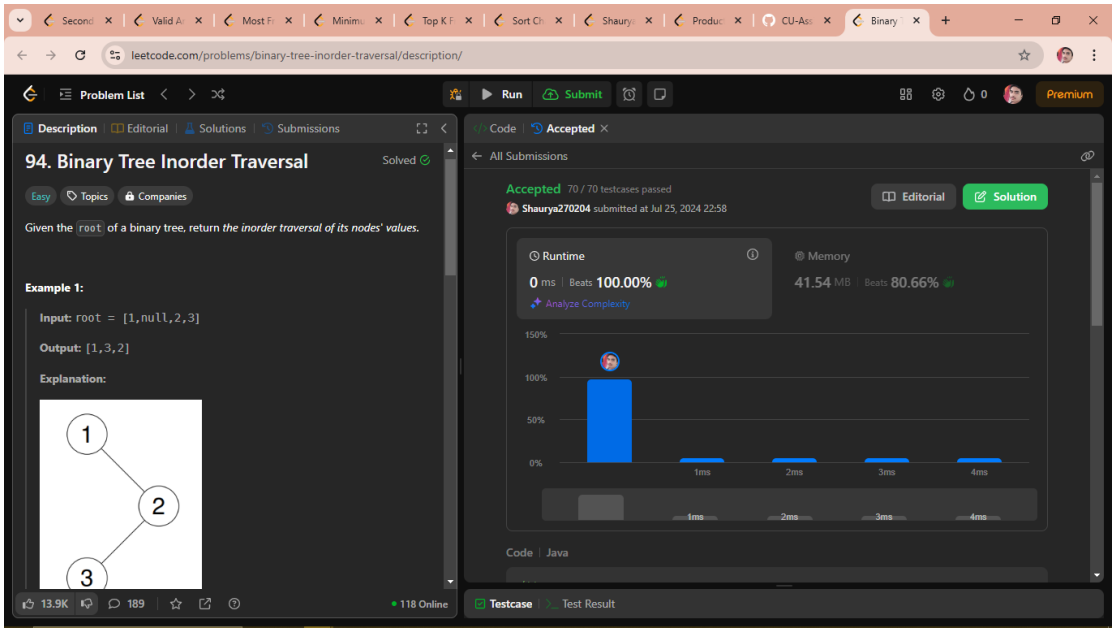
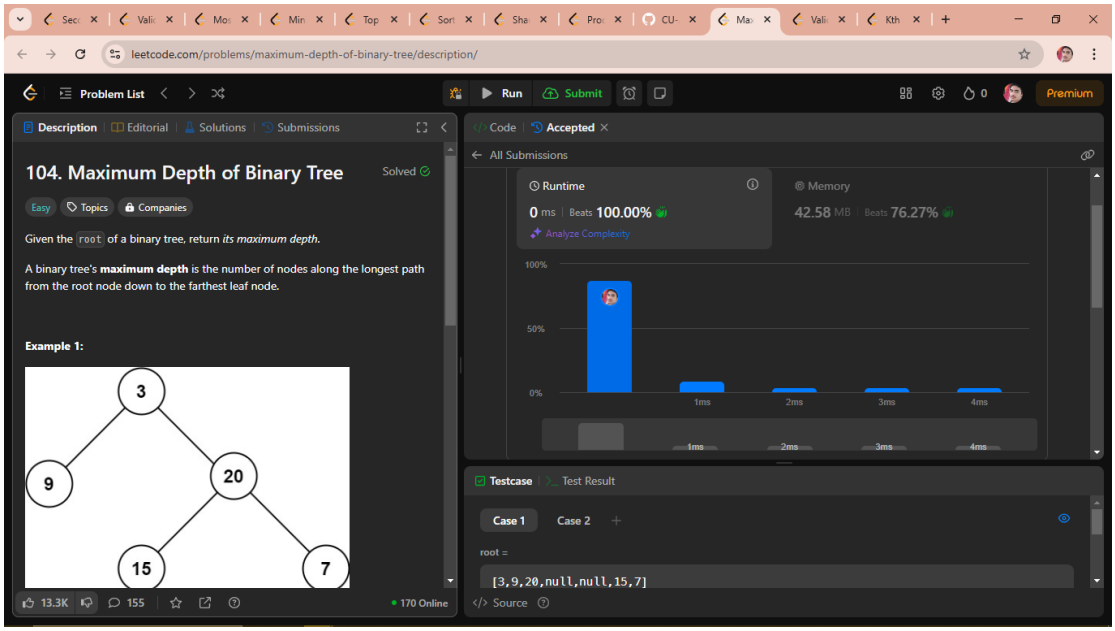
94. Binary Tree Inorder Traversal



