

Experiment 2

Student Name: Garv Gupta

Branch: BE-CSE

Semester: 6th

Subject Name: AP Lab-II

UID: 22BCS11750

Section/Group: 22BCS_IOT-638/B

Date of Performance: 21/01/2025

Subject Code: 22CSP-351

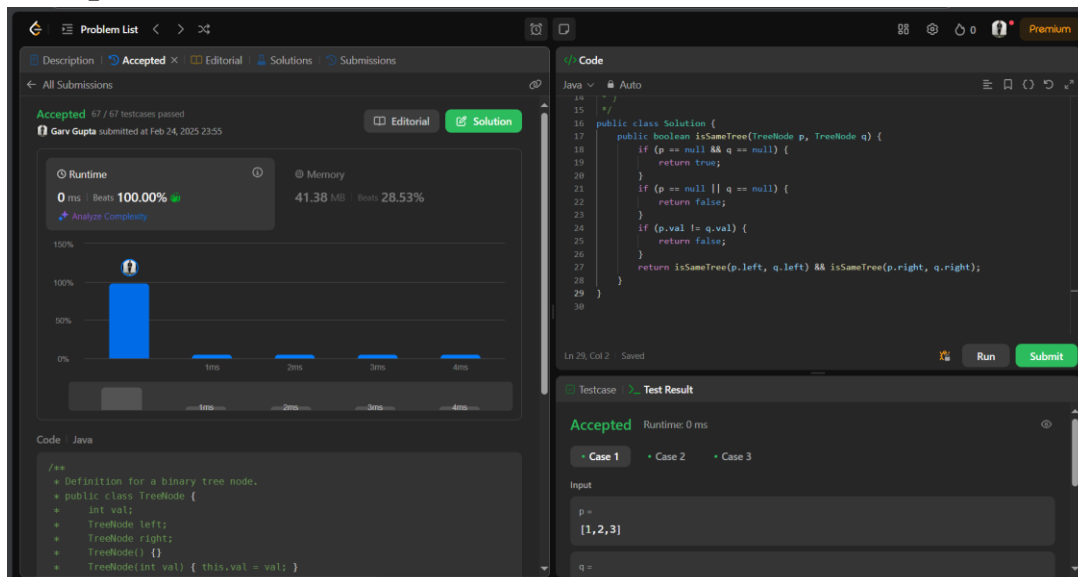
A. Same Tree

- 1. Aim:** Given the roots of two binary trees p and q, write a function to check if they are the same or not. Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

2. Code:

```
public class Solution {  
    public boolean isSameTree(TreeNode p, TreeNode q) {  
        if (p == null && q == null) {  
            return true;  
        }  
        if (p == null || q == null) {  
            return false;  
        }  
        if (p.val != q.val) {  
            return false;  
        }  
        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);  
    }  
}
```

