**94.Binary Tree Inorder Traversal**

**class Solution:**

**def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:**

**st = []**

**res = []**

**while root or st:**

**while root:**

**st.append(root)**

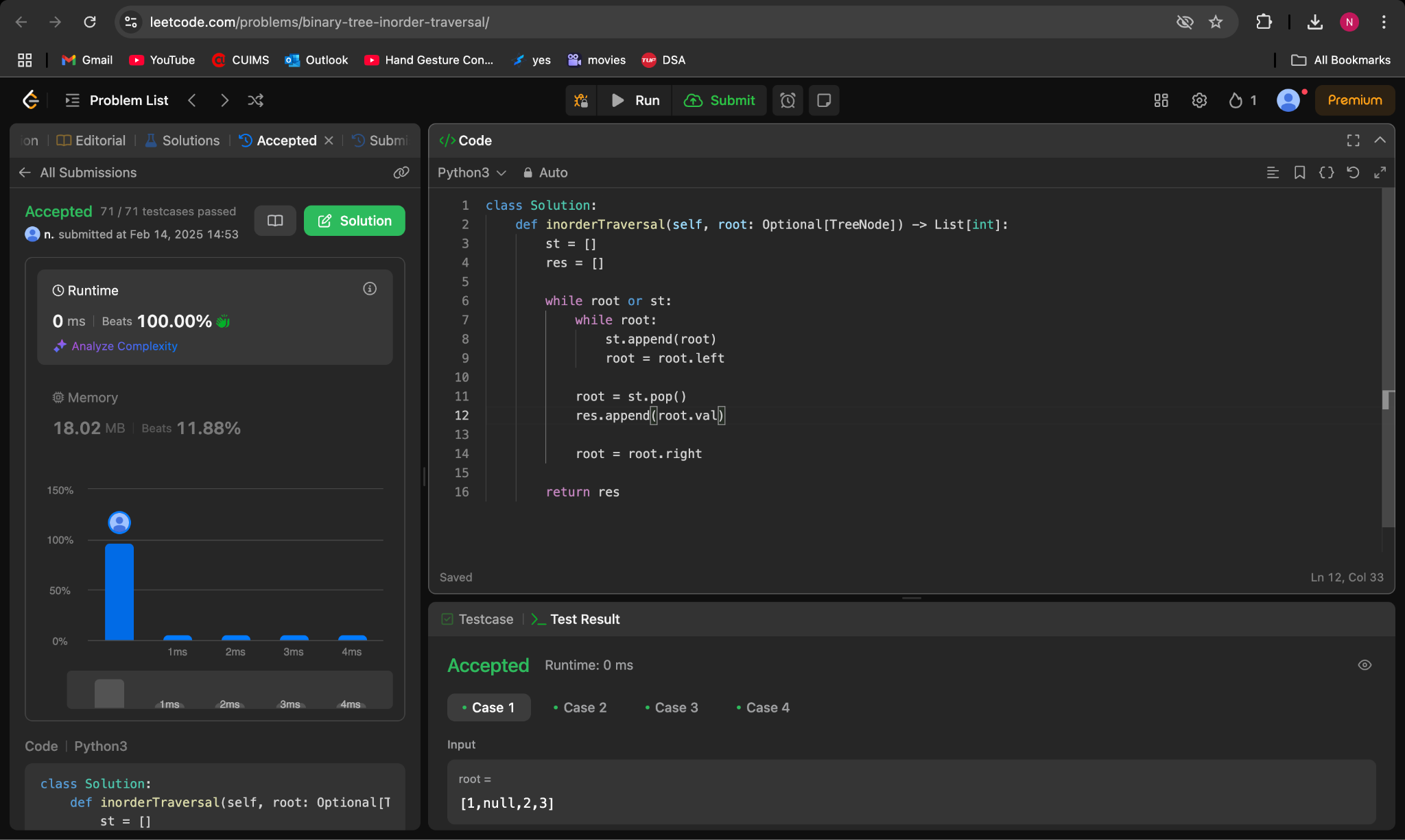
**root = root.left**

**root = st.pop()**

**res.append(root.val)**

**root = root.right**

**return res**

****

**101.Symmetric Tree**

**class Solution(object):**

**def isSymmetric(self, root):**

**if not root:**

**return true;**

**return self.isSame(root.left, root.right)**

**def isSame(self, leftroot, rightroot):**

**if leftroot == None and rightroot == None:**

**return True**

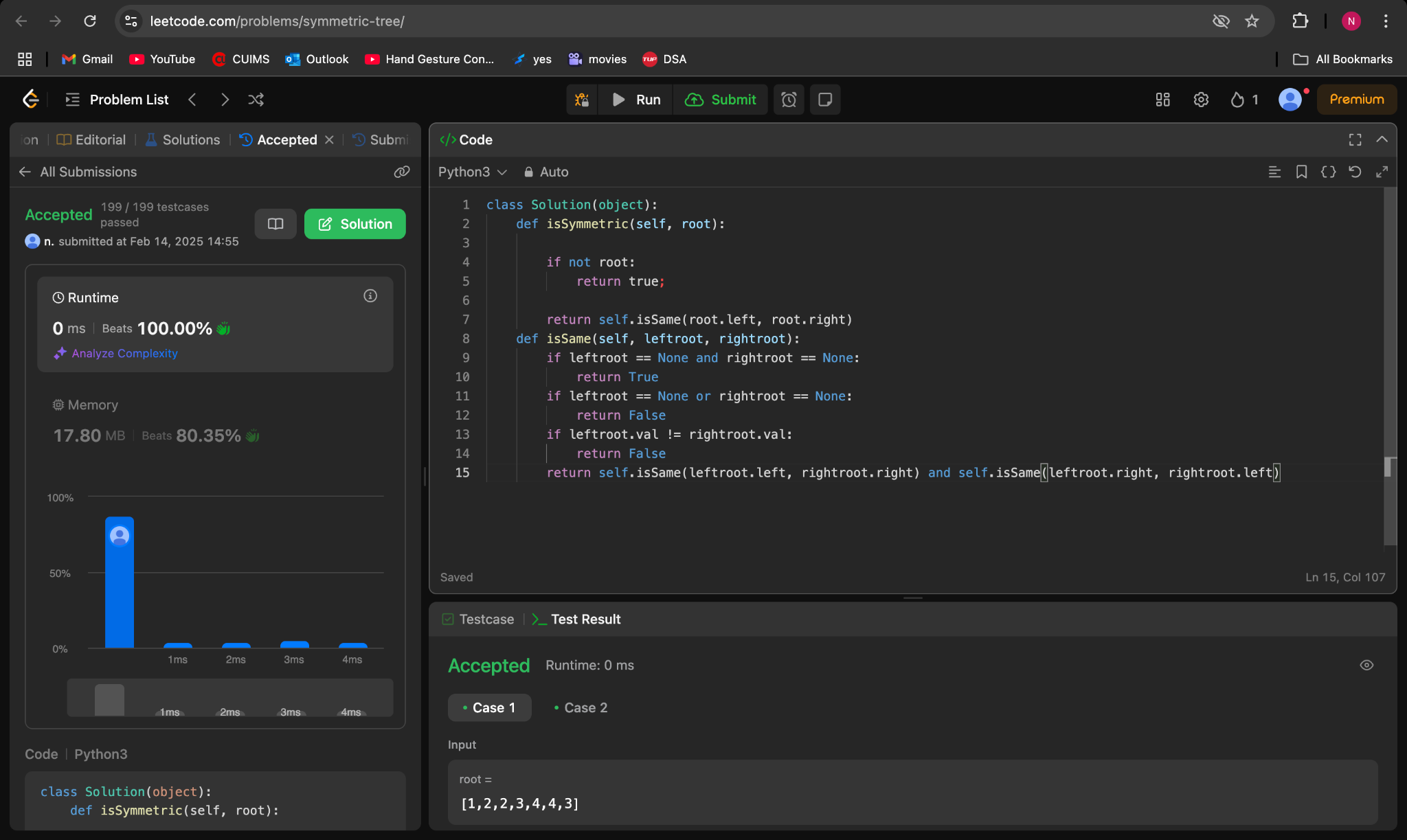
**if leftroot == None or rightroot == None:**