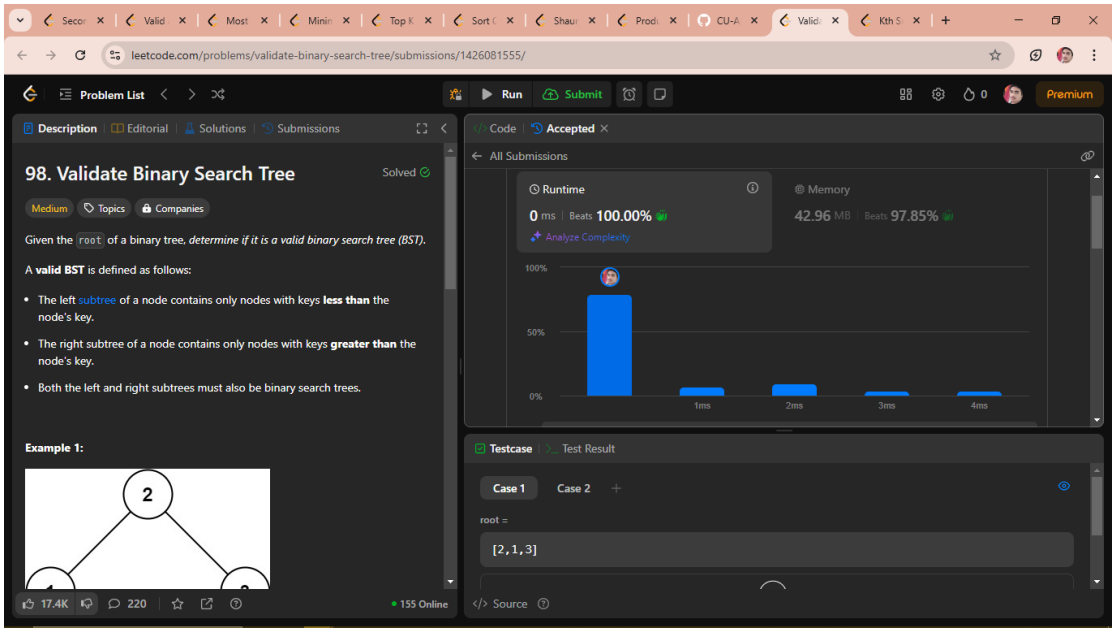

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> list = new ArrayList<>();
        util (root, list);
        return list;
    }

    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



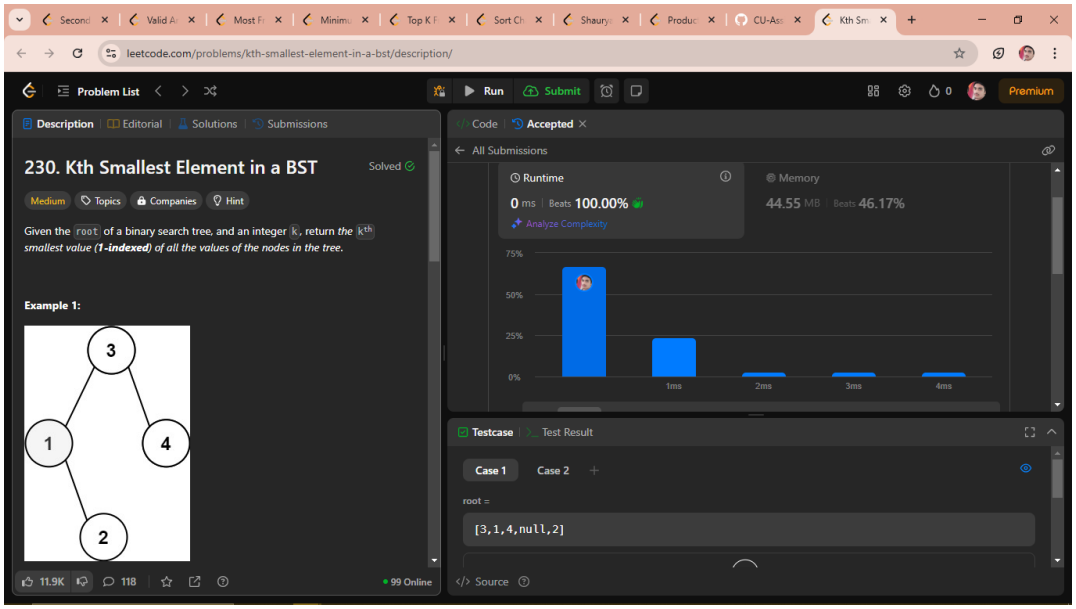

```
class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) return 0;
        return Math.max(maxDepth(root.left),maxDepth(root.right))+1;
    }
}
```

