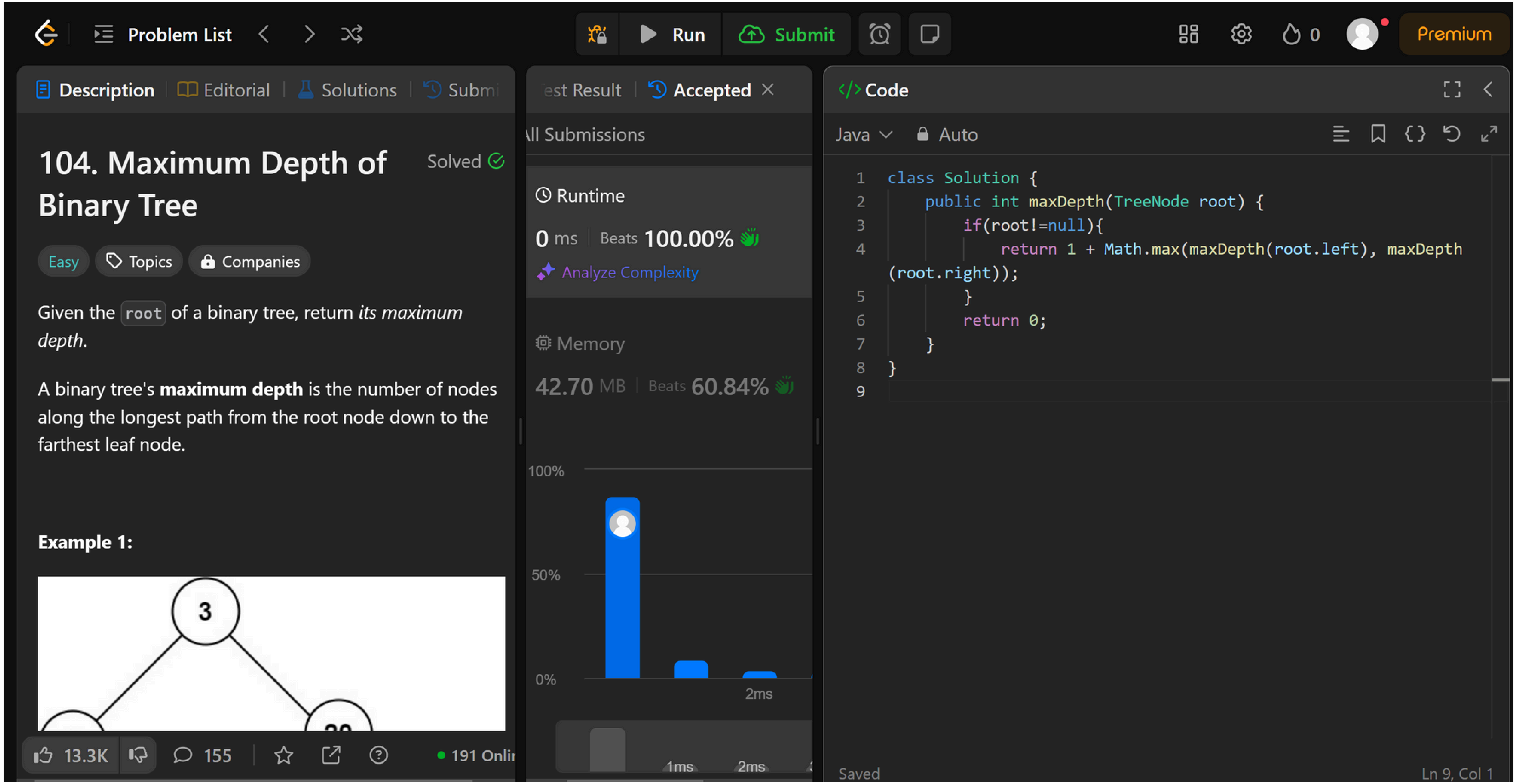
DSA Day - 14

PRESENTATION

Leetcode Problems With Solutions

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:

ll Submissions

Runtime

0 ms | Beats 100.00%

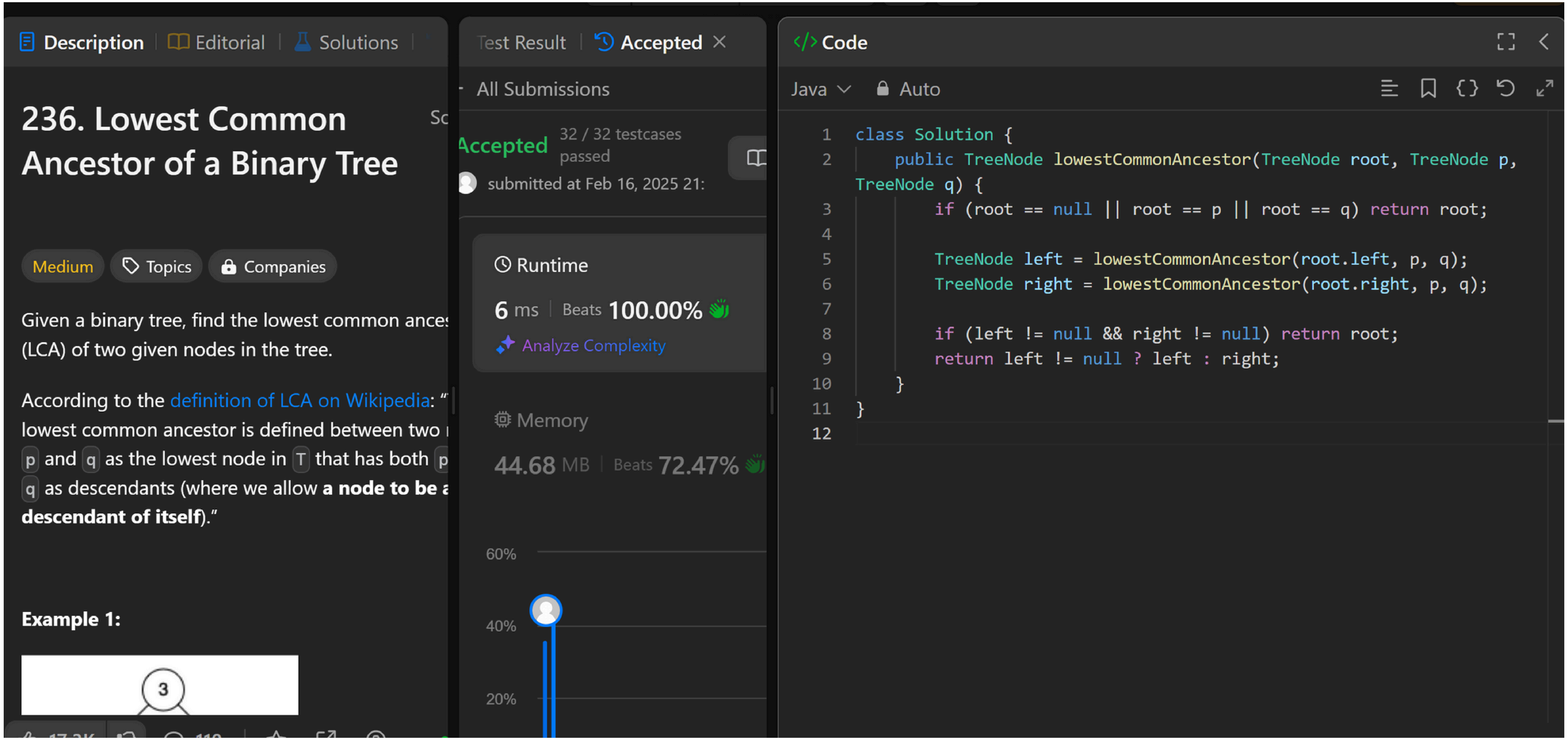
Analyze Complexity

Memory

42.70 MB | Beats 60.84%

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

236. Lowest Common Ancestor of a Binary Tree



236. Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ance (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: " lowest common ancestor is defined between two p and q as the lowest node in T that has both p q as descendants (where we allow **a node to be a descendant of itself**)."

Example 1:

Accepted 32 / 32 testcases passed

submitted at Feb 16, 2025 21:

