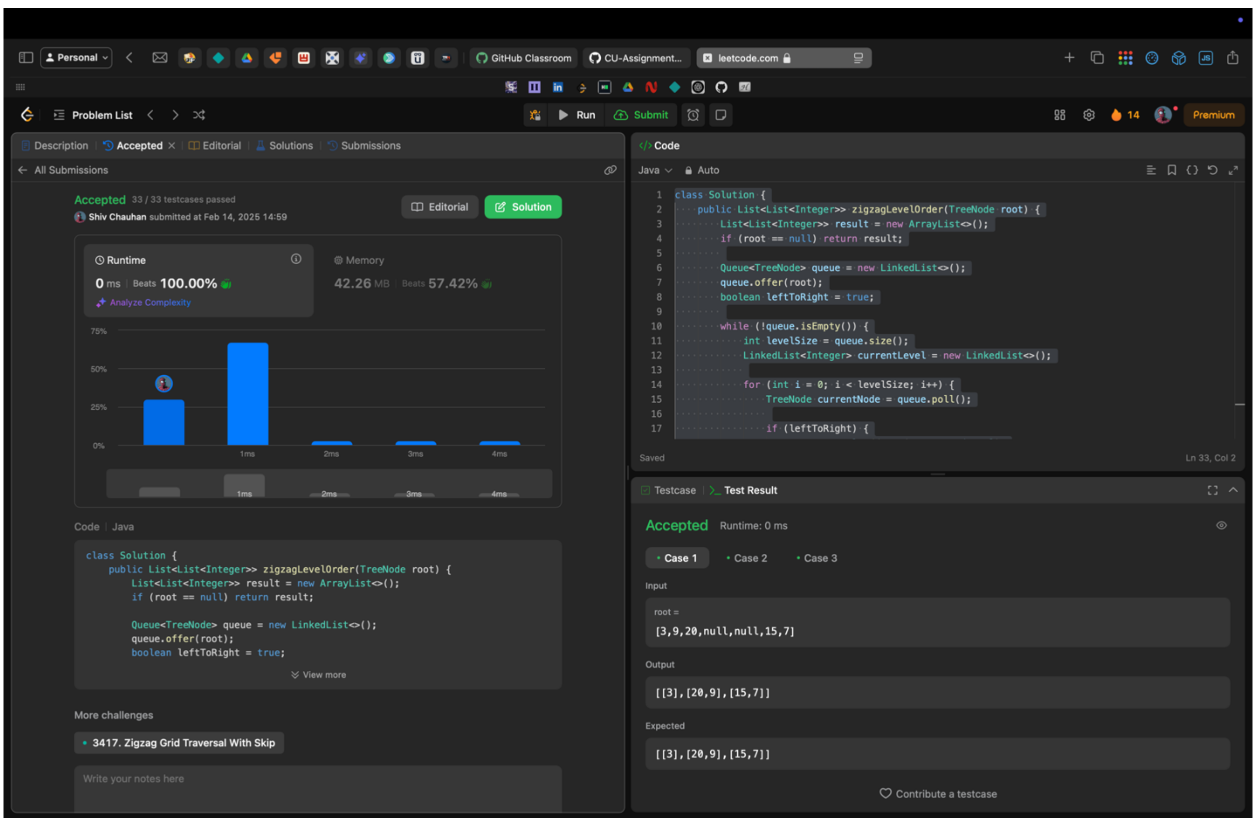
Binary Tree Level Order Traversal II

```java
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        List<List<Integer>> result = new LinkedList<>();
        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(0, currentLevel);
        }

        return result;
    }
}
```



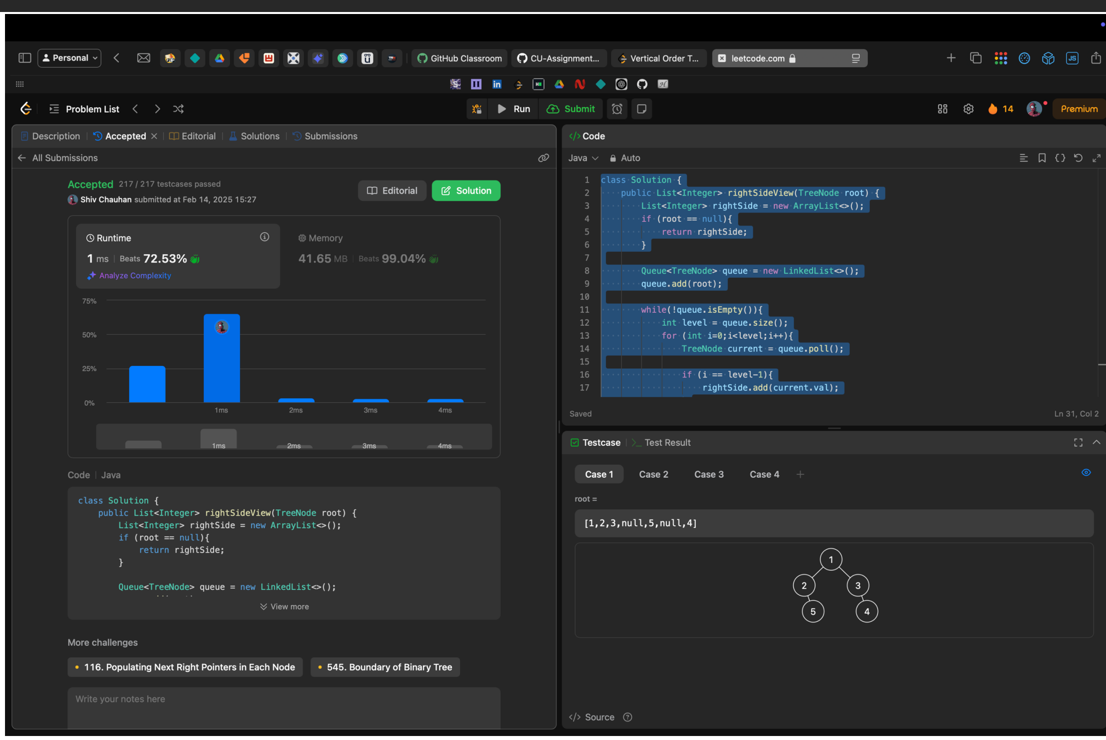
Binary Tree ZigZag Order Traversal

```java
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 levelSize = queue.size();
            LinkedList<Integer> currentLevel = new LinkedList<>();

            for (int i = 0; i < levelSize; i++) {
                TreeNode currentNode = queue.poll();

                if (leftToRight) {
                    currentLevel.addLast(currentNode.val);
                } else {
                    currentLevel.addFirst(currentNode.val);
                }

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

            result.add(currentLevel);
            leftToRight = !leftToRight;
        }

        return result;
    }
}
```



Binary Tree Right Side View

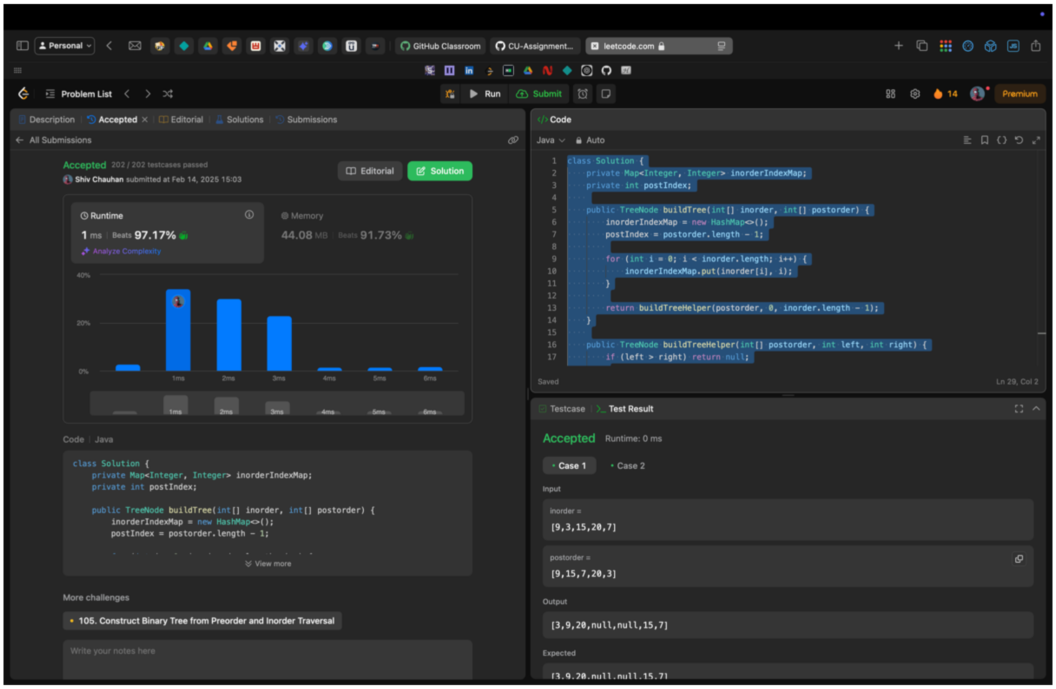
```java