98. Validate Binary Search Tree



```
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);
   }
}
```

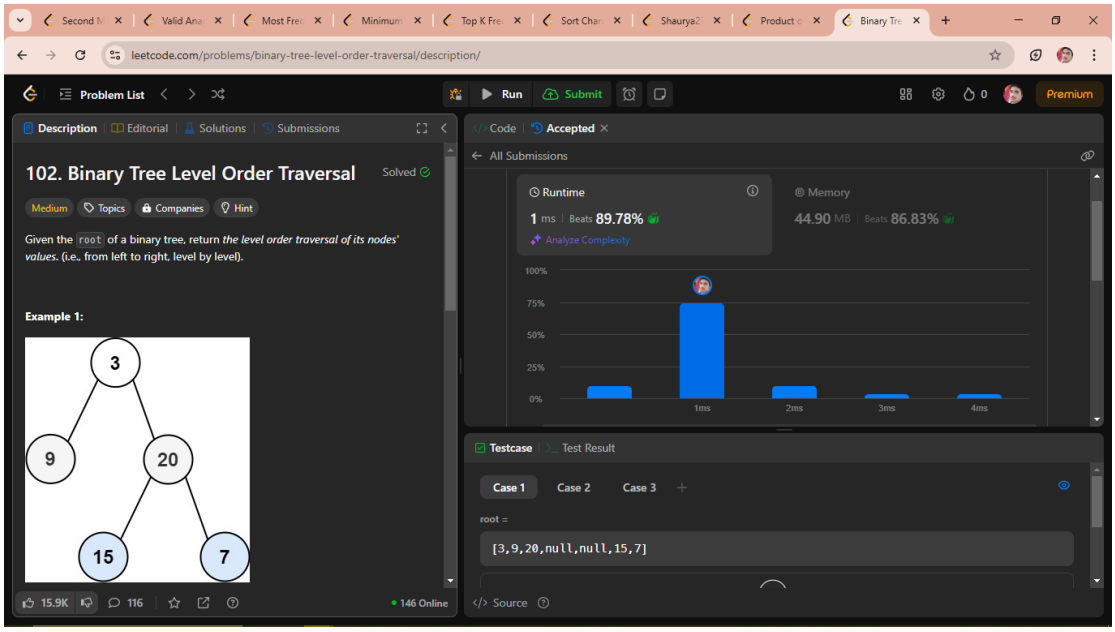
230. Kth Smallest Element in a BST




```
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;
    }
}
```