Runtime

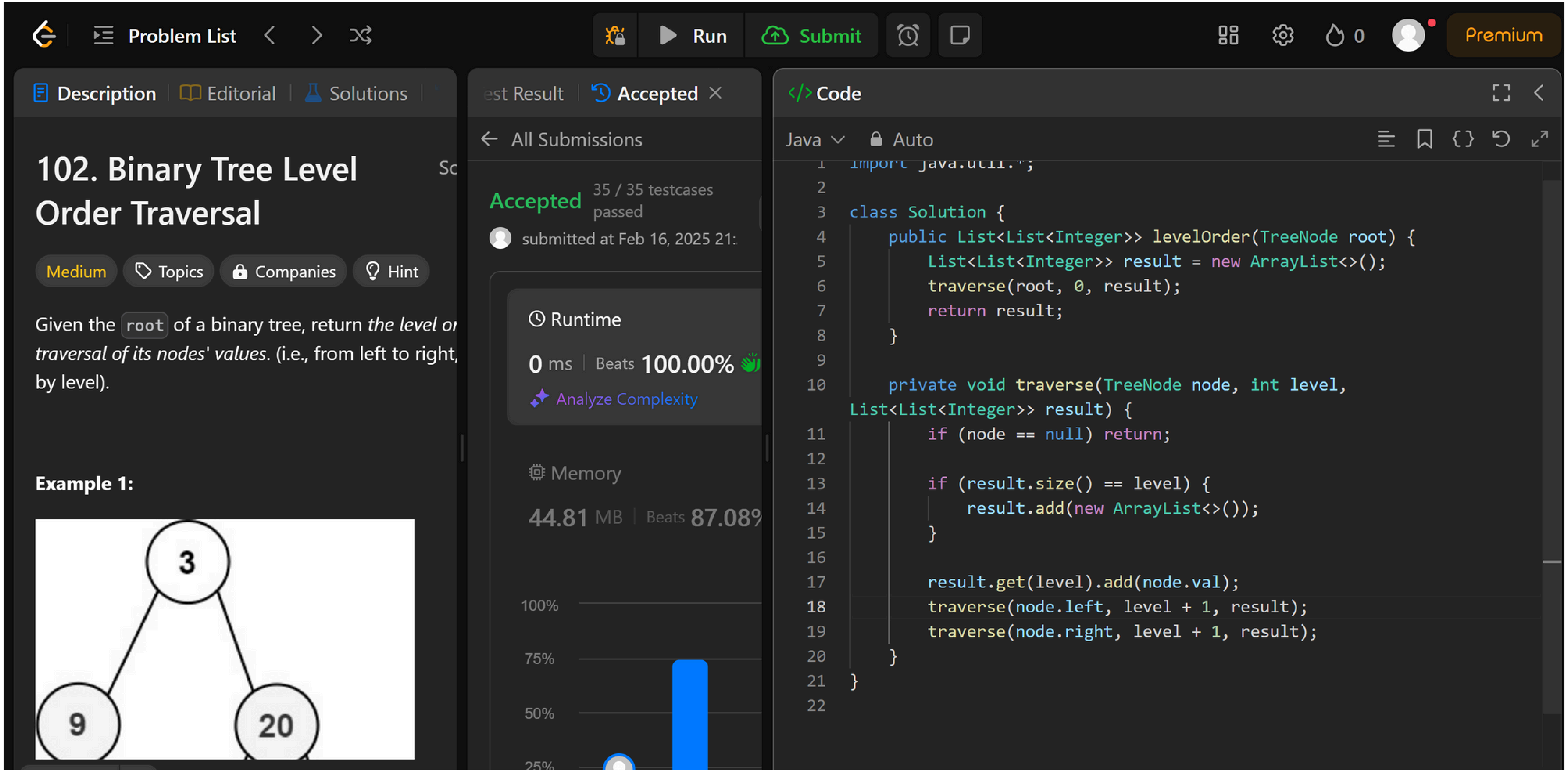
6 ms | Beats 100.00%

Memory

44.68 MB | Beats 72.47%

```java
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || root == p || root == q) return root;

        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);

        if (left != null && right != null) return root;
        return left != null ? left : right;
    }
}
```

102. Binary Tree Level Order Traversal



102. Binary Tree Level Order Traversal

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

Example 1:

Accepted 35 / 35 testcases passed

submitted at Feb 16, 2025 21:

Runtime

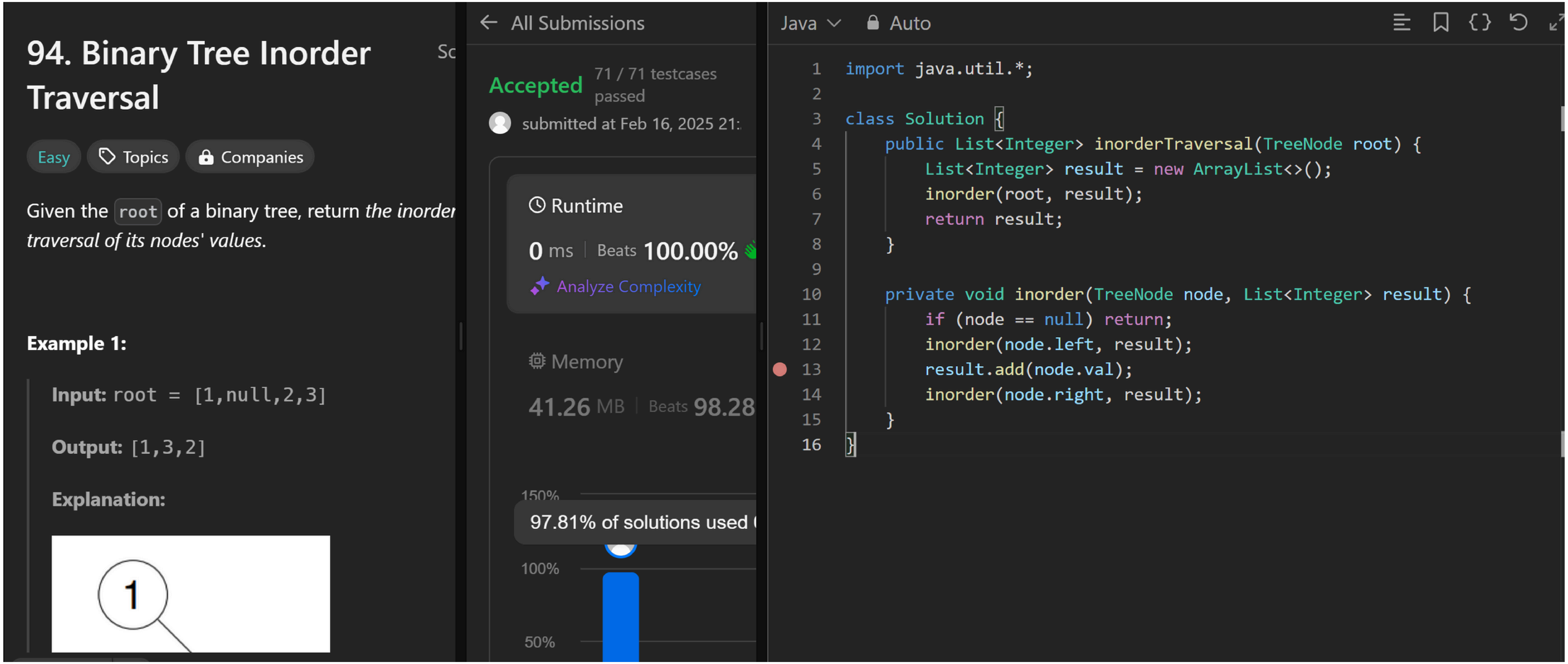
0 ms | Beats 100.00%

Memory

44.81 MB | Beats 87.08%

```java
import java.util.*;

class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        traverse(root, 0, result);
        return result;
    }

    private void traverse(TreeNode node, int level, List<List<Integer>> result) {
        if (node == null) return;

        if (result.size() == level) {
            result.add(new ArrayList<>());
        }

        result.get(level).add(node.val);
        traverse(node.left, level + 1, result);
        traverse(node.right, level + 1, result);
    }
}
```

94. Binary Tree Inorder Traversal



94. 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]`

Explanation:

Accepted 71 / 71 testcases passed

submitted at Feb 16, 2025 21:

Runtime

0 ms | Beats 100.00%

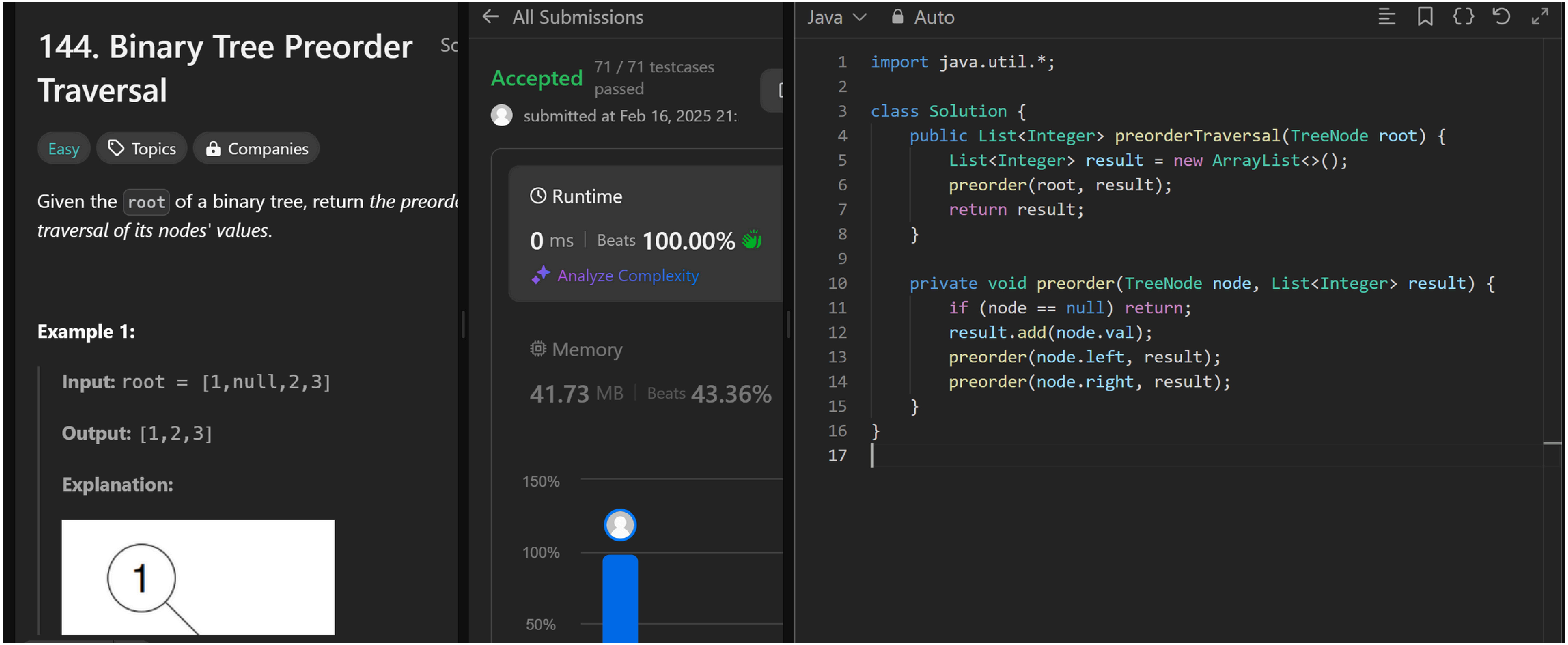
Memory

41.26 MB | Beats 98.28

97.81% of solutions used (

```java
import java.util.*;

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

    private void inorder(TreeNode node, List<Integer> result) {
        if (node == null) return;
        inorder(node.left, result);
        result.add(node.val);
        inorder(node.right, result);
    }
}
```

144. Binary Tree Preorder Traversal



144. Binary Tree Preorder Traversal

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

Example 1:

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

Output: `[1,2,3]`

Explanation:

Accepted 71 / 71 testcases passed

submitted at Feb 16, 2025 21:

Runtime

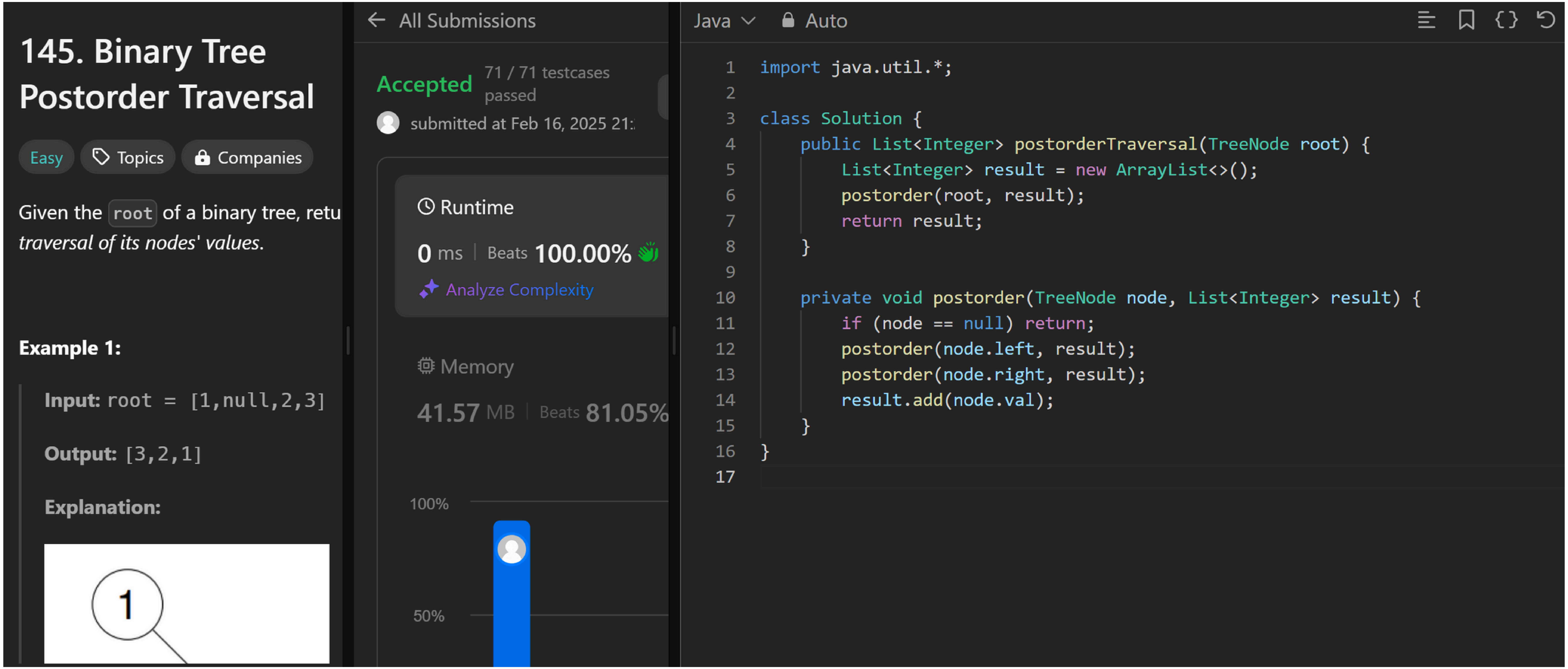
0 ms | Beats 100.00%

Memory

41.73 MB | Beats 43.36%

```java
import java.util.*;

class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        preorder(root, result);
        return result;
    }

    private void preorder(TreeNode node, List<Integer> result) {
        if (node == null) return;
        result.add(node.val);
        preorder(node.left, result);
        preorder(node.right, result);
    }
}
```

145. Binary Tree Postorder Traversal



145. Binary Tree Postorder Traversal

Easy Topics Companies

Given the `root` of a binary tree, retu
traversal of its nodes' values.

Example 1:

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

Output: `[3,2,1]`

Explanation:

All Submissions

Accepted 71 / 71 testcases passed

submitted at Feb 16, 2025 21:

Runtime

0 ms | Beats 100.00%

Analyze Complexity

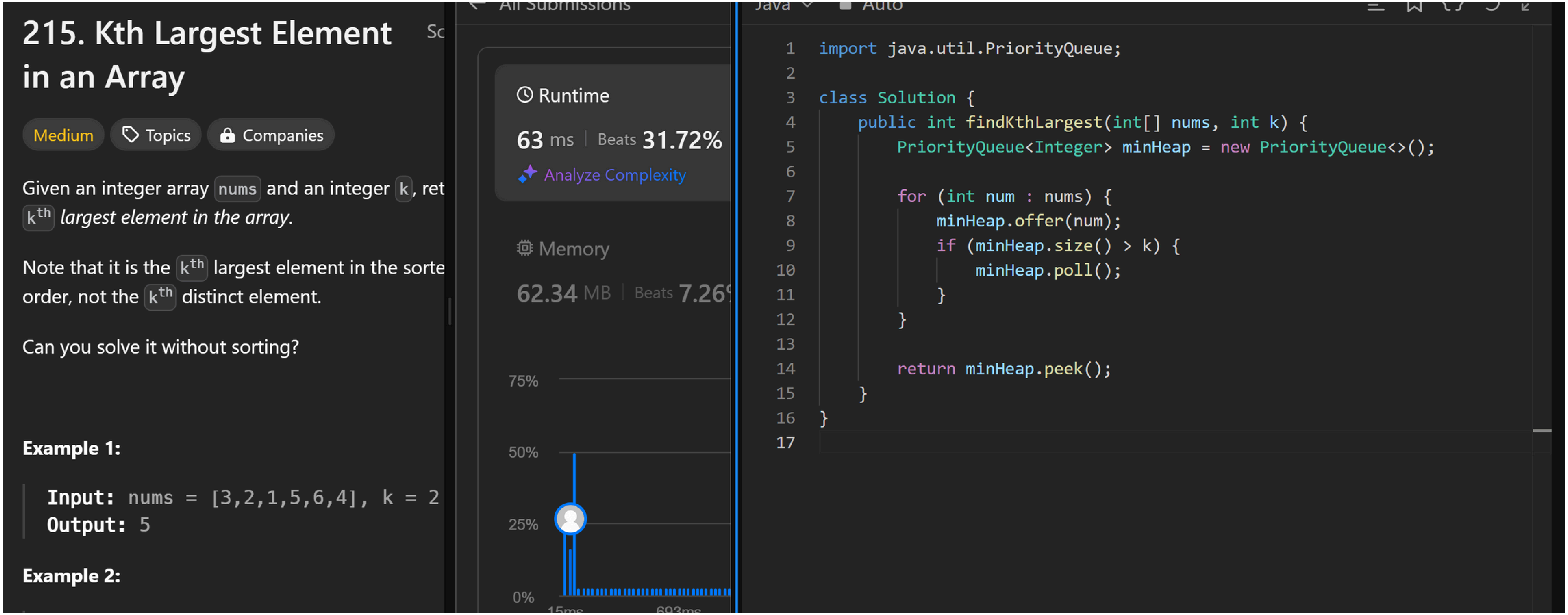
Memory

41.57 MB | Beats 81.05%

Java Auto

```java
import java.util.*;

class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        postorder(root, result);
        return result;
    }

    private void postorder(TreeNode node, List<Integer> result) {
        if (node == null) return;
        postorder(node.left, result);
        postorder(node.right, result);
        result.add(node.val);
    }
}
```

215. Kth Largest Element in an Array



215. Kth Largest Element in an Array

Given an integer array `nums` and an integer `k`, ret
`k`th *largest element in the array*.

