

Q1: [Binary Tree Inorder Traversal](#)

Given the root of a binary tree, return *the inorder traversal of its nodes' values*.

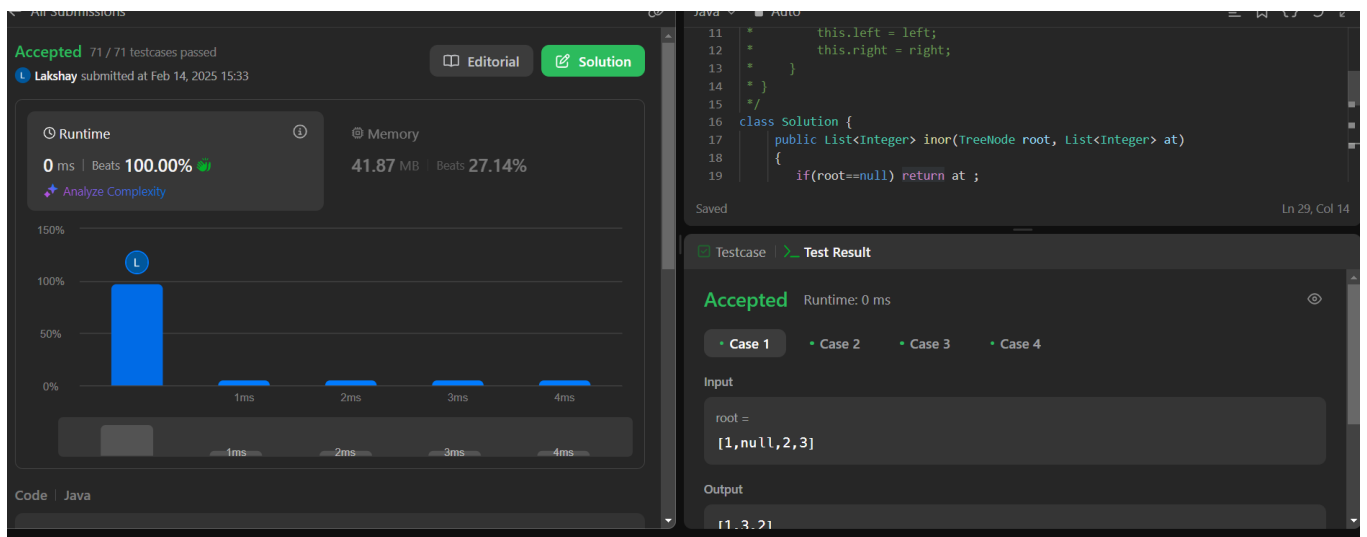
Example 1:

Input: root = [1,null,2,3]

Output: [1,3,2]

SOLUTION:

```
class Solution {  
    public List<Integer> inor(TreeNode root, List<Integer> at)  
    {  
        if(root==null) return at ;  
  
        inor(root.left,at);  
        at.add(root.val);  
        inor(root.right,at);  
        return at;  
    }  
    public List<Integer> inorderTraversal(TreeNode root) {  
        List<Integer> at = new ArrayList<Integer>();  
        return inor(root,at);  
    }  
}
```



Q2: [Symmetric Tree](#)

Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

SOLUTION:

```

class Solution {
    public boolean isSymmetric(TreeNode root) {
        if (root == null) {
            return true;
        }
        return isMirror(root.left, root.right);
    }

    private boolean isMirror(TreeNode node1, TreeNode node2) {
        if (node1 == null && node2 == null) {
            return true;
        }
        if (node1 == null || node2 == null) {
            return false;
        }

```

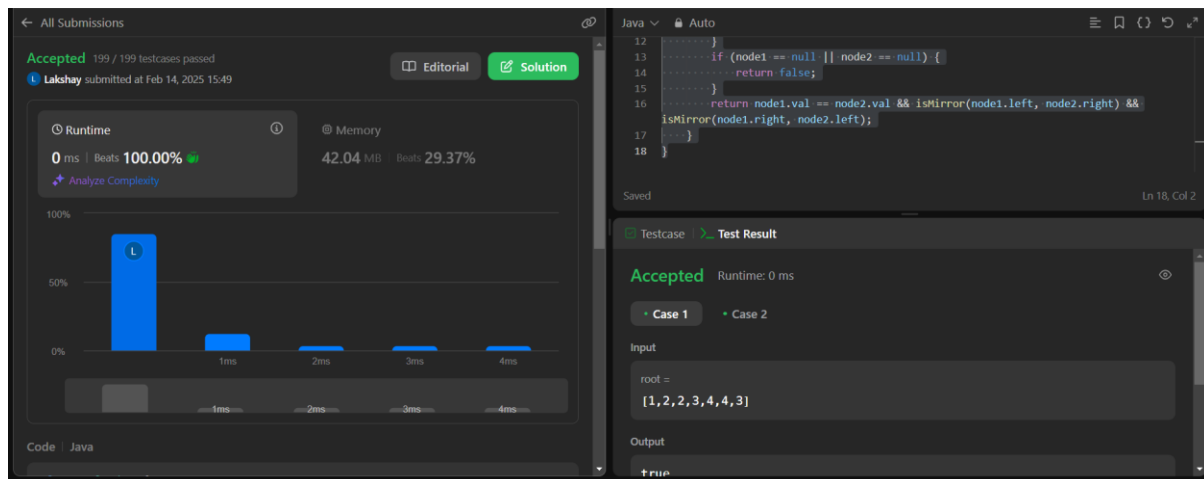
```

return node1.val == node2.val && isMirror(node1.left, node2.right) && isMirror(node1.right,
node2.left);

}

}