3. Output:



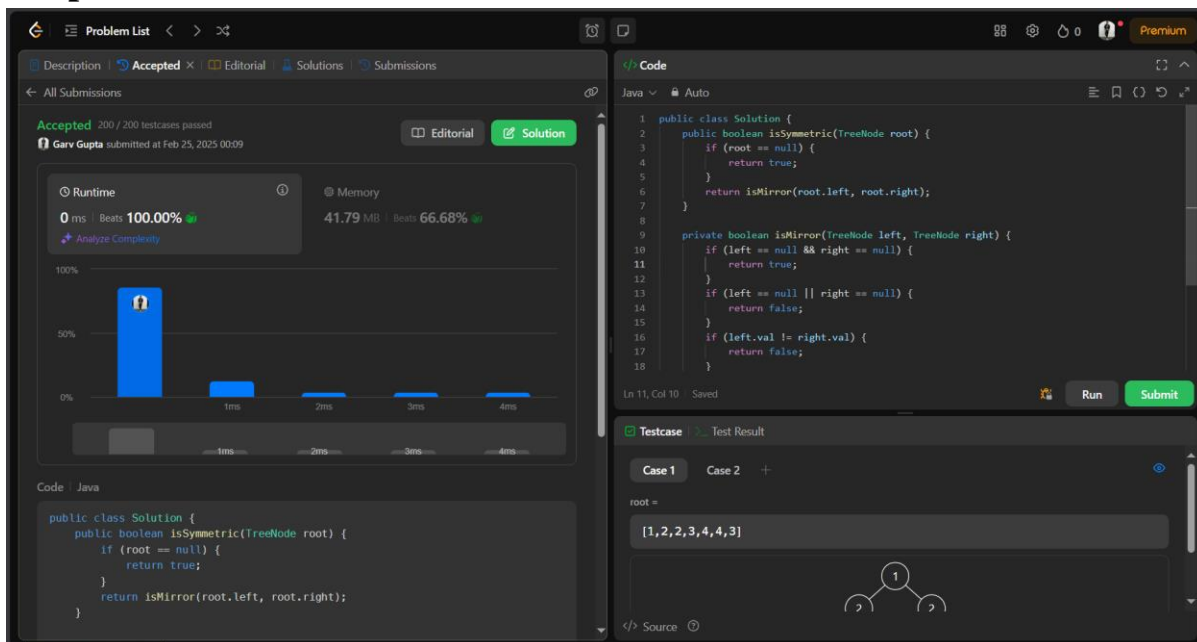
B. Symmetric Tree

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

2. Code:

```
public class Solution {
    public boolean isSymmetric(TreeNode root) {
        if (root == null) {
            return true;
        }
        return isMirror(root.left, root.right);
    }
    private boolean isMirror(TreeNode left, TreeNode right) {
        if (left == null && right == null) {
            return true;
        }
        if (left == null || right == null) {
            return false;
        }
        if (left.val != right.val) {
            return false;
        }
        return isMirror(left.left, right.right) && isMirror(left.right, right.left);
    }
}
```

3. Output:



C. Balanced Binary Tree

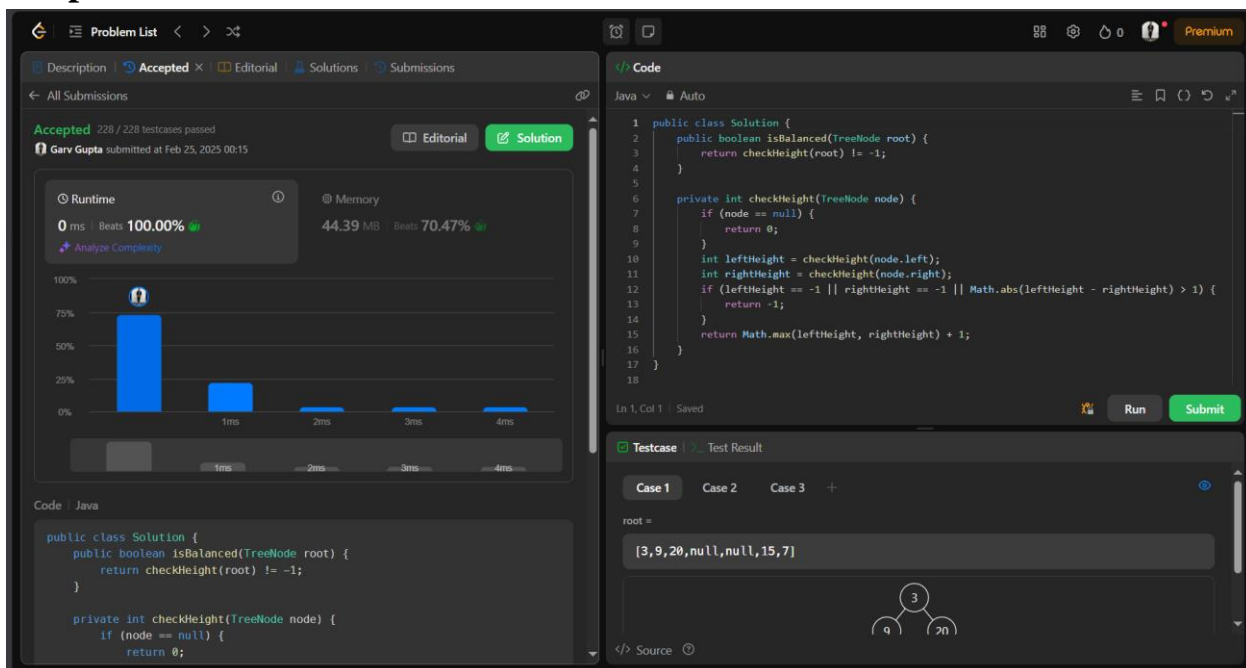
1. Aim: Given a binary tree, determine if it is height-balanced.