class Solution {
    public List<Integer> rightSideView(TreeNode root) {
        List<Integer> rightSide = new ArrayList<>();
        if (root == null){
            return rightSide;
        }

        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);

        while(!queue.isEmpty()){
            int level = queue.size();
            for (int i=0;i<level;i++){
                TreeNode current = queue.poll();

                if (i == level-1){
                    rightSide.add(current.val);
                }

                if (current.left != null){
                    queue.add(current.left);
                }

                if (current.right != null){
                    queue.add(current.right);
                }
            }
        }
        return rightSide;
    }
}
```



Siddharth 22BCS15779

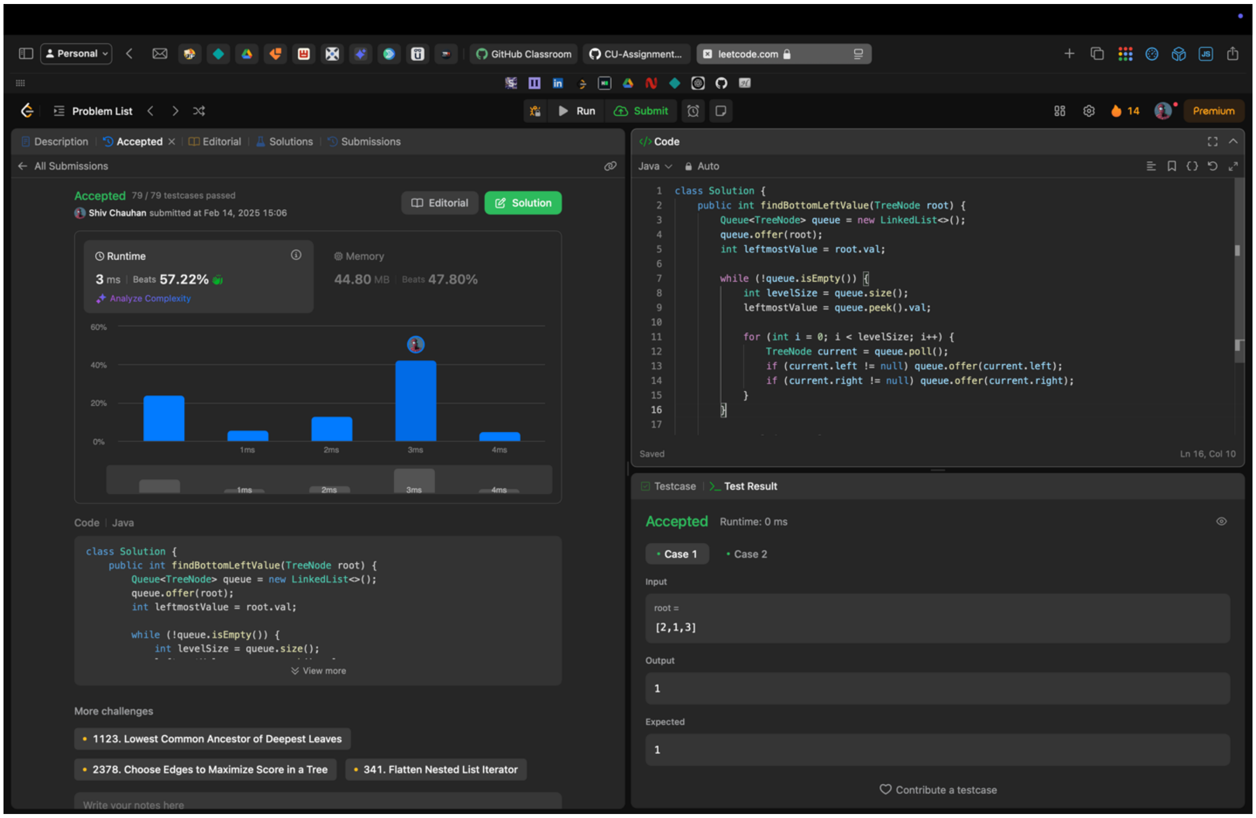
Construct Binary Tree from Inorder and Postorder Traversal

```
class Solution {
    private Map<Integer, Integer> inorderIndexMap;
    private int postIndex;

    public TreeNode buildTree(int[] inorder, int[] postorder) {
        inorderIndexMap = new HashMap<>();
