DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CHANDIGARH UNIVERSITY
Discover. Learn. Empower.

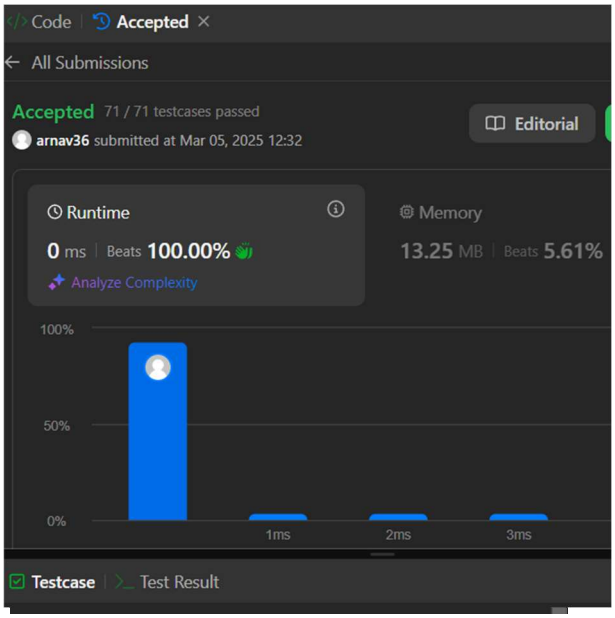
ASSIGNMENT 3

STUDENT NAME: Arnav Jain
UID: 22BCS15161
BRANCH: CSE
SECTION: 22BCS_FL_IOT_601A
SEMESTER: 6
DATE OF SUBMISSION: 4/3/25
SUBJECT NAME: AP LAB -2
SUBJECT CODE: 22CSP-351

LEET CODE QUESTIONS :

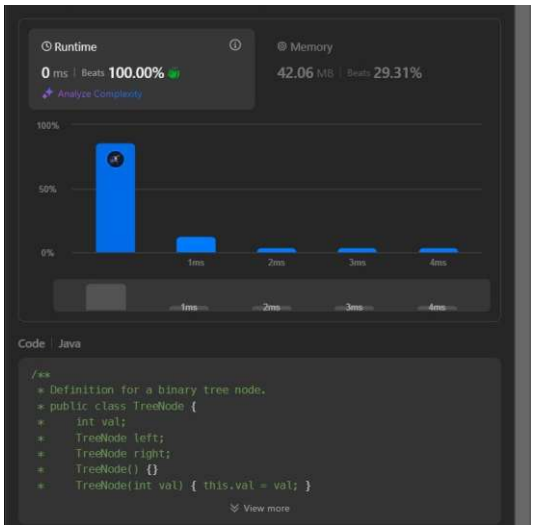
94.BINARY TREE INORDER TRAVERSAL

```
import java.util.*;
class Solution {
    public List<Integer> inorderTraversal(TreeNode
root) {
        List<Integer> result = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        TreeNode curr = root;
        while (curr != null || !stack.isEmpty()) {
            while (curr != null) {
                stack.push(curr);
                curr = curr.left;
            }
            curr = stack.pop();
            result.add(curr.val);
            curr = curr.right;
        }
        return result;
    }
}
```



