ASSIGNMENT - 5

Student Name: Sameer **UID:** 22BCS15631

Branch: Computer Science & Engineering **Section/Group:** IOT-614/B

Semester: 6th **Date of Performance:** 10/03/2025

Subject Name: Advanced Programming Lab-2 **Subject Code:** 22CSP-351

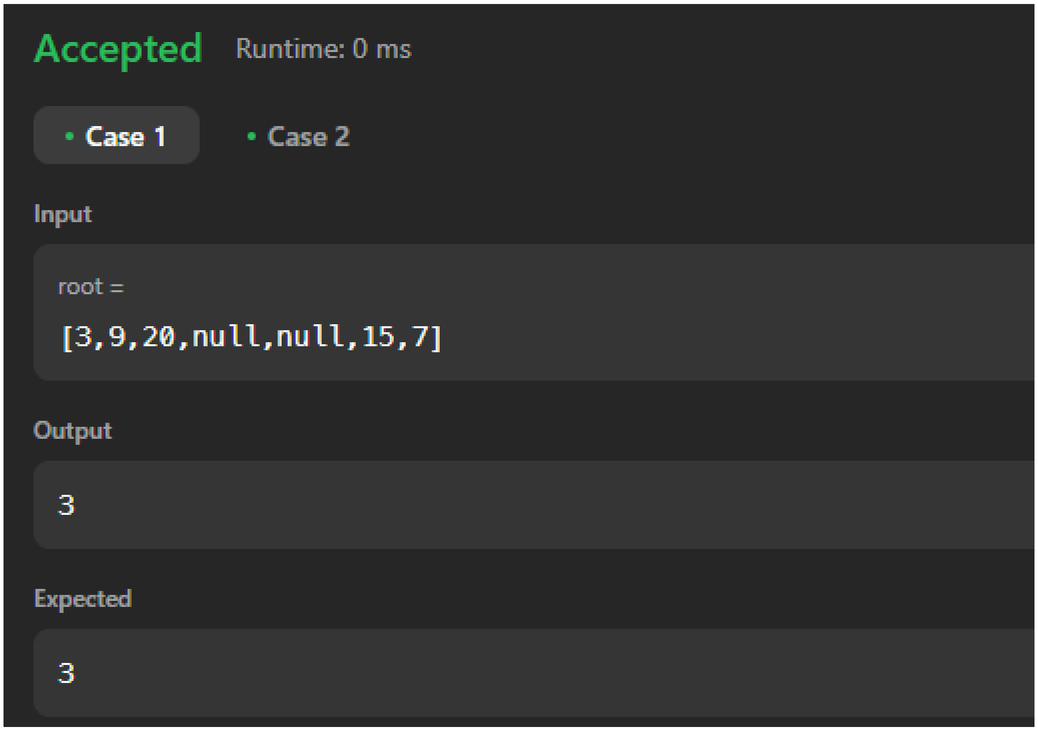
Q.1. Maximum Depth of Binary Tree

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.

Code:

```
class Solution {
public:
   int maxDepth(TreeNode* root)
   {
      if (root == NULL)
      {
         return 0 ;
      }

      int left = maxDepth(root->left) ;
      int right = maxDepth(root->right) ;
      return max(left, right) + 1 ;
   }
};
```

Output:



Accepted Runtime: 0 ms

Input

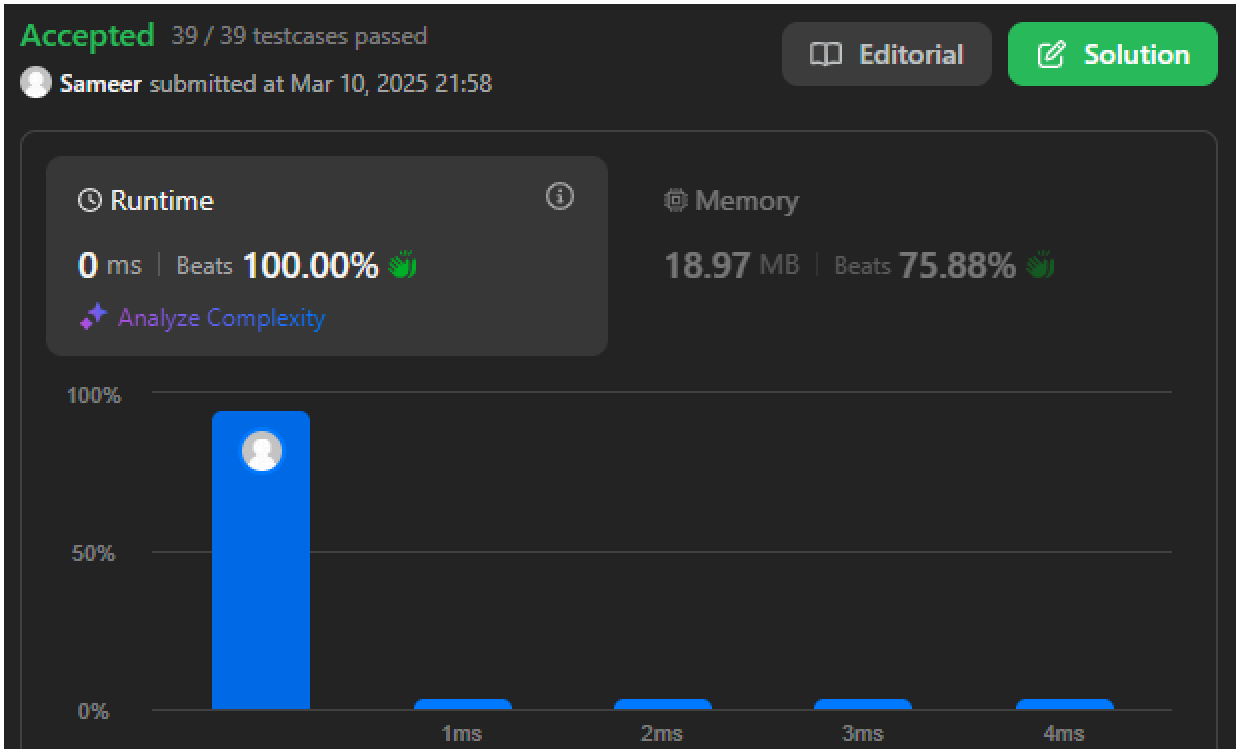
```
root =
[3,9,20,null,null,15,7]
```

Output

```
3
```

Expected

```
3
```



Accepted 39 / 39 testcases passed

Sameer submitted at Mar 10, 2025 21:58

Runtime

0 ms | Beats 100.00%

Memory

18.97 MB | Beats 75.88%

Q.2. Validate Binary Search Tree

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.

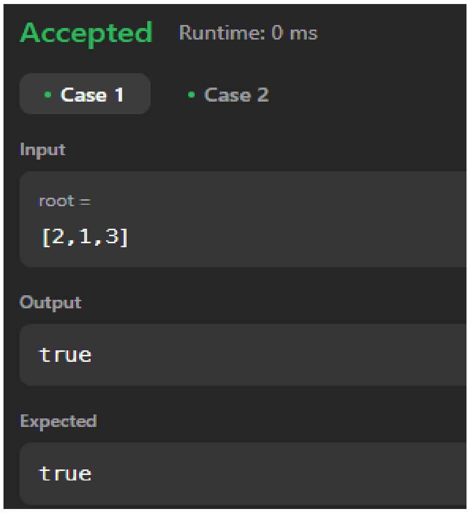
The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left and right subtrees must also be binary search trees.

Code:

```
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        return isBST(root, LONG_MIN, LONG_MAX);
    }

    bool isBST(TreeNode* node, long minVal, long maxVal) {
        if (!node) return true;
        if (node->val <= minVal || node->val >= maxVal) return false;
        return isBST(node->left, minVal, node->val) && isBST(node->right, node->val, maxVal);
    }
};
```

Output:



Accepted Runtime: 0 ms

Input

root =

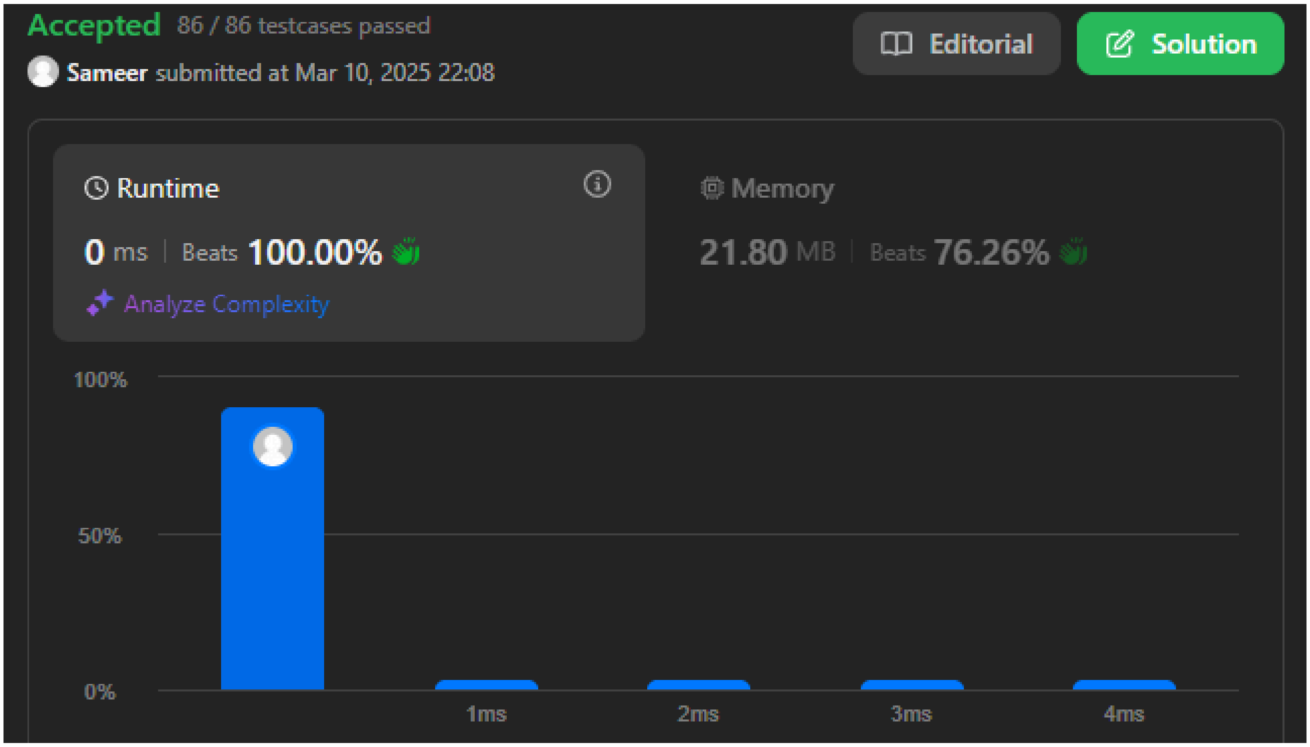
[2,1,3]

Output

true

Expected

true



Accepted 86 / 86 testcases passed

Sameer submitted at Mar 10, 2025 22:08

Runtime

0 ms | Beats 100.00%

Memory

21.80 MB | Beats 76.26%

Q.3. Symmetric Tree