2. Code:

```
public class Solution {
    public boolean isBalanced(TreeNode root) {
        return checkHeight(root) != -1;
    }

    private int checkHeight(TreeNode node) {
        if (node == null) {
            return 0;
        }
        int leftHeight = checkHeight(node.left);
        int rightHeight = checkHeight(node.right);
        if (leftHeight == -1 || rightHeight == -1 || Math.abs(leftHeight -
rightHeight) > 1) {
            return -1;
        }
        return Math.max(leftHeight, rightHeight) + 1;
    }
}
```

3. Output:



The screenshot displays a coding platform interface with the following components:

- Problem List:** Shows the problem is "Accepted" with 228/228 testcases passed. The user "Garv Gupta" submitted it on Feb 25, 2025, at 00:15.
- Runtime and Memory:**
 - Runtime: 0 ms, Beats 100.00%
 - Memory: 44.39 MB, Beats 70.47%
- Bar Chart:** A bar chart showing the distribution of runtime performance across different percentiles.
- Code Editor:** Contains the Java code for the solution, including the `isBalanced` and `checkHeight` methods.
- Testcase:** Shows "Case 1" with the input `[3,9,20,null,null,15,7]`. Below the input, a binary tree diagram is shown with root 3, left child 9, right child 20, and 20's left child 15.
- Buttons:** "Run" and "Submit" buttons are visible at the bottom right of the code editor.

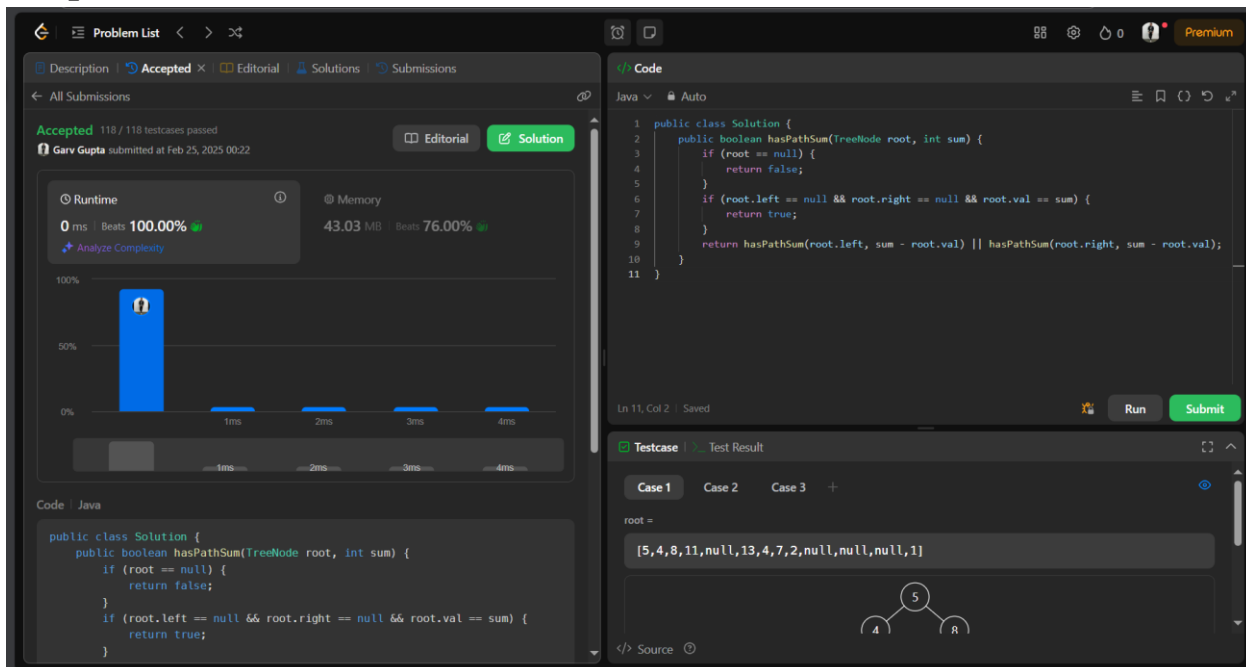
D. Path Sum

1. Aim: Given the root of a binary tree and an integer target Sum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals target Sum. A leaf is a node with no children.

2. Code:

```
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if (root == null) {
            return false;
        }
        if (root.left == null && root.right == null && root.val == sum) {
            return true;
        }
        return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum -
root.val);
    }
}
```

3. Output:



The screenshot shows a code editor interface for a problem titled "Path Sum". The left sidebar displays the problem description, which is marked as "Accepted" with 118/118 testcases passed. The user "Garv Gupta" submitted the solution on Feb 25, 2025. Performance metrics show a runtime of 0 ms (Beats 100.00%) and memory usage of 43.03 MB (Beats 76.00%).