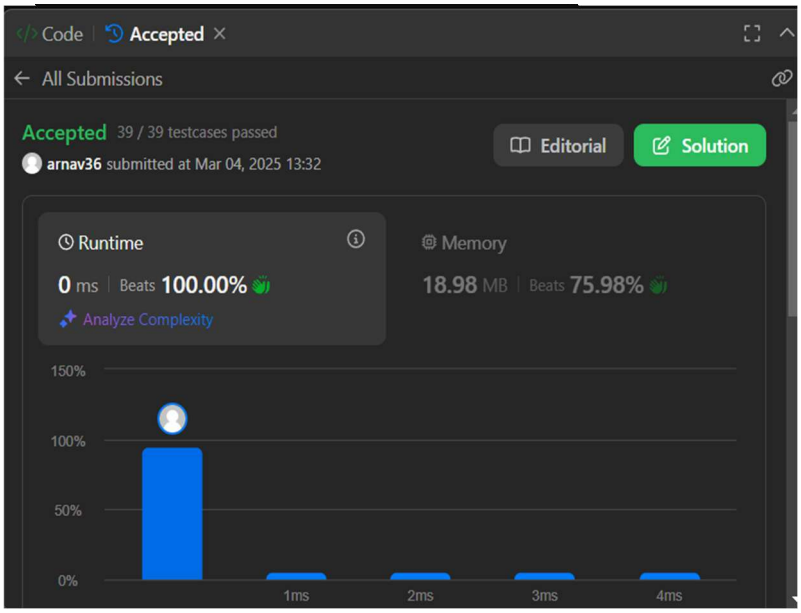
101. SYMMETRIC TREE

```
class Solution {
    public boolean isSymmetric(TreeNode root) {
        return root == null || isMirror(root.left, root.right);
    }
    private boolean isMirror(TreeNode t1, TreeNode t2) {
        if (t1 == null || t2 == null) return t1 == t2;
        return (t1.val == t2.val) && isMirror(t1.left, t2.right) && isMirror(t1.right, t2.left);
    }
}
```



104. MAXIMUM DEPTH OF BINARY TREE

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

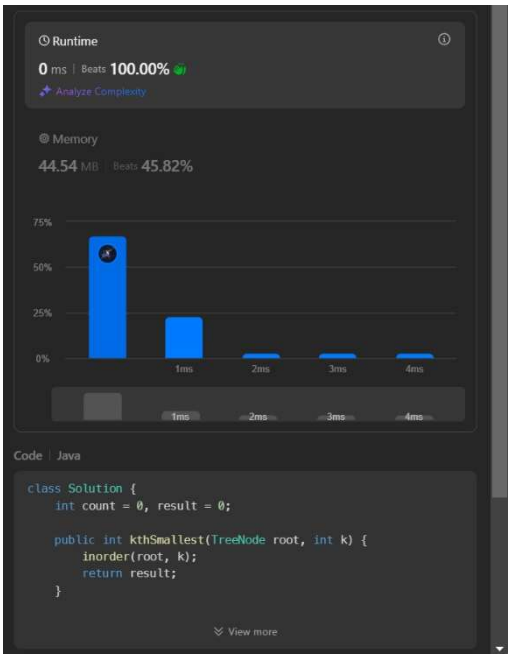


98.VALIDATE BINARY SEARCH TREE

```
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);
    }
}
```

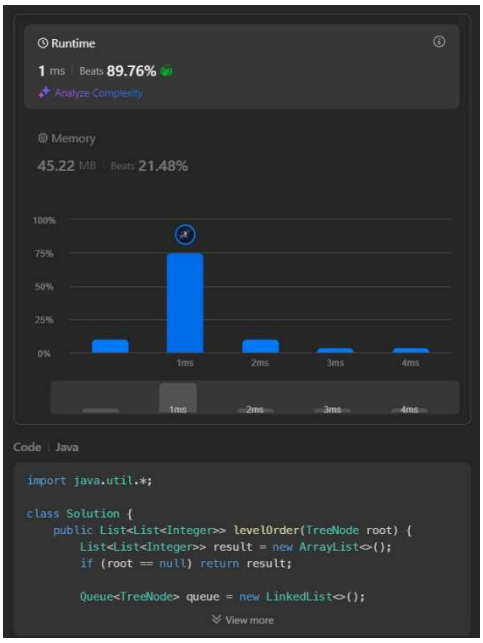
230.KTH SMALLEST ELEMENT IN A BST

```
class Solution {
    int count = 0, result = 0;
    public int kthSmallest(TreeNode root, int k) {
        inorder(root, k);
        return result;
    }
    private void inorder(TreeNode node, int k) {
        if (node == null) return;
        inorder(node.left, k);
        count++;
        if (count == k) result = node.val;
        inorder(node.right, k);
    }
}
```