Note that it is the `k`th largest element in the sorte
order, not the `k`th distinct element.

Can you solve it without sorting?

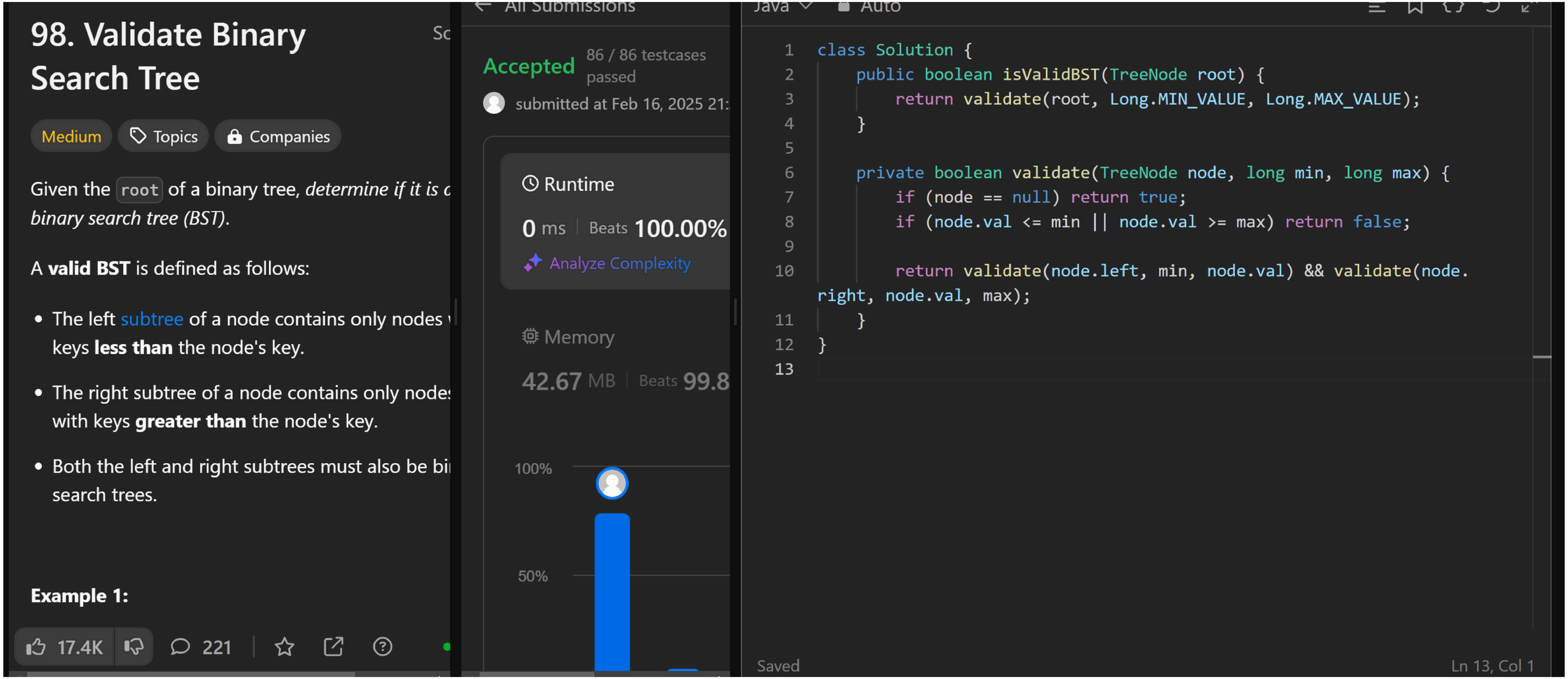
Example 1:

```
Input: nums = [3,2,1,5,6,4], k = 2
Output: 5
```

Example 2:

```java
import java.util.PriorityQueue;

class Solution {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int num : nums) {
            minHeap.offer(num);
            if (minHeap.size() > k) {
                minHeap.poll();
            }
        }

        return minHeap.peek();
    }
}
```

98. Validate Binary Search Tree



98. Validate Binary Search Tree

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

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

- The left subtree of a node contains only nodes w
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 bi
search trees.