**return False**

**if leftroot.val != rightroot.val:**

**return False**

**return self.isSame(leftroot.left, rightroot.right) and self.isSame(leftroot.right, rightroot.left)**

****

**104.Maximum Depth of Binary Tree**

**class Solution:**

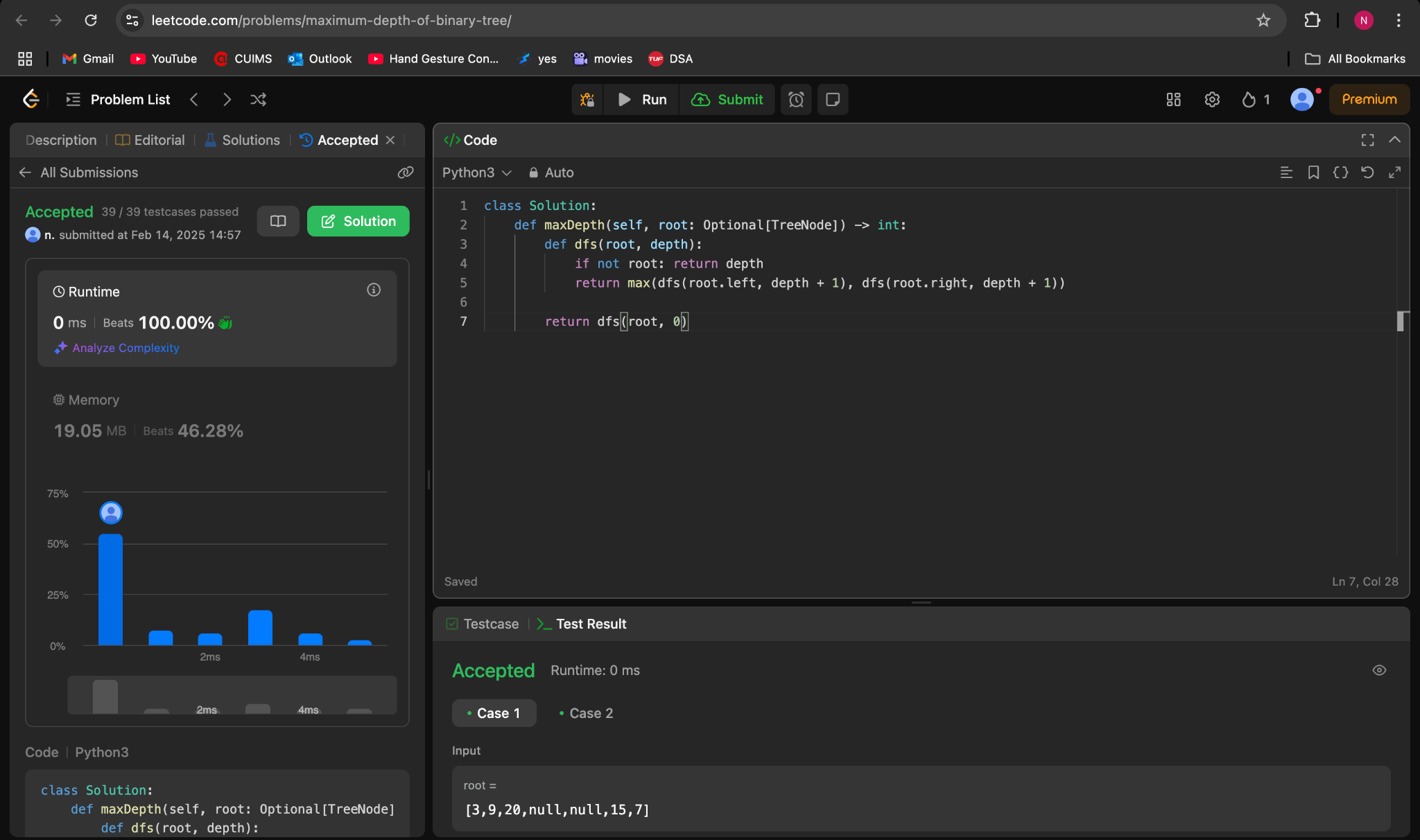
**def maxDepth(self, root: Optional[TreeNode]) -> int:**