The main editor area shows the Java code for the solution:

```
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if (root == null) {
            return false;
        }
        if (root.left == null && root.right == null && root.val == sum) {
            return true;
        }
        return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);
    }
}
```

The bottom right panel shows a test case with the input array: [5, 4, 8, 11, null, 13, 4, 7, 2, null, null, null, 1]. Below the array is a binary tree diagram with root 5, left child 4, and right child 8. Node 4 has a right child 11. Node 8 has a left child 13. Node 13 has a left child 4. Node 4 has a right child 7. Node 7 has a left child 2. Node 2 has a right child 1.

E. Count Complete Tree

1. **Aim:** Given the root of a complete binary tree, return the number of the nodes in the tree. According to Wikipedia, every level, except possibly the last, is completely filled in a complete binary tree, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes inclusive at the last level h . Design an algorithm that runs in less than $O(n)$ time complexity.

2. Code:

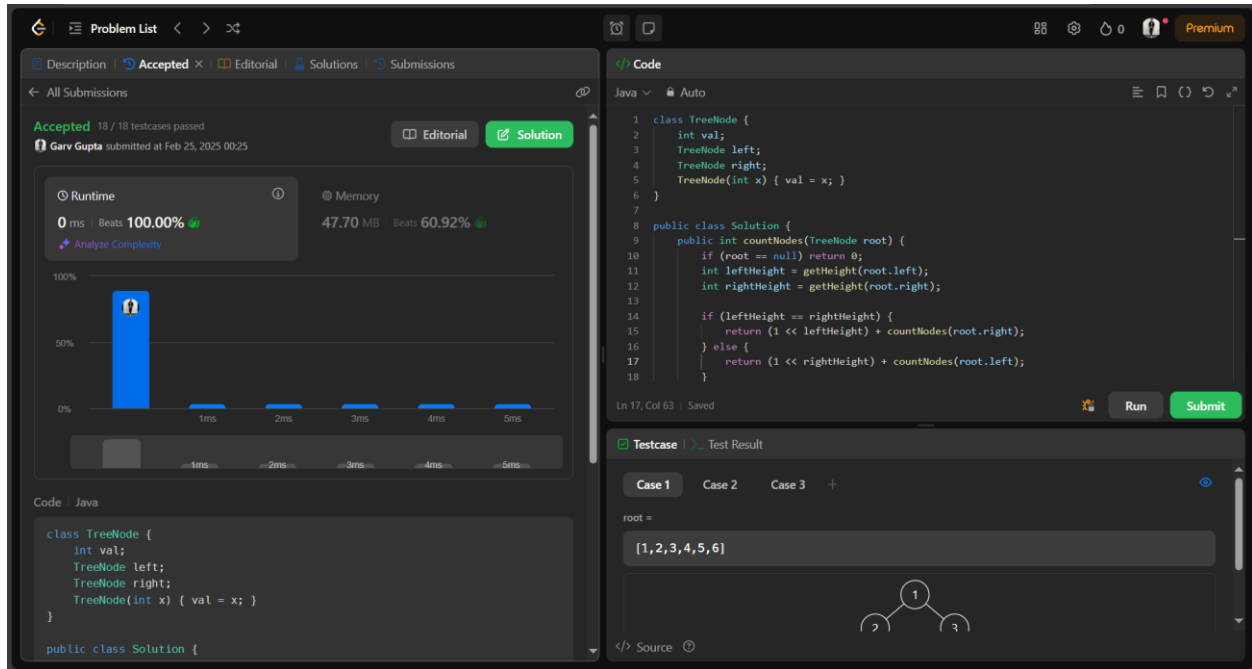
```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class Solution {
    public int countNodes(TreeNode root) {
        if (root == null) return 0;
        int leftHeight = getHeight(root.left);
        int rightHeight = getHeight(root.right);

        if (leftHeight == rightHeight) {
            return (1 << leftHeight) + countNodes(root.right);
        } else {
            return (1 << rightHeight) + countNodes(root.left);
        }
    }

    private int getHeight(TreeNode node) {
        int height = 0;
        while (node != null) {
            height++;
            node = node.left;
        }
        return height;
    }
}
```

3. Output:



F. Delete Node in a BST

- Aim:** Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST. Basically, the deletion can be divided into two stages: Search for a node to remove. If the node is found, delete the node.