Example 1:

Accepted 86 / 86 testcases passed

submitted at Feb 16, 2025 21:

Runtime

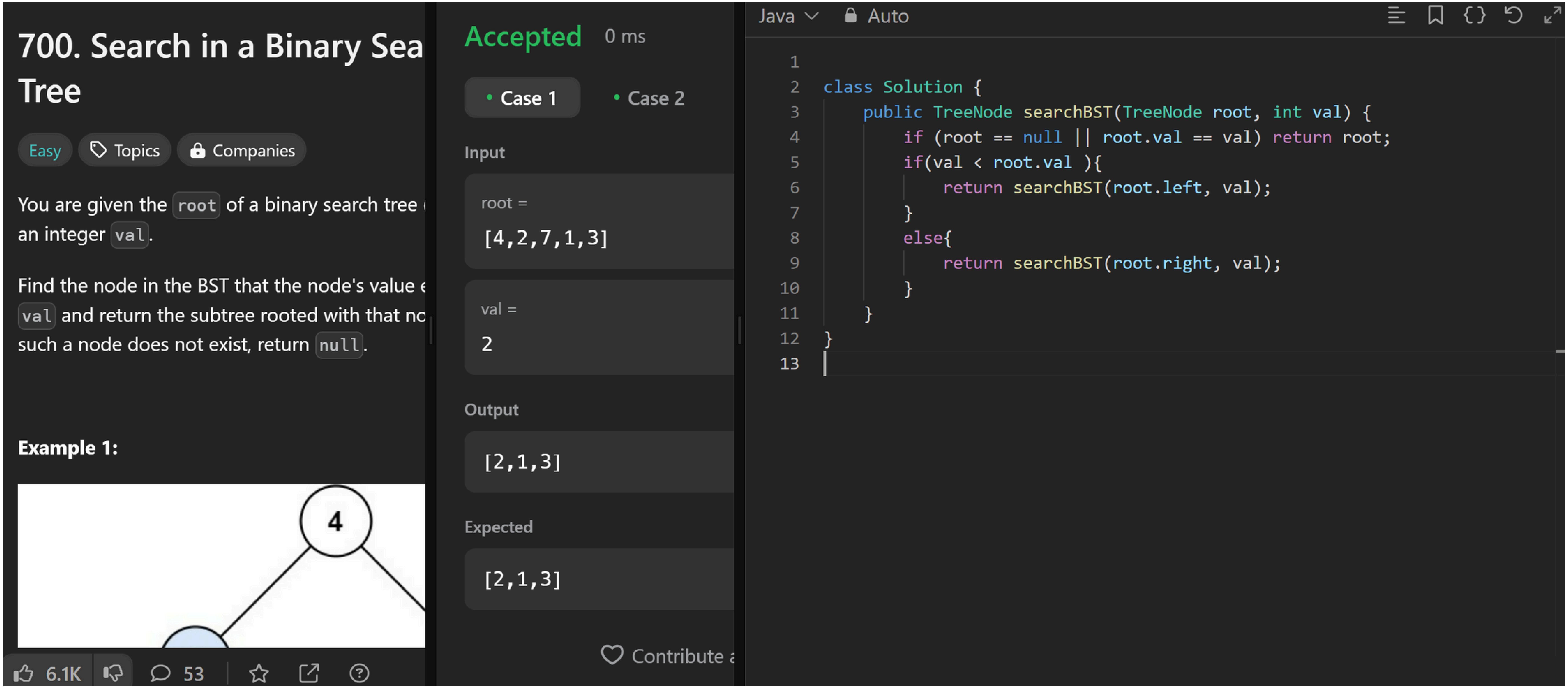
0 ms | Beats 100.00%

Memory

42.67 MB | Beats 99.8

```java
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);
    }
}
```

700. Search in a Binary Search Tree



700. Search in a Binary Sea Tree

You are given the `root` of a binary search tree (an integer `val`.

Find the node in the BST that the node's value e `val` and return the subtree rooted with that no such a node does not exist, return `null`.

Example 1:

Accepted 0 ms

Input

root =

```
[4,2,7,1,3]
```

val =

```
2
```

Output

```
[2,1,3]
```

Expected

```
[2,1,3]
```

```java

class Solution {
    public TreeNode searchBST(TreeNode root, int val) {
        if (root == null || root.val == val) return root;
        if(val < root.val ){
            return searchBST(root.left, val);
        }
        else{
            return searchBST(root.right, val);
        }
    }
}

```