**def dfs(root, depth):**

**if not root: return depth**

**return max(dfs(root.left, depth + 1), dfs(root.right, depth + 1))**

**return dfs(root, 0)**

****

**98.Validate Binary Search Tree**

**class Solution:**

**def isValidBST(self, root: Optional[TreeNode]) -> bool:**

**prev = float('-inf')**

**def inorder(node):**

**nonlocal prev**

**if not node:**

**return True**

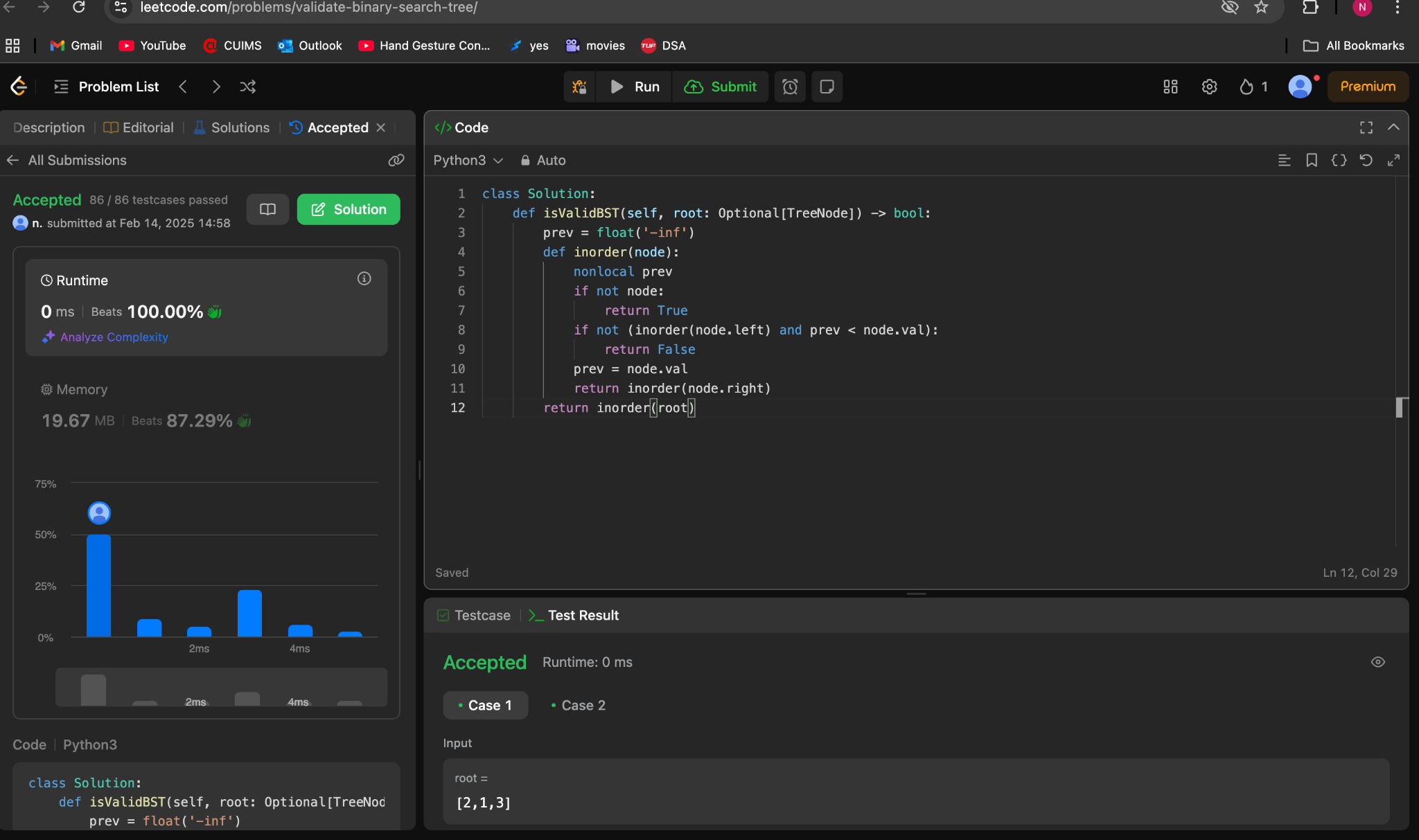
**if not (inorder(node.left) and prev < node.val):**