```



Q3: [Maximum Depth of Binary Tree](#)

Given the root of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

SOLUTION:

```

class TreeNode {

    int val;

    TreeNode left;

    TreeNode right;

    TreeNode(int x) { val = x; }

}

public class Solution {

    public int maxDepth(TreeNode root) {

        return treeHeight(root);

    }

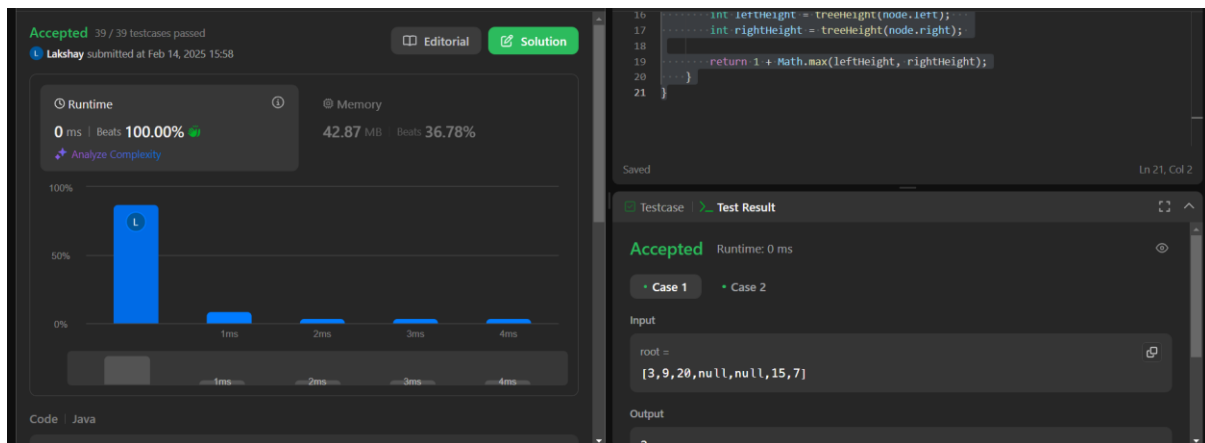
}

```

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

    int leftHeight = treeHeight(node.left);
    int rightHeight = treeHeight(node.right);

    return 1 + Math.max(leftHeight, rightHeight);
}
}
```



Q4: [Validate Binary Search Tree](#)

Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

- The left

subtree

of a node contains only nodes with keys **less than** the node's key.

- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

SOLUTION:

```
class TreeNode {
    int val;
```

```

TreeNode left;

TreeNode right;

TreeNode(int x) { val = x; }

}

public class Solution {

    public boolean isValidBST(TreeNode root) {

        return helper(root, Long.MIN_VALUE, Long.MAX_VALUE);

    }

    private boolean helper(TreeNode node, long min, long max) {

        if (node == null) return true;