2. Code:

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

public class Solution {
    public TreeNode deleteNode(TreeNode root, int key) {
        if (root == null) return null;

        if (key < root.val) {
            root.left = deleteNode(root.left, key);
        } else if (key > root.val) {
            root.right = deleteNode(root.right, key);
        } else {
            // Node to be deleted found
            // Logic to delete the node
        }
    }
}
```

```

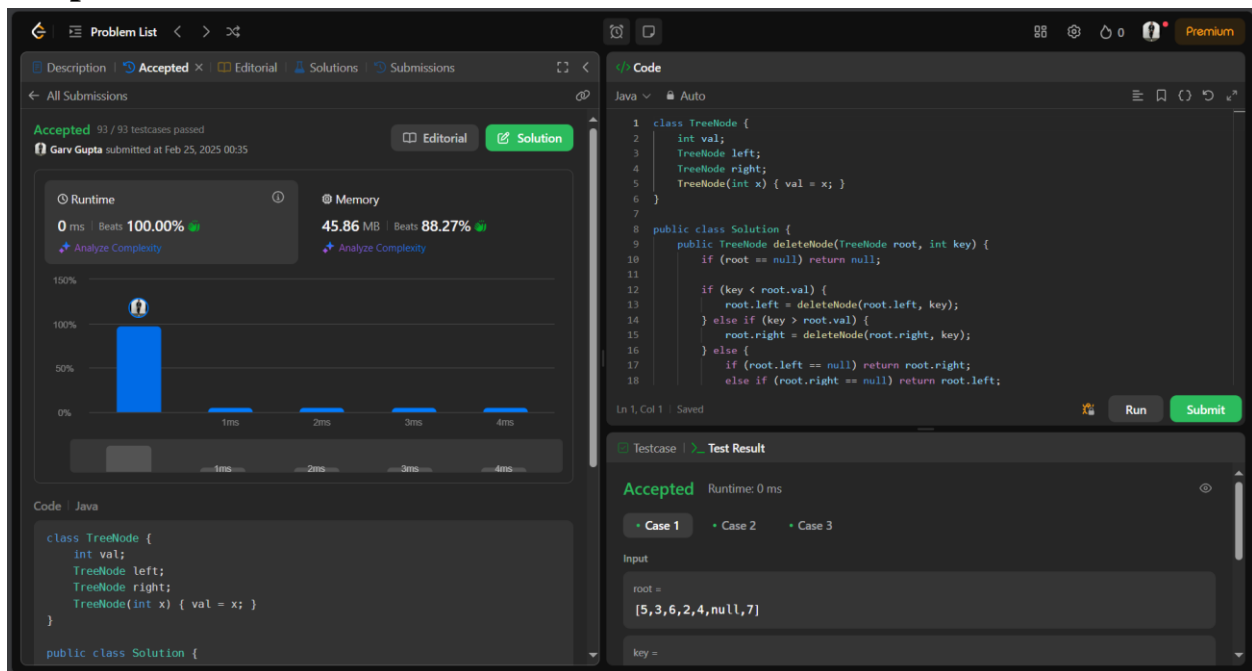
    } else {
        if (root.left == null) return root.right;
        else if (root.right == null) return root.left;

        TreeNode minNode = findMin(root.right);
        root.val = minNode.val;
        root.right = deleteNode(root.right, root.val);
    }
    return root;
}

private TreeNode findMin(TreeNode node) {
    while (node.left != null) {
        node = node.left;
    }
    return node;
}
}

```

3. Output:



The screenshot displays a coding platform interface with the following components:

- Problem List:** Shows the current problem and navigation options.
- Submission Status:** Indicates "Accepted" with 93/93 testcases passed. The user "Garv Gupta" submitted on Feb 25, 2025, at 00:35.
- Performance Metrics:**
 - Runtime:** 0 ms, Beats 100.00%.
 - Memory:** 45.86 MB, Beats 88.27%.
- Graph:** A bar chart showing the user's performance relative to other submissions.
- Code Editor:** Contains the Java code for the solution.