**return False**

**prev = node.val**

**return inorder(node.right)**

**return inorder(root)**

****

**230.Kth Smallest Element in a BST**

**class Solution:**

**def kthSmallest(self, root: TreeNode | None, k: int) -> int:**

**def countNodes(root: TreeNode | None) -> int:**

**if not root:**

**return 0**

**return 1 + countNodes(root.left) + countNodes(root.right)**

**leftCount = countNodes(root.left)**

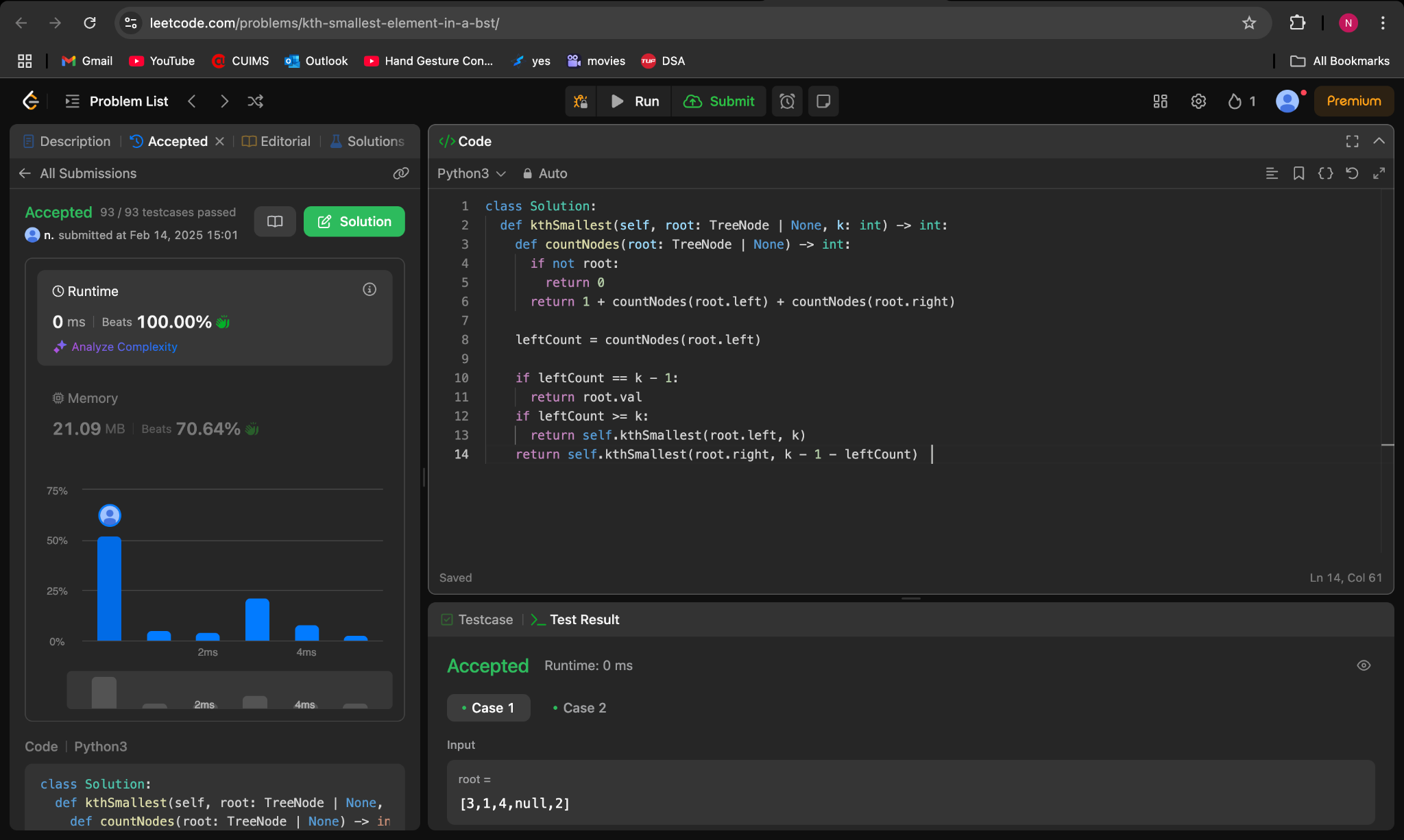
**if leftCount == k - 1:**

**return root.val**

**if leftCount >= k:**

**return self.kthSmallest(root.left, k)**

**return self.kthSmallest(root.right, k - 1 - leftCount)**

****

**102. Binary Tree Level Order Traversal**

**# Definition for a binary tree node.**

**# class TreeNode:**

**# def \_\_init\_\_(self, val=0, left=None, right=None):**

**# self.val = val**

**# self.left = left**

**# self.right = right**

**class Solution:**

**def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:**

**ans = []**

**if root is None:**

**return ans**

**q = deque([root])**

**while q:**

**t = []**

**for \_ in range(len(q)):**

**node = q.popleft()**

**t.append(node.val)**

**if node.left:**

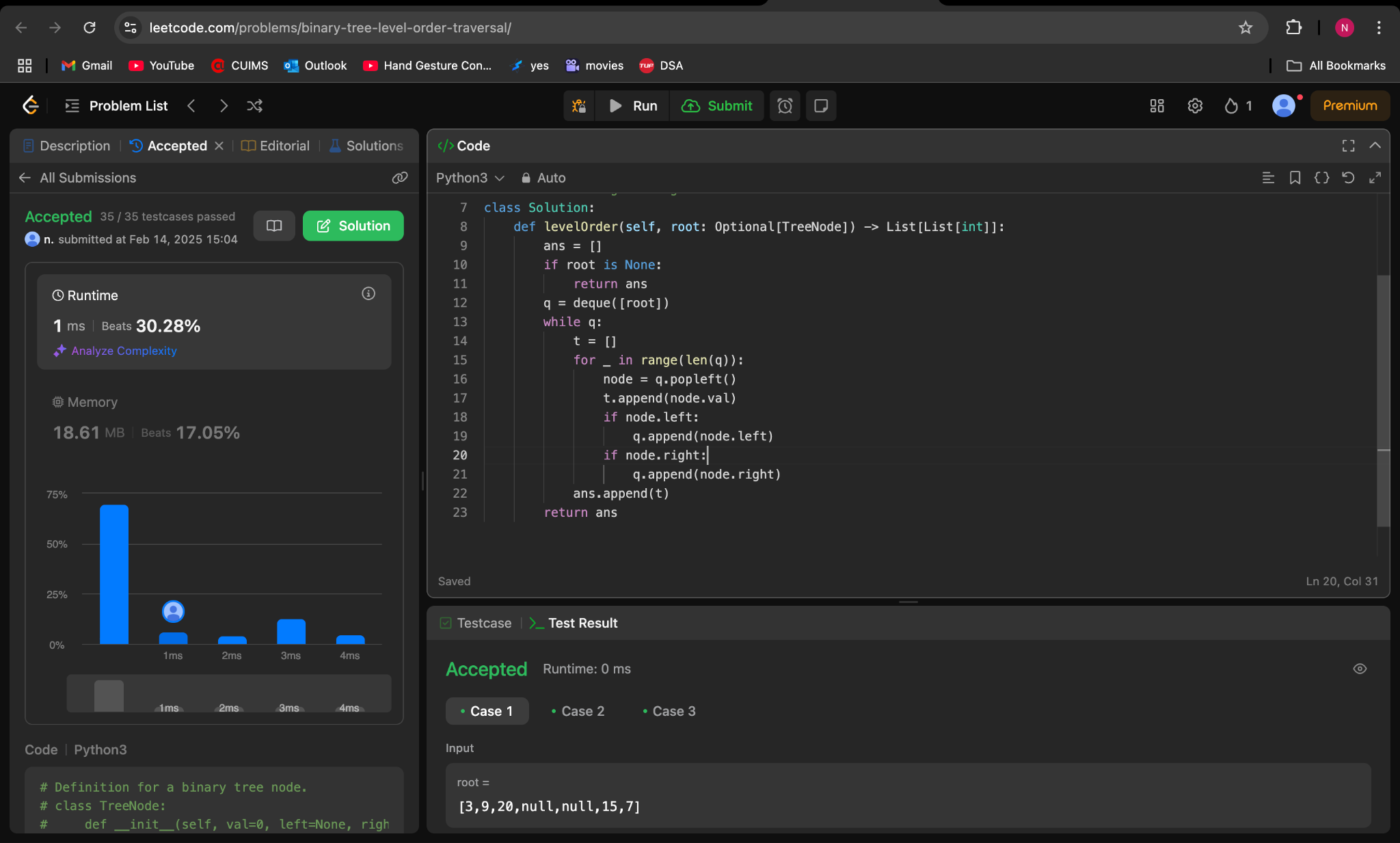
