

94. Binary Tree Inorder Traversal

The screenshot displays the LeetCode interface for problem 94. On the left, the problem description states: "Given the `root` of a binary tree, return the *inorder traversal* of its nodes' values." An example is provided with input `root = [1,null,2,3]` and output `[1,3,2]`. A diagram shows a binary tree with root 1, left child 3, and right child 2. On the right, the submission results show "Accepted" for 70/70 testcases, with a runtime of 0 ms (beats 100.00%) and memory usage of 41.54 MB (beats 80.66%).

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

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

104. Maximum Depth of Binary Tree

104. Maximum Depth of Binary Tree Solved

Easy Topics Companies

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.

Example 1:

```
graph TD; 3((3)) --- 9((9)); 3 --- 20((20)); 20 --- 15((15)); 20 --- 7((7))
```

Runtime: 0 ms | Beats 100.00%
Memory: 42.58 MB | Beats 76.27%

Testcase: Case 1 Case 2 +

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

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

98. Validate Binary Search Tree

98. Validate Binary Search Tree Solved

Medium Topics Companies

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.

Example 1:

```
graph TD; 2((2)) --- 1((1)); 2 --- 3((3))
```

Runtime: 0 ms | Beats 100.00%
Memory: 42.96 MB | Beats 97.85%

Testcase: Case 1 Case 2 +

root = [2, 1, 3]

```

class Solution {
    public static boolean isValidBSTHelper(TreeNode root, TreeNode min, TreeNode
max) {
        if (root == null){
            return true;
        }
        if (min != null && root.val <= min.val) {
            return false;
        } else if (max != null && root.val >= max.val) {
            return false;
        }
        return isValidBSTHelper(root.left, min, root) && isValidBSTHelper(root.right,
root, max);
    }
    public boolean isValidBST(TreeNode root) {
        return isValidBSTHelper(root, null, null);
    }
}

```

230. Kth Smallest Element in a BST

230. Kth Smallest Element in a BST Solved

Medium Topics Companies Hint

Given the `root` of a binary search tree, and an integer `k`, return the `kth` smallest value (1-indexed) of all the values of the nodes in the tree.

Example 1:

```

graph TD
    3((3)) --- 1((1))
    3 --- 4((4))
    1 --- 2((2))

```

Runtime: 0 ms | Beats 100.00% | Memory: 44.55 MB | Beats 46.17%

Testcase: Case 1 Case 2 +

root = [3,1,4,null,2]

```

class Solution {
    int val = 0;
    int idx = 0;
    public void helper(TreeNode root, int k) {
        if (root == null) return;
        kthSmallest(root.left,k);
        if (++idx == k){
            val = root.val;
            return;
        }
        kthSmallest(root.right,k);
    }
}

```

```

    }
    public int kthSmallest(TreeNode root, int k) {
        helper(root,k);
        return val;
    }
}

```

102. Binary Tree Level Order Traversal

102. Binary Tree Level Order Traversal Solved

Medium Topics Companies Hint

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).

Example 1:

```

graph TD
    3((3)) --- 9((9))
    3 --- 20((20))
    9 --- 15((15))
    20 --- 7((7))

```

Runtime: 1 ms | Beats: 89.78% | Memory: 44.90 MB | Beats: 86.83%

Testcase: Case 1 Case 2 Case 3

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

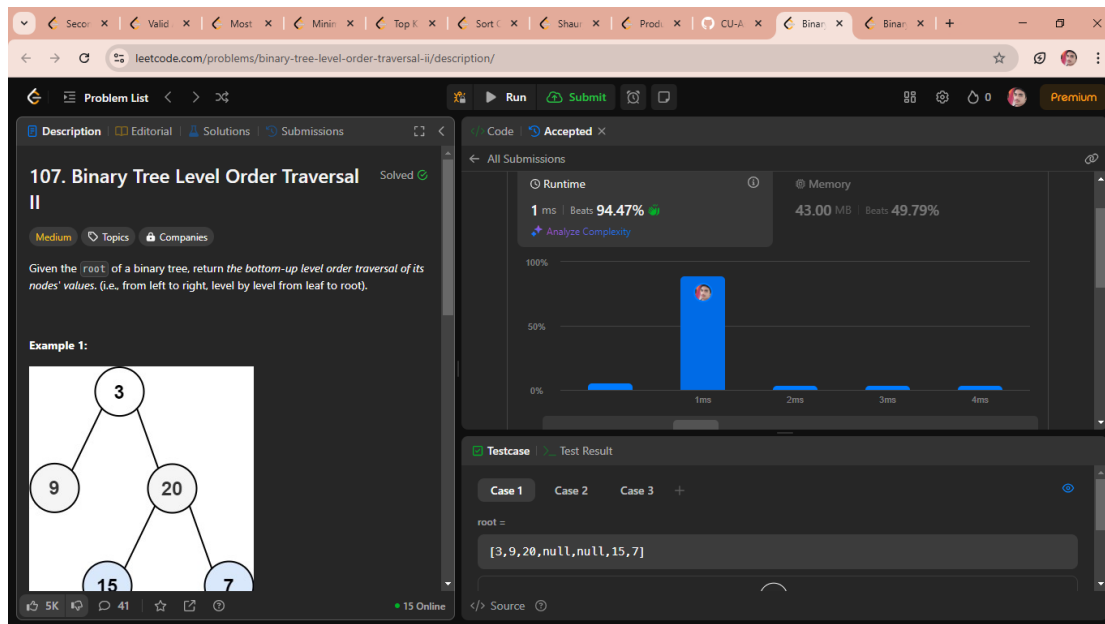
```

class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> list = new LinkedList<>();
        if (root == null) return list;
        Queue<TreeNode> q = new LinkedList<>();
        q.add(root);
        while(!q.isEmpty()) {
            int levelSize = q.size();
            List<Integer> l = new LinkedList<>();
            for (int i=0; i<levelSize; i++) {
                TreeNode curr = q.remove();
                l.add(curr.val);
                if (curr.left != null) {
                    q.add(curr.left);
                }
                if (curr.right != null) {
                    q.add(curr.right);
                }
            }
            list.add(l);
        }
        return list;
    }
}