```

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

public class Solution {
    // ... (code continues)
}

```
- Test Result:** Shows the test case results.
 - Accepted:** Runtime: 0 ms.
 - Testcase 1:** Case 1, Case 2, Case 3.
 - Input:** root = [5,3,6,2,4,null,7], key = 3.

G. Diameter of Binary Tree

- 1. Aim:** Given the root of a binary tree, return the length of the diameter of the tree. The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root. The length of a path between two nodes is represented by the number of edges between them.

2. Code:

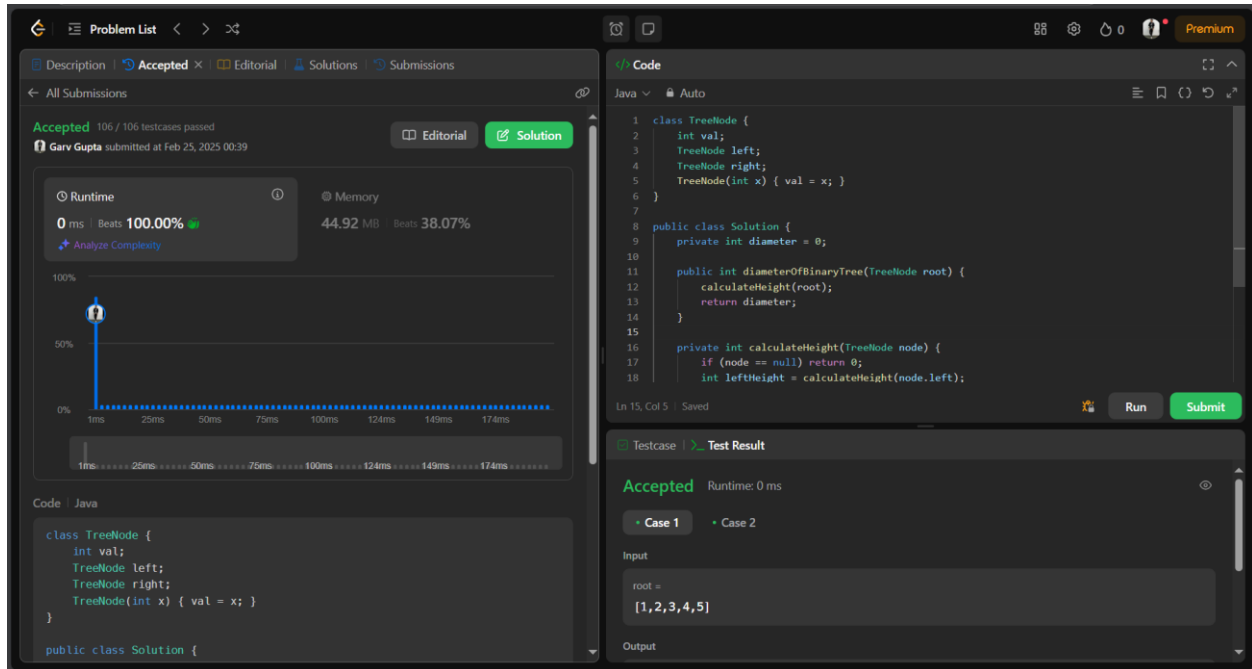
```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class Solution {
    private int diameter = 0;

    public int diameterOfBinaryTree(TreeNode root) {
        calculateHeight(root);
        return diameter;
    }

    private int calculateHeight(TreeNode node) {
        if (node == null) return 0;
        int leftHeight = calculateHeight(node.left);
        int rightHeight = calculateHeight(node.right);
        diameter = Math.max(diameter, leftHeight + rightHeight);
        return Math.max(leftHeight, rightHeight) + 1;
    }
}
```


3. Output:



The screenshot displays a coding platform interface with the following components:

- Problem List:** Shows the problem status as "Accepted" with 106 / 106 testcases passed. The user "Garv Gupta" submitted the solution on Feb 25, 2025 at 00:39.
- Runtime and Memory:** The solution achieved a runtime of 0 ms (Beats 100.00%) and used 44.92 MB of memory (Beats 38.07%).
- Code Editor:** The code is written in Java. It defines a `TreeNode` class with attributes `val`, `left`, and `right`, and a constructor `TreeNode(int x) { val = x; }`. The `Solution` class contains a method `diameterOfBinaryTree(TreeNode root)` that calculates the diameter of the tree, and a private helper method `calculateHeight(TreeNode node)` to find the height of the tree.
- Testcase:** The input for the first test case is `root = [1,2,3,4,5]`. The output is `Accepted` with a runtime of 0 ms.