**q.append(node.left)**

**if node.right:**

**q.append(node.right)**

**ans.append(t)**

**return ans**

****

**107.Binary Tree Level Order Traversal II**

**from collections import deque**

**class TreeNode:**

**def \_\_init\_\_(self, val=0, left=None, right=None):**

**self.val = val**

**self.left = left**

**self.right = right**

**class Solution:**

**def levelOrderBottom(self, root: Optional[TreeNode]) -> List[List[int]]:**

**levels\_reversed = []**

**if not root:**

**return levels\_reversed**

**queue = deque([root])**

**while queue:**

**level\_values = []**

**for \_ in range(len(queue)):**

**current\_node = queue.popleft()**

**level\_values.append(current\_node.val)**

**if current\_node.left:**

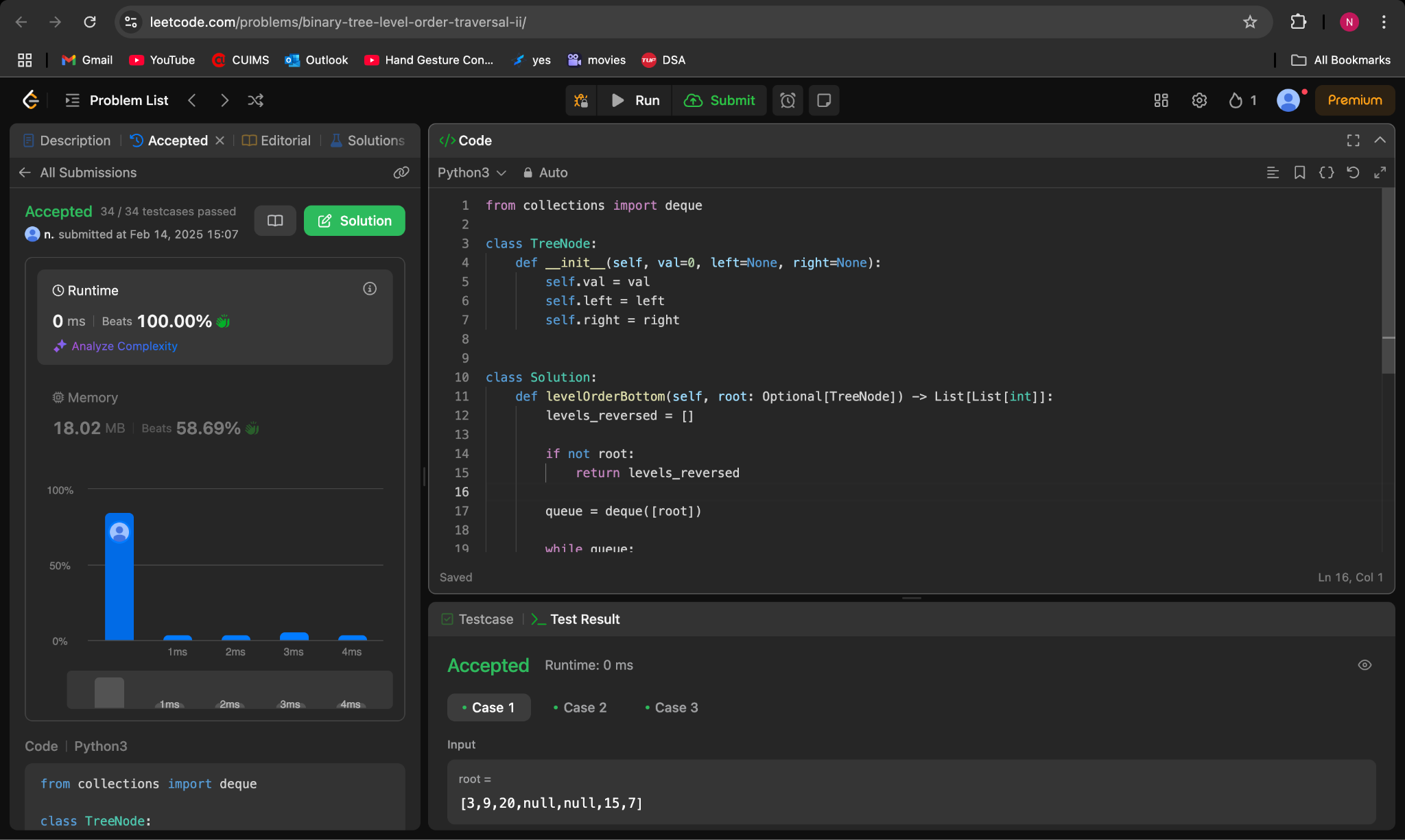
**queue.append(current\_node.left)**

**if current\_node.right:**

**queue.append(current\_node.right)**

**levels\_reversed.append(level\_values)**

**return levels\_reversed[::-1]**

****

**103.Binary Tree Zigzag Level Order Traversal**

**from collections import deque**

**class TreeNode:**

**def \_\_init\_\_(self, val=0, left=None, right=None):**

**self.val = val**

**self.left = left**

**self.right = right**

**class Solution:**

**def zigzagLevelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:**

**result = []**

**if root is None:**

**return result**

**queue = deque([root])**

**left\_to\_right = True**