```

}

107. Binary Tree Level Order Traversal II

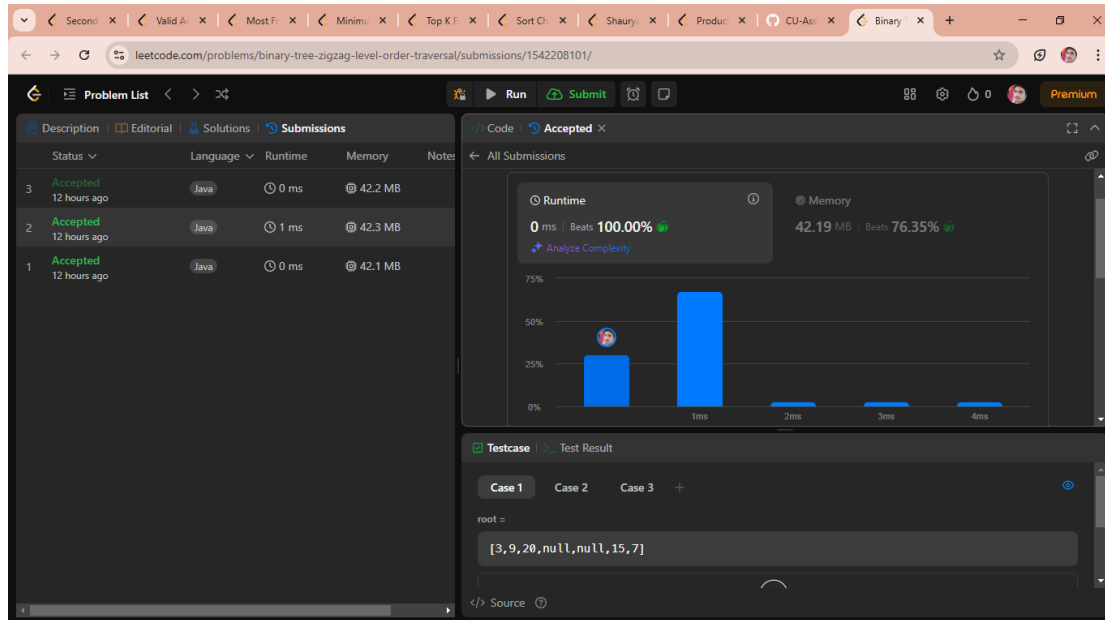


```
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        List<List<Integer>> list = new LinkedList<>();
        if (root == null) return list;
        Queue<TreeNode> q = new LinkedList<>();
        List<Integer> l = new LinkedList<>();
        q.add(root);
        q.add(null);

        while (!q.isEmpty()) {
            TreeNode curr = q.remove();
            if (curr == null) {
                if (q.isEmpty()) {
                    break;
                } else {
                    list.addFirst(l);
                    l = new LinkedList<>();
                    q.add(null);
                }
            } else {
                l.add(curr.val);
                if (curr.left != null) q.add(curr.left);
                if (curr.right != null) q.add(curr.right);
            }
        }
        list.addFirst(l);
        return list;
    }
}
```

}

103. Binary Tree Zigzag Level Order Traversal



```
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> list = new LinkedList<>();
        if (root == null) return list;
        List<Integer> l = new LinkedList<>();
        Queue<TreeNode> q = new LinkedList<>();
        q.add(root);
        q.add(null);
        boolean leftToRight = true;
        while (!q.isEmpty()) {
            TreeNode currNode = q.remove();
            if (currNode == null) {
                list.add(l);
                if (q.isEmpty()) {
                    break;
                } else {
                    l = new LinkedList<>();
                    q.add(null);
                    leftToRight = !leftToRight;
                }
            } else {
                if (leftToRight) {
                    l.add(currNode.val);
                } else {
                    l.addFirst(currNode.val);
                }
                if (currNode.left != null) q.add(currNode.left);
            }
        }
    }
}
```

```
        if (currNode.right != null) q.add(currNode.right);
    }
}
return list;
}
```