        postIndex = postorder.length - 1;

        for (int i = 0; i < inorder.length; i++) {
            inorderIndexMap.put(inorder[i], i);
        }

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

    public TreeNode buildTreeHelper(int[] postorder, int left, int right) {
        if (left > right) return null;

        int rootValue = postorder[postIndex--];
        TreeNode root = new TreeNode(rootValue);

        int inorderIndex = inorderIndexMap.get(rootValue);

        root.right = buildTreeHelper(postorder, inorderIndex + 1, right);
        root.left = buildTreeHelper(postorder, left, inorderIndex - 1);

        return root;
    }
}
```



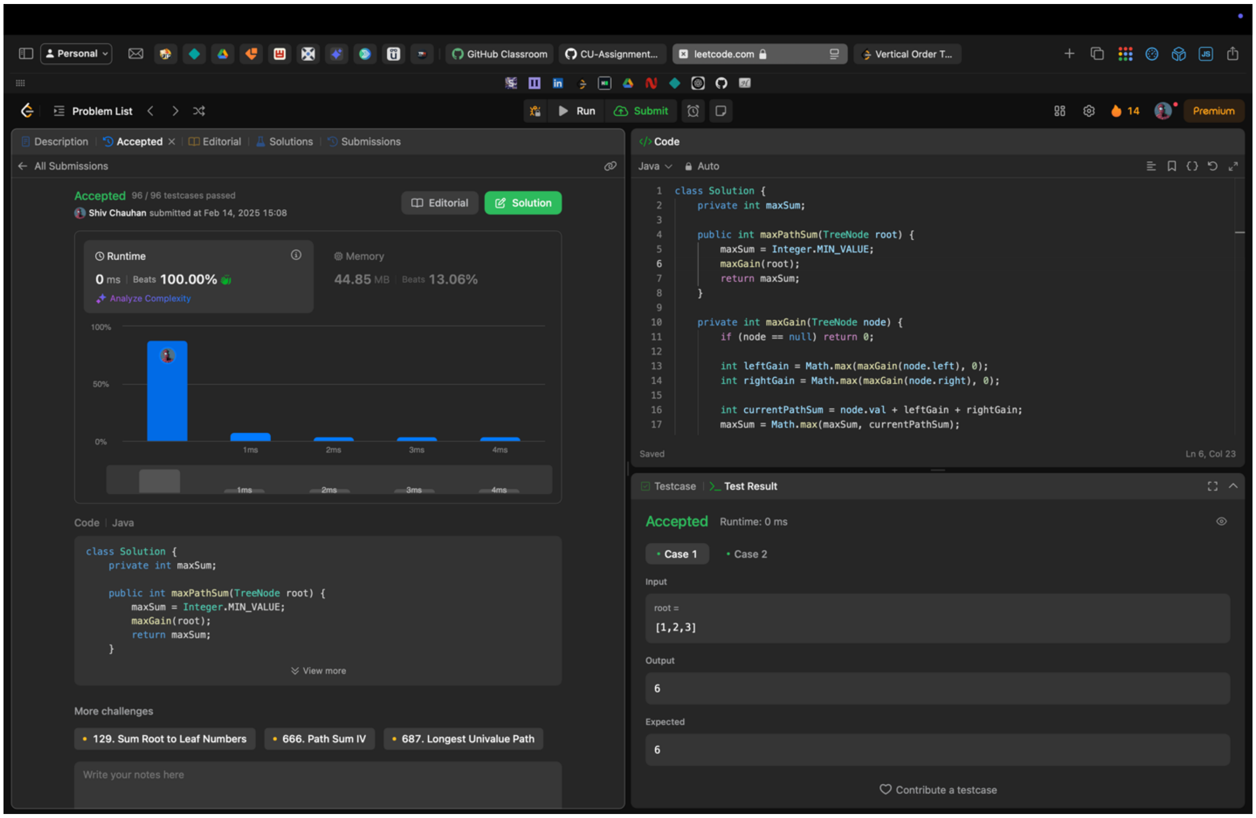
Find Bottom Left Tree Value

```
class Solution {
    public int findBottomLeftValue(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        int leftmostValue = root.val;

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            leftmostValue = queue.peek().val;

            for (int i = 0; i < levelSize; i++) {
                TreeNode current = queue.poll();