102. BINARY TREE LEVEL ORDER TRAVERSAL

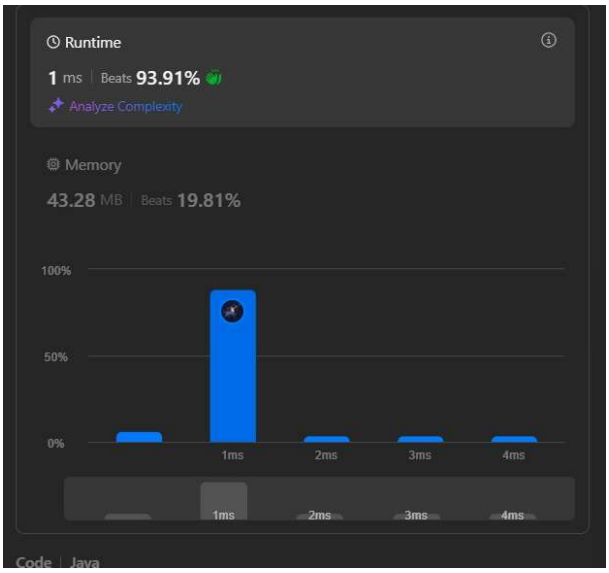
```
import java.util.*;
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) return result;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            List<Integer> level = new ArrayList<>();
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                level.add(node.val);
                if (node.left != null) queue.add(node.left);
```



```
    if (node.right != null) queue.add(node.right);
            }
            result.add(level);
        }
        return result;
    }
}
```

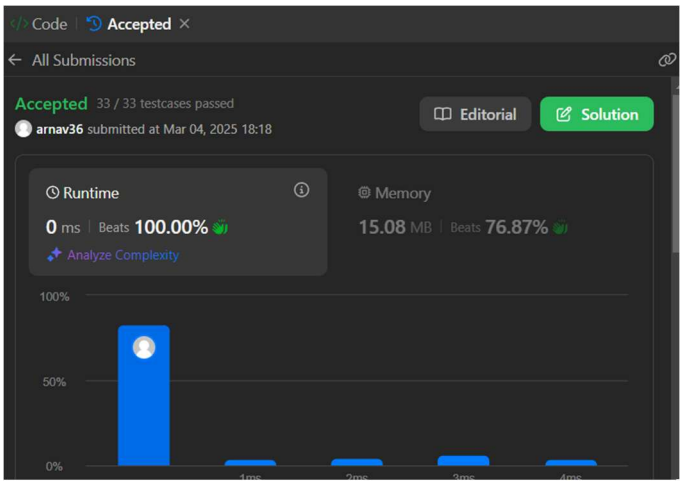
107.BINARY TREE LEVEL ORDER TRAVERSAL II

```
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        LinkedList<List<Integer>> result = new LinkedList<>();
        if (root == null) return result;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            List<Integer> level = new ArrayList<>();
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                level.add(node.val);
                if (node.left != null) queue.add(node.left);
                if (node.right != null)
queue.add(node.right);
            }
            result.addFirst(level);
        }
        return result;
    }
}
```



103. BINARY TREE ZIGZAG LEVEL ORDER TRAVERSAL

```
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.add(root);
        boolean leftToRight = true;
        while (!queue.isEmpty()) {
            LinkedList<Integer> level = new
LinkedList<>();
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                if (leftToRight) level.addLast(node.val);
                else level.addFirst(node.val);
                if (node.left != null)
queue.add(node.left);
                if (node.right != null)
queue.add(node.right);
            }
            result.add(level);
            leftToRight = !leftToRight;
        }
        return result;
    }
}
```



199.BINARY TREE RIGHT SIDE VIEW

```
class Solution {
    public List<Integer> rightSideView(TreeNode
root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) return result;
        Queue<TreeNode> queue = new
LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                if (i == size - 1) result.add(node.val);
                if (node.left != null) queue.add(node.left);
                if (node.right != null) queue.add(node.right);
            }
        }
        return result;
    }
}
```