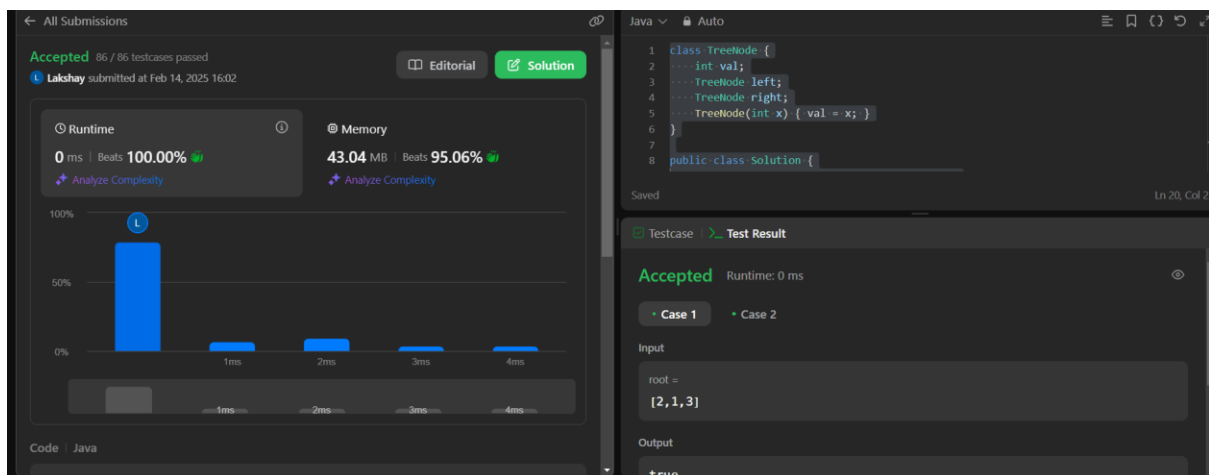
        if (node.val <= min || node.val >= max) return false;

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

    }

}

```

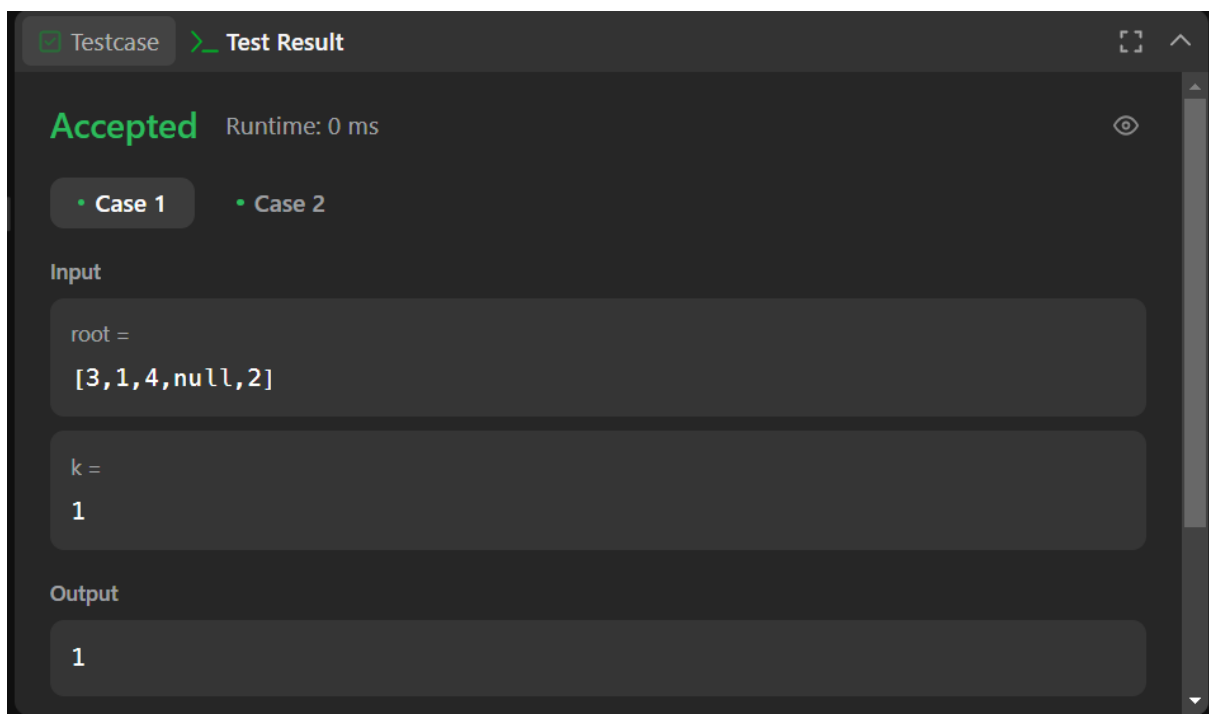


Q5: [Kth Smallest Element in a BST](#)

Given the root of a binary search tree, and an integer k , return the k^{th} smallest value (**1-indexed**) of all the values of the nodes in the tree.

SOLUTION:

```
class Solution {  
    public void inorder(TreeNode root, ArrayList<Integer> arr) {  
        if (root == null) {  
            return;  
        }  
        inorder(root.left, arr);  
        arr.add(root.val);  
        inorder(root.right, arr);  
    }  
  
    public int kthSmallest(TreeNode root, int k) {  
        ArrayList<Integer> fin = new ArrayList<>();  
        inorder(root, fin);  
        return fin.get(k - 1);  
    }  
}
```



The screenshot shows a dark-themed interface for testing a solution. At the top, there are two tabs: "Testcase" and "Test Result", with "Test Result" being the active tab. Below the tabs, the status "Accepted" is displayed in green, followed by "Runtime: 0 ms". There are two sub-tabs: "Case 1" and "Case 2", with "Case 1" being selected. Under the "Input" section, there are two text boxes: the first contains "root =" and "[3,1,4,null,2]", and the second contains "k =" and "1". Under the "Output" section, there is a text box containing "1".

Q6: [Binary Tree Level Order Traversal](#)

Given the root of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

SOLUTION:

```
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.offer(root);
        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            List<Integer> level = new ArrayList<>();
            for (int i = 0; i < levelSize; i++) {
                TreeNode current = queue.poll();
                level.add(current.val);
                if (current.left != null) queue.offer(current.left);
                if (current.right != null) queue.offer(current.right);
            }
            result.add(level);
        }
        return result;
    }
}
```

☒ Testcase
 ☒ Test Result

Accepted Runtime: 0 ms

• Case 1
 • Case 2
 • Case 3

Input

```
root =
[3,9,20,null,null,15,7]
```

Output

```
[[3],[9,20],[15,7]]
```

Q7: [Binary Tree Level Order Traversal II](#)

Given the root of a binary tree, return *the bottom-up level order traversal of its nodes' values*. (i.e., from left to right, level by level from leaf to root).

SOLUTION:

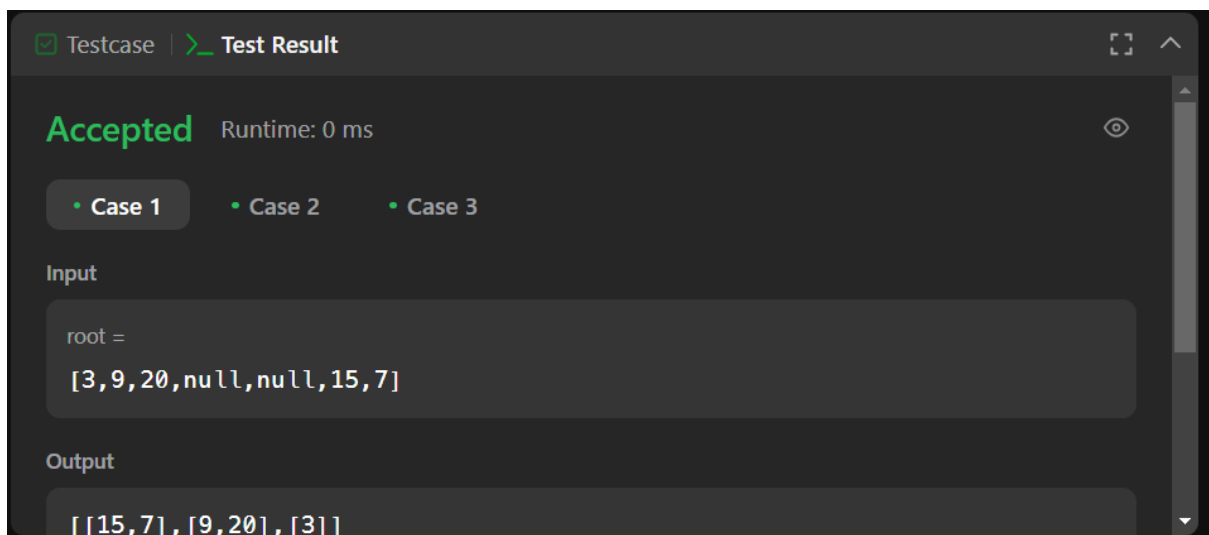
```
import java.util.*;
```

```
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;
    }
}
```

```
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
```



```
List<List<Integer>> levels = new ArrayList<>();  
  
if (root == null) return levels;  
  
Queue<TreeNode> queue = new LinkedList<>();  
queue.offer(root);  
  
while (!queue.isEmpty()) {  
    int n = queue.size();  
  