**while queue:**

**level\_values = []**

**for \_ in range(len(queue)):**

**node = queue.popleft()**

**level\_values.append(node.val)**

**if node.left:**

**queue.append(node.left)**

**if node.right:**

**queue.append(node.right)**

**if left\_to\_right:**

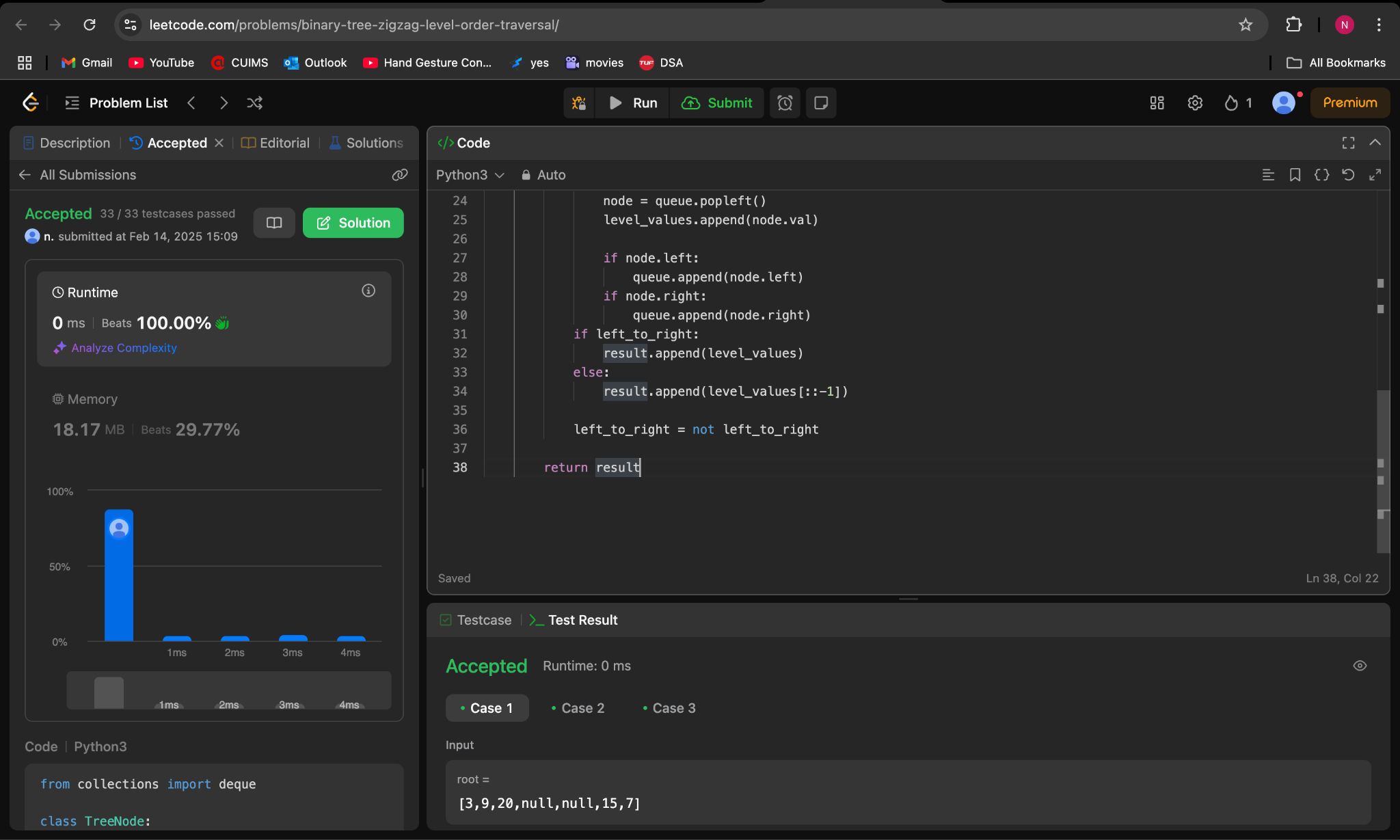
**result.append(level\_values)**

**else:**

**result.append(level\_values[::-1])**

**left\_to\_right = not left\_to\_right**

**return result**

****

**199.Binary Tree Right Side View**

**from collections import deque**

**class TreeNode:**

**def \_\_init\_\_(self, val=0, left=None, right=None):**

**self.val = val**

**self.left = left**

**self.right = right**

**class Solution:**

**def rightSideView(self, root: Optional[TreeNode]) -> List[int]:**

**right\_side\_view = []**

**if root is None:**

**return right\_side\_view**

**queue = deque([root])**

**while queue:**

**right\_side\_view.append(queue[-1].val)**

**for \_ in range(len(queue)):**

**current\_node = queue.popleft()**

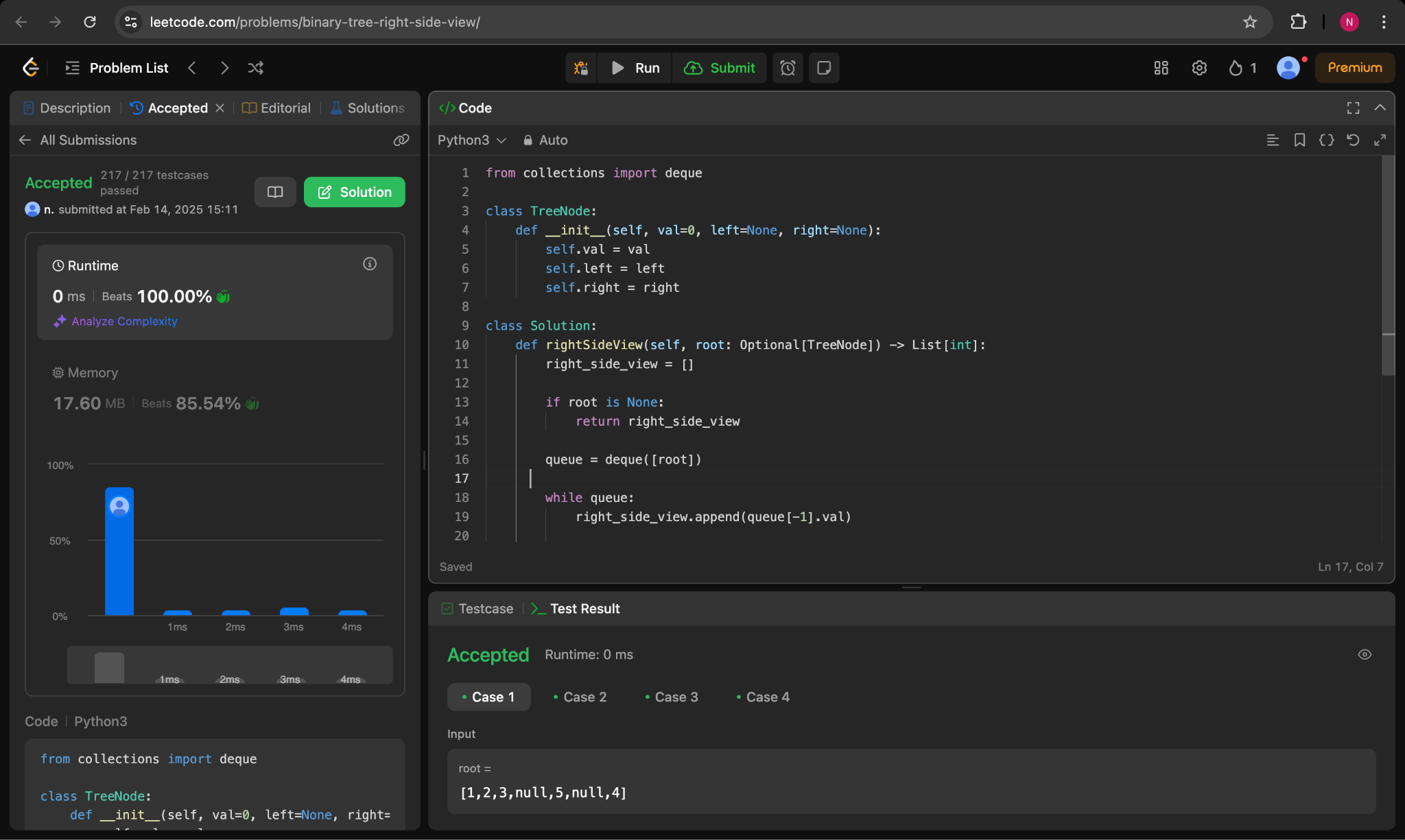
**if current\_node.left:**

**queue.append(current\_node.left)**

**if current\_node.right:**

**queue.append(current\_node.right)**

**return right\_side\_view**

****

**106.Construct Binary Tree from Inorder and Postorder Traversal**

**class