106.CONSTRUCT BINARY TREE FROM INORDER AND POSTORDER TRAVERSAL

```
import java.util.*;
class Solution {
    private int postIndex;
    private Map<Integer, Integer> inorderMap;
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        postIndex = postorder.length - 1;
        inorderMap = new HashMap<>();
        for (int i = 0; i < inorder.length; i++) {
            inorderMap.put(inorder[i], i);
        }
```

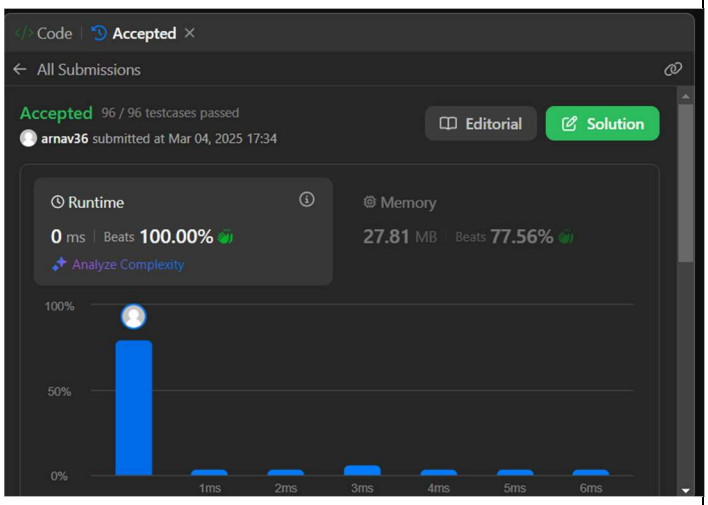
```
        return build(0, inorder.length - 1, postorder);
    }
    private TreeNode build(int left, int right, int[]
postorder) {
        if (left > right) return null;
        int val = postorder[postIndex--];
        TreeNode node = new TreeNode(val);
        node.right = build(inorderMap.get(val) + 1, right,
postorder);
        node.left = build(left, inorderMap.get(val) - 1,
postorder);
        return node;
    }
}
```

513.FIND BOTTOM LEFT TREE VALUE

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

        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();
            if (node.right != null) queue.add(node.right);
            if (node.left != null) queue.add(node.left);
            bottomLeft = node.val;
        }
        return bottomLeft;
    }
}
```

124. BINARY TREE MAXIMUM PATH SUM




```
class Solution {
    int maxSum = Integer.MIN_VALUE;
    public int maxPathSum(TreeNode root) {
        maxGain(root);
        return maxSum;
    }
    private int maxGain(TreeNode node) {
        if (node == null) return 0;
        int left = Math.max(maxGain(node.left), 0);
        int right = Math.max(maxGain(node.right),
0);
        maxSum = Math.max(maxSum, left + right +
node.val);
        return node.val + Math.max(left, right);
    }
}
```

987.VERTICAL ORDER TRAVERSAL OF A BINARY TREE

```
import java.util.*;
class Solution {
    class Tuple {
        TreeNode node;
        int row, col;
        public Tuple(TreeNode node, int row, int col) {
            this.node = node;
            this.row = row;
            this.col = col;
        }
    }
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        TreeMap<Integer, TreeMap<Integer, PriorityQueue<Integer>>> map = new
TreeMap<>();
```

```
        Queue<Tuple> queue = new LinkedList<>();
        queue.offer(new Tuple(root, 0, 0)
        while (!queue.isEmpty()) {
            Tuple t = queue.poll();
            TreeNode node = t.node;
            int row = t.row, col = t.col;
            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 Tuple(node.left, row + 1, col - 1));
            if (node.right != null) queue.offer(new Tuple(node.right, row + 1, col + 1));
        }
        List<List<Integer>> result = new ArrayList<>();
        for (TreeMap<Integer, PriorityQueue<Integer>> rowMap : map.values()) {
            List<Integer> vertical = new ArrayList<>();
            for (PriorityQueue<Integer> nodes : rowMap.values()) {
                while (!nodes.isEmpty()) {
                    vertical.add(nodes.poll());
                }
            }
            result.add(vertical);
        }
        return result;
    }
}
```