    List<Integer> level = new ArrayList<>();  
  
    for (int i = 0; i < n; i++) {  
        TreeNode node = queue.poll();  
  
        level.add(node.val);  
  
        if (node.left != null) queue.offer(node.left);  
        if (node.right != null) queue.offer(node.right);  
    }  
  
    levels.add(0, level);  
}  
  
return levels;  
}  
}
```

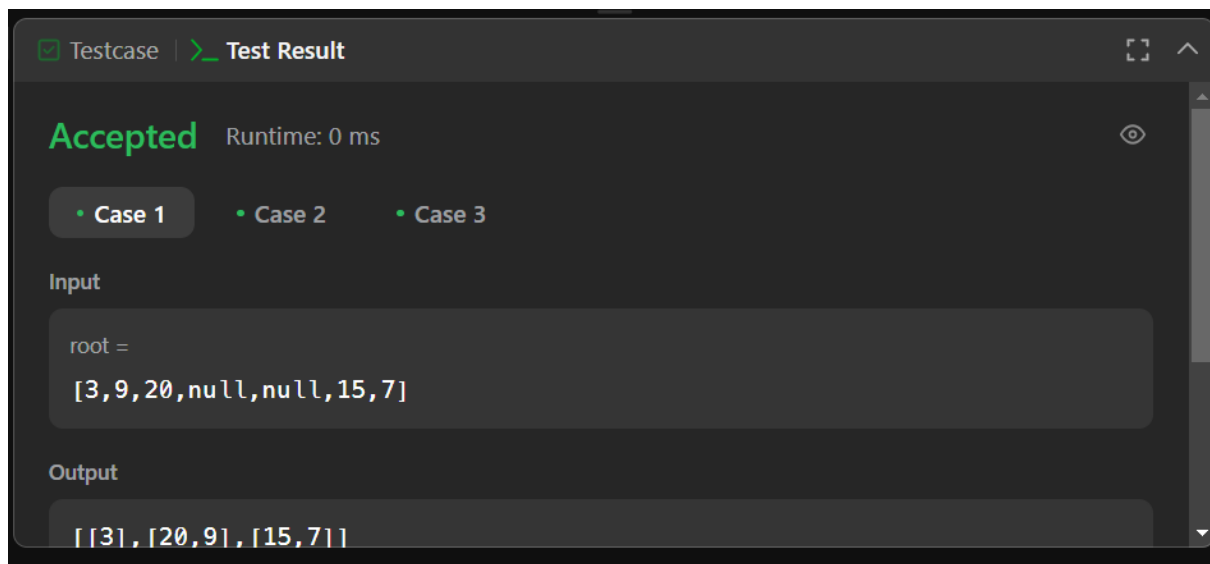


Q8: [. Binary Tree Zigzag Level Order Traversal](#)

Given the root of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

SOLUTION:

```
public class Solution {  
    public List<List<Integer>> zigzagLevelOrder(TreeNode root)  
    {  
        List<List<Integer>> sol = new ArrayList<>();  
        travel(root, sol, 0);  
        return sol;  
    }  
  
    private void travel(TreeNode curr, List<List<Integer>> sol, int level)  
    {  
        if(curr == null) return;  
  
        if(sol.size() <= level)  
        {  
            List<Integer> newLevel = new LinkedList<>();  
            sol.add(newLevel);  
        }  
  
        List<Integer> collection = sol.get(level);  
        if(level % 2 == 0) collection.add(curr.val);  
        else collection.add(0, curr.val);  
  
        travel(curr.left, sol, level + 1);  
        travel(curr.right, sol, level + 1);  
    } }  
}
```



Q9: [Binary Tree Right Side View](#)

Given the root of a binary tree, imagine yourself standing on the **right side** of it, return *the values of the nodes you can see ordered from top to bottom*.

SOLUTION:

```
import java.util.*;
```

```
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);

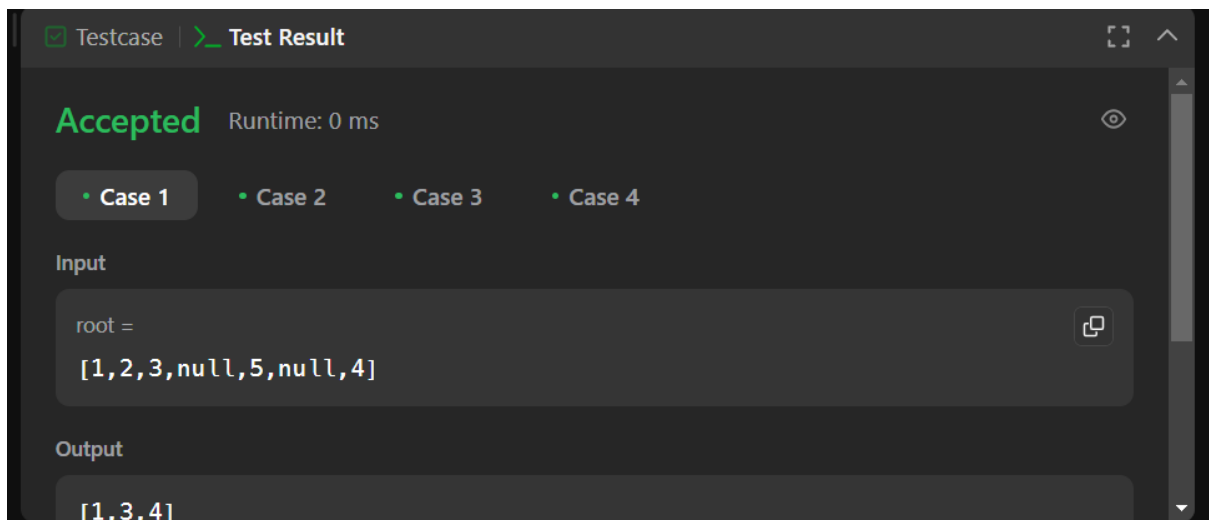
        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            for (int i = 0; i < levelSize; i++) {
                TreeNode node = queue.poll();
                if (i == levelSize - 1) result.add(node.val);
                if (node.left != null) queue.add(node.left);
            }
        }
    }
}
```