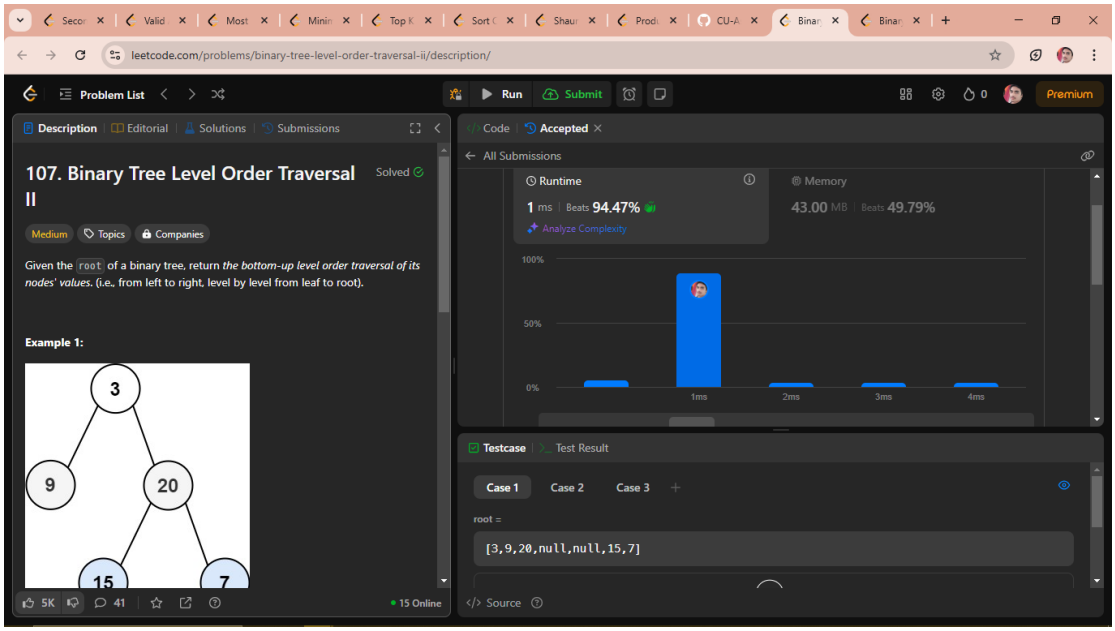
102. Binary Tree Level Order Traversal




```
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;
    }
```

}

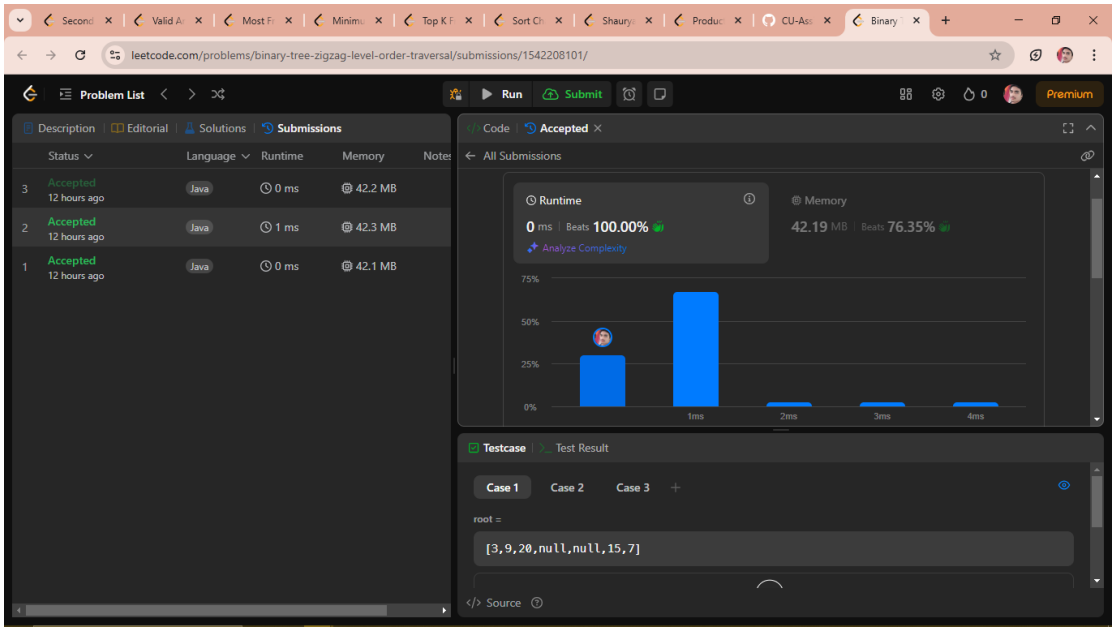
107. Binary Tree Level Order Traversal II




```
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;
    }
}
```