                if (current.left != null) queue.offer(current.left);
                if (current.right != null) queue.offer(current.right);
            }
        }

        return leftmostValue;
    }
}
```



Binary Tree Maximum Path Sum

```java
class Solution {
    private int maxSum;

    public int maxPathSum(TreeNode root) {
        maxSum = Integer.MIN_VALUE;
        maxGain(root);
        return maxSum;
    }

    private int maxGain(TreeNode node) {
        if (node == null) return 0;

        int leftGain = Math.max(maxGain(node.left), 0);
        int rightGain = Math.max(maxGain(node.right), 0);

        int currentPathSum = node.val + leftGain + rightGain;
        maxSum = Math.max(maxSum, currentPathSum);

        return node.val + Math.max(leftGain, rightGain);
    }
}
```



Vertical Order Traversal of Binary Tree

```
class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}

class Solution {
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        TreeMap<Integer, TreeMap<Integer, PriorityQueue<Integer>>> map = new
TreeMap<>();
        Queue<Pair<TreeNode, int[]>> queue = new LinkedList<>();
        queue.offer(new Pair<>(root, new int[]{0, 0}));

        while (!queue.isEmpty()) {
            Pair<TreeNode, int[]> pair = queue.poll();
            TreeNode node = pair.getKey();
            int col = pair.getValue()[0], row = pair.getValue()[1];

            map.putIfAbsent(col, new TreeMap<>());
            map.get(col).putIfAbsent(row, new PriorityQueue<>());
            map.get(col).get(row).offer(node.val);

            if (node.left != null) queue.offer(new Pair<>(node.left, new
int[]{col - 1, row + 1}));
            if (node.right != null) queue.offer(new Pair<>(node.right, new
int[]{col + 1, row + 1}));
        }

        List<List<Integer>> result = new ArrayList<>();
        for (TreeMap<Integer, PriorityQueue<Integer>> colMap : map.values())
{
            List<Integer> colList = new ArrayList<>();
            for (PriorityQueue<Integer> nodes : colMap.values()) {
                while (!nodes.isEmpty()) {
```

```
                        colList.add(nodes.poll());
                    }
                }
                result.add(colList);
            }

            return result;
        }
}
```