```

        if (node.right != null) queue.add(node.right);
    }
}

return result;
}
}

```



Q10: [Construct Binary Tree from Inorder and Postorder Traversal](#)

Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return *the binary tree*.

SOLUTION:

```
import java.util.HashMap;
```

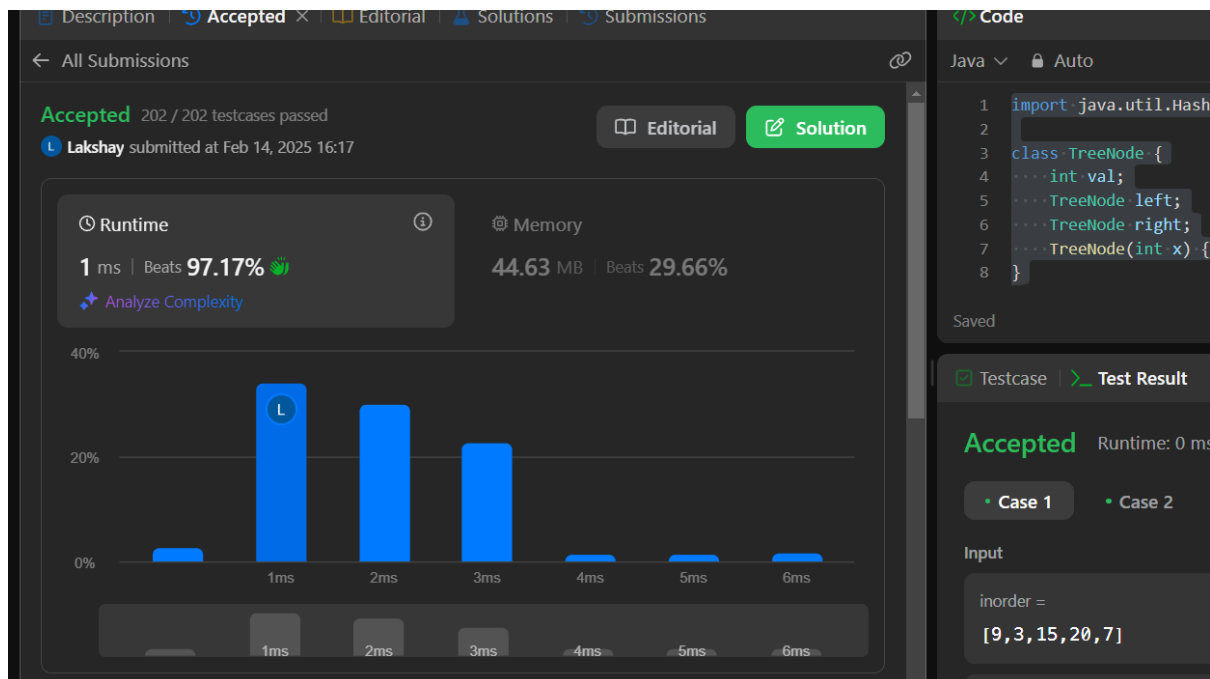
```

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

```

```
class Solution {  
  
    public TreeNode buildTree(int[] inorder, int[] postorder) {  
  
        HashMap<Integer, Integer> rec = new HashMap<>();  
  
        for (int i = 0; i < inorder.length; i++) {  
            rec.put(inorder[i], i);  
        }  
  
        return helper(inorder, postorder, 0, inorder.length - 1, 0, postorder.length - 1, rec);  
    }  
  