Solution:**

**def buildTree(**

**self,**

**inorder: list[int],**

**postorder: list[int],**

**) -> TreeNode | None:**

**inToIndex = {num: i for i, num in enumerate(inorder)}**

**def build(**

**inStart: int,**

**inEnd: int,**

**postStart: int,**

**postEnd: int,**

**) -> TreeNode | None:**

**if inStart > inEnd:**

**return None**

**rootVal = postorder[postEnd]**

**rootInIndex = inToIndex[rootVal]**

**leftSize = rootInIndex - inStart**

**root = TreeNode(rootVal)**

**root.left = build(inStart, rootInIndex - 1, postStart,**

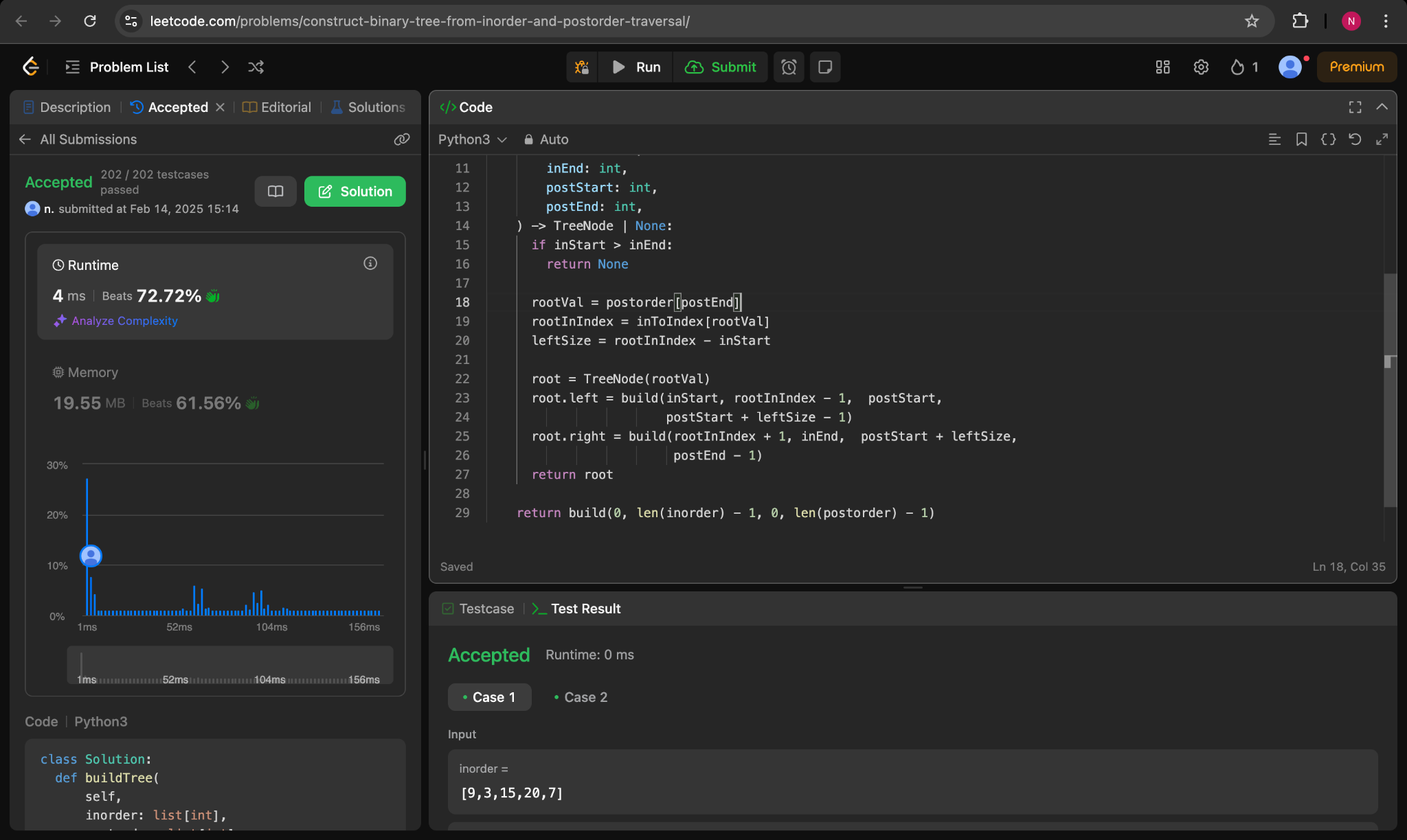
**postStart + leftSize - 1)**

**root.right = build(rootInIndex + 1, inEnd, postStart + leftSize,**

**postEnd - 1)**

**return root**

**return build(0, len(inorder) - 1, 0, len(postorder) - 1)**

****

**513.Find Bottom Left Tree Value**

**from collections import deque**

**class TreeNode:**

**def \_\_init\_\_(self, val=0, left=None, right=None):**

**self.val = val**

**self.left = left**

**self.right = right**

**class Solution:**

**def findBottomLeftValue(self, root: TreeNode) -> int:**

**node\_queue = deque([root])**

**bottom\_left\_value = 0**

**while node\_queue:**

**bottom\_left\_value = node\_queue[0].val**

**for \_ in range(len(node\_queue)):**

**node = node\_queue.popleft()**

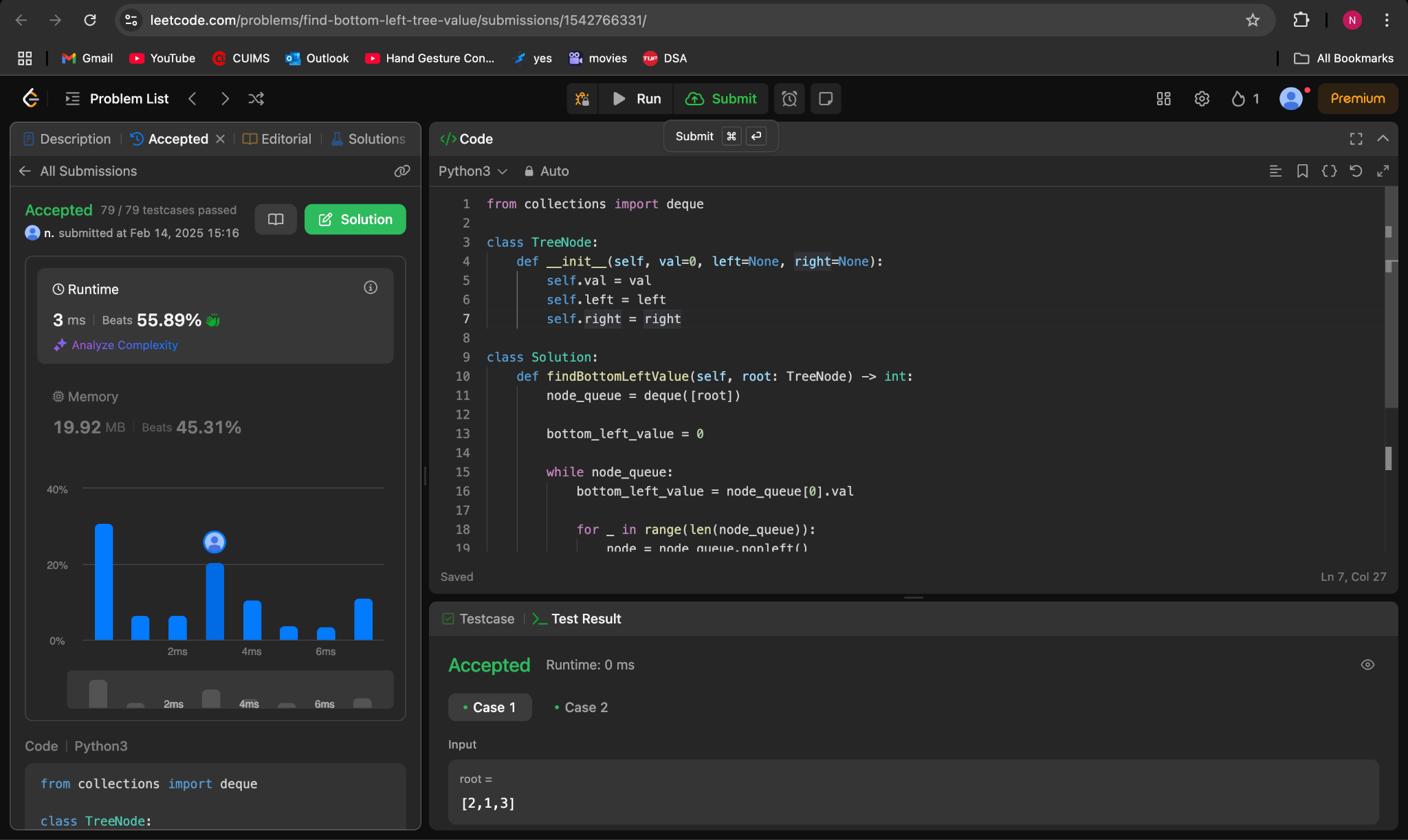