    private TreeNode helper(int[] inorder, int[] postorder,  
                            int inStart, int inEnd,  
                            int postStart, int postEnd,  
                            HashMap<Integer, Integer> rec) {  
  
        if (inStart > inEnd || postStart > postEnd) return null;  
  
        int val = postorder[postEnd];  
        TreeNode root = new TreeNode(val);  
        int idx = rec.get(val);  
        int leftSubtreeSize = idx - inStart;  
  
        root.left = helper(inorder, postorder,  
                          inStart, idx - 1,  
                          postStart, postStart + leftSubtreeSize - 1,  
                          rec);  
  
        root.right = helper(inorder, postorder,  
                           idx + 1, inEnd,  
                           postStart + leftSubtreeSize, postEnd - 1,  
                           rec);  
  
        return root;  
    }  
}
```

```
}  
}
```



Q11 : [Find Bottom Left Tree Value](#)

Given the root of a binary tree, return the leftmost value in the last row of the tree.

SOLUTION:

```
public class Solution {  
    public int findBottomLeftValue(TreeNode root) {  
        if (root == null)  
            return 0;
```

```
        Queue<TreeNode> q = new LinkedList<>();  
        q.add(root);  
        int leftNode = 0;
```

```
        while (!q.isEmpty()) {  
            int size = q.size();  
            leftNode = q.peek().val;  
  
            for (int i = 0; i < size; i++) {
```

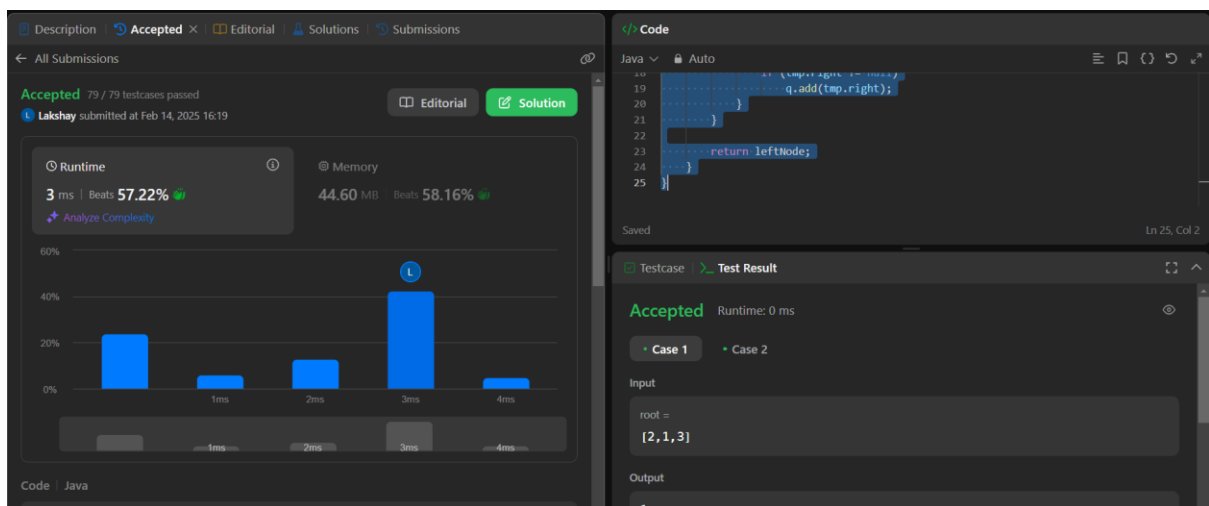
```

TreeNode tmp = q.poll();

if (tmp.left != null)
    q.add(tmp.left);
if (tmp.right != null)
    q.add(tmp.right);
}
}

return leftNode;
}
}

```



Q12: [Binary Tree Maximum Path Sum](#)

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

Given the root of a binary tree, return *the maximum path sum of any non-empty path*.

SOLUTION:

```

class Solution {
    private int maxSum = Integer.MIN_VALUE;

```

```

public int maxPathSum(TreeNode root) {

    helper(root);

    return maxSum;

}

private int helper(TreeNode node) {

    if (node == null) {

        return 0;

    }

    int leftMaxPath = Math.max(helper(node.left), 0);

    int rightMaxPath = Math.max(helper(node.right), 0);

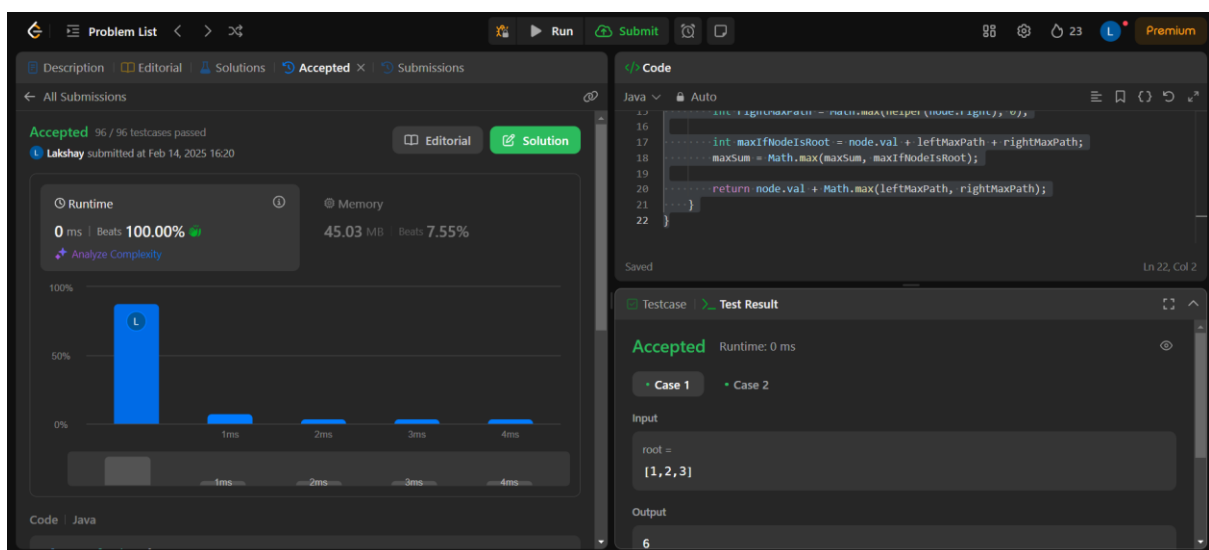
    int maxIfNodeIsRoot = node.val + leftMaxPath + rightMaxPath;

    maxSum = Math.max(maxSum, maxIfNodeIsRoot);

    return node.val + Math.max(leftMaxPath, rightMaxPath);

}
}

```



The screenshot displays a code editor interface for a Java solution. The top bar shows the 'Problem List' and 'Run' buttons. The left sidebar indicates the solution is 'Accepted' with 96/96 testcases passed, submitted by 'Lakshay' on Feb 14, 2025. The runtime is 0 ms (Beats 100.00%) and memory usage is 45.03 MB (Beats 7.55%). A bar chart shows the runtime performance across different test cases. The main editor shows the Java code for the 'maxPathSum' function. The right sidebar shows the 'Test Result' for 'Case 1' with input 'root = [1, 2, 3]' and output '6'.

Q13: [Vertical Order Traversal of a Binary Tree](#)

Given the root of a binary tree, calculate the **vertical order traversal** of the binary tree.

For each node at position (row, col), its left and right children will be at positions (row + 1, col - 1) and (row + 1, col + 1) respectively. The root of the tree is at (0, 0).

The **vertical order traversal** of a binary tree is a list of top-to-bottom orderings for each column index starting from the leftmost column and ending on the rightmost column. There may be multiple nodes in the same row and same column. In such a case, sort these nodes by their values.

Return *the vertical order traversal of the binary tree*.

SOLUTION:

```
/**
 * Definition for a binary tree node.
 * public 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;
 *     }
 * }
 */

class Solution {
    TreeMap<Integer,TreeMap<Integer,ArrayList<Integer>>> map=new TreeMap<>();

    public List<List<Integer>> verticalTraversal(TreeNode root) {
        List<List<Integer>> res=new ArrayList<>();

        if(root==null) return res;

        dfs(root,0,0);

        for(Map.Entry<Integer,TreeMap<Integer,ArrayList<Integer>>> entry: map.entrySet()){
```

```
        TreeMap<Integer,ArrayList<Integer>> level=entry.getValue();

        ArrayList<Integer> list=new ArrayList<>();

        for(Map.Entry<Integer,ArrayList<Integer>> subentry: level.entrySet()){

            ArrayList<Integer> sub=subentry.getValue();

            Collections.sort(sub);

            list.addAll(sub);

        }

        res.add(list);

    }

    return res;

}

public void dfs(TreeNode root,int col,int level){

    if(root==null) return;

    if(!map.containsKey(col)){

        map.put(col,new TreeMap<>());

    }

    if(!map.get(col).containsKey(level)){

        map.get(col).put(level,new ArrayList<>());

    }

    map.get(col).get(level).add(root.val);

    dfs(root.left,col-1,level+1);

    dfs(root.right,col+1,level+1);

}

}
```