**if node.left:**

**node\_queue.append(node.left)**

**if node.right:**

**node\_queue.append(node.right)**

**return bottom\_left\_value**

****

**124. Binary Tree Maximum Path Sum**

**class Solution:**

**def maxPathSum(self, root: TreeNode | None) -> int:**

**ans = -math.inf**

**def maxPathSumDownFrom(root: TreeNode | None) -> int:**

**nonlocal ans**

**if not root:**

**return 0**

**l = max(0, maxPathSumDownFrom(root.left))**

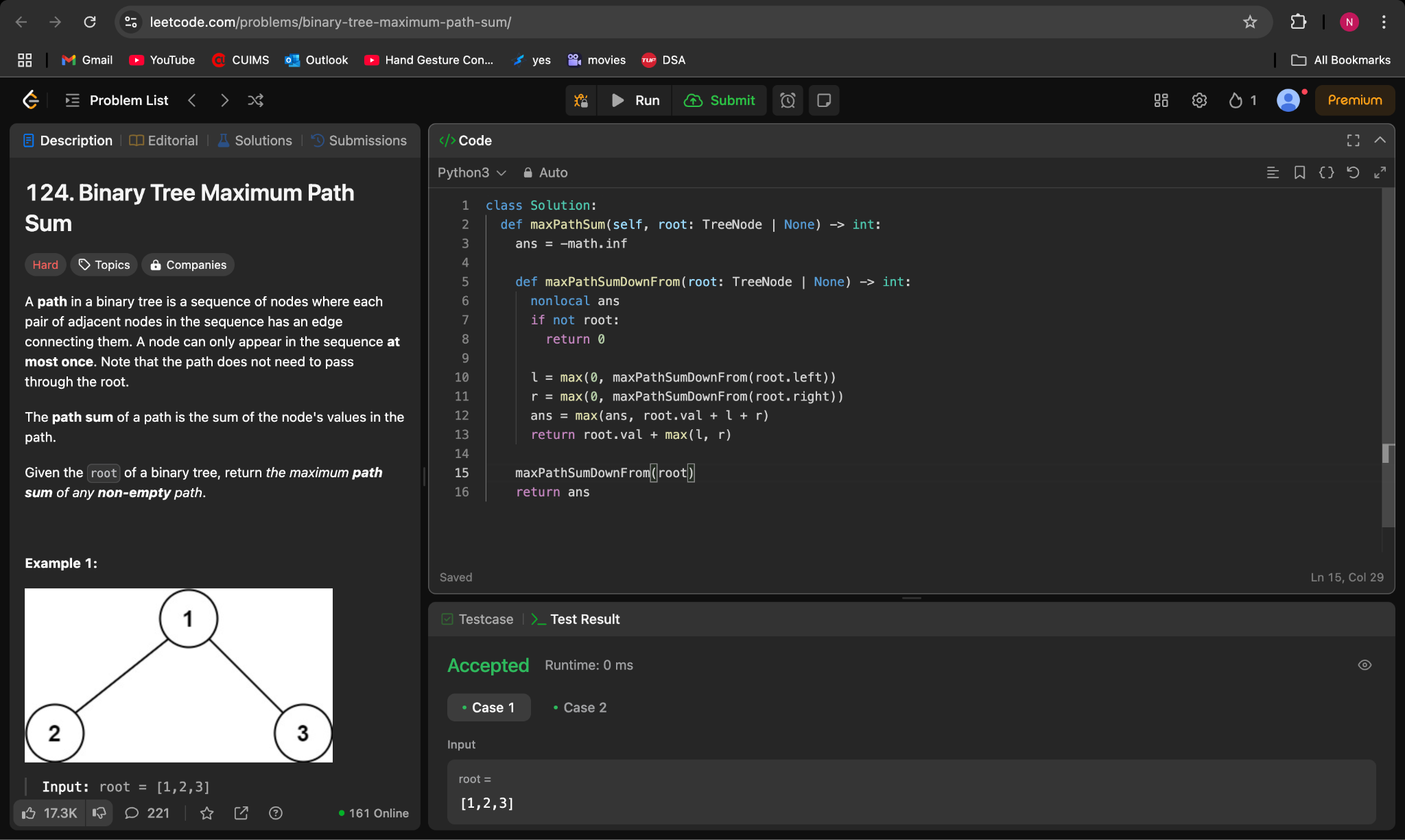
**r = max(0, maxPathSumDownFrom(root.right))**

**ans = max(ans, root.val + l + r)**

**return root.val + max(l, r)**

**maxPathSumDownFrom(root)**

**return ans**

****

**987.Vertical Order Traversal of a Binary Tree**

**class Solution:**

**def verticalTraversal(self, root: TreeNode | None) -> list[list[int]]:**

**ans = []**

**xToNodes = collections.defaultdict(list)**

**def dfs(node: TreeNode | None, x: int, y: int) -> None:**

**if not node:**

**return**

**xToNodes[x].append((-y, node.val))**

**dfs(node.left, x - 1, y - 1)**

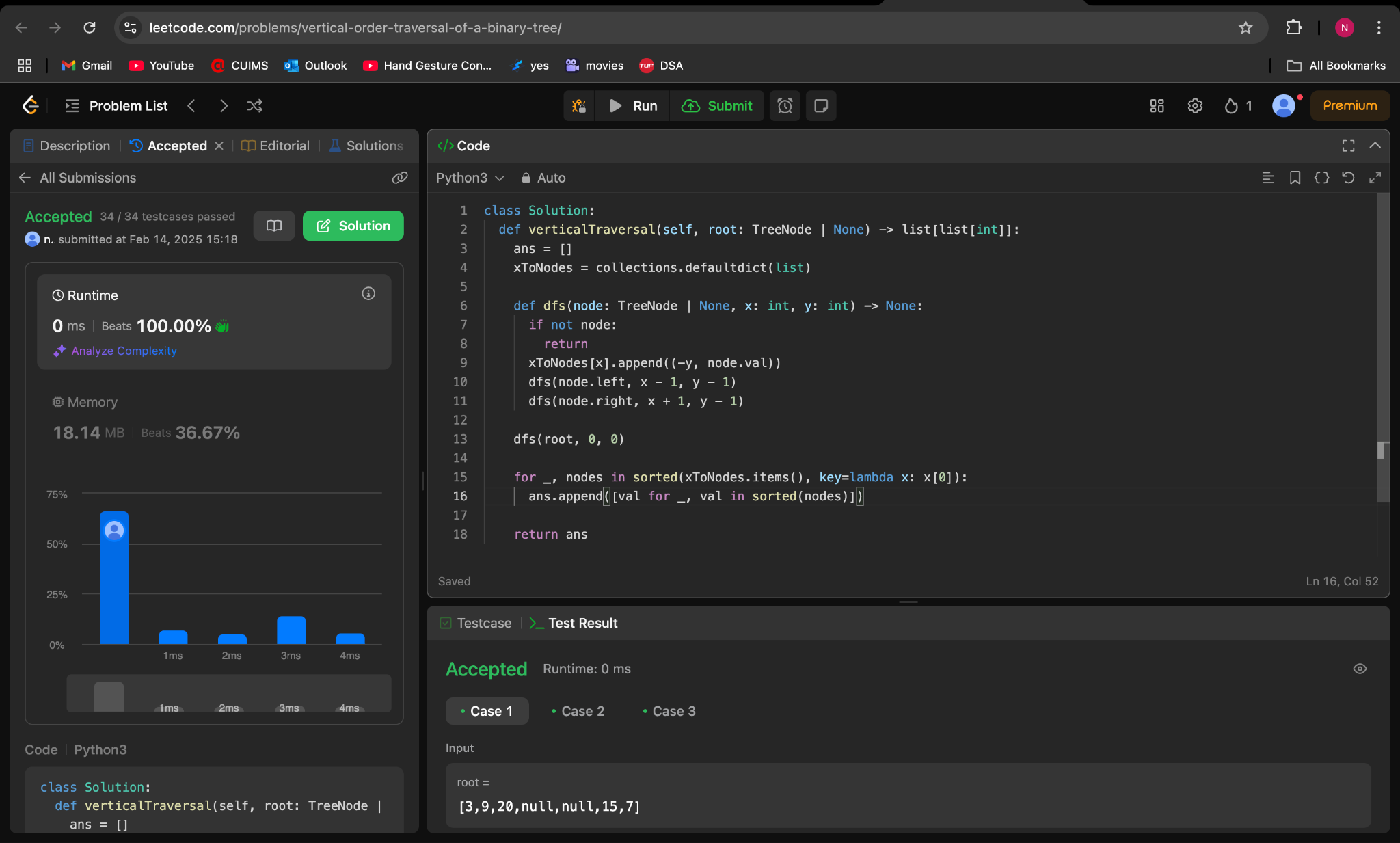
**dfs(node.right, x + 1, y - 1)**

**dfs(root, 0, 0)**

**for \_, nodes in sorted(xToNodes.items(), key=lambda x: x[0]):**

**ans.append([val for \_, val in sorted(nodes)])**

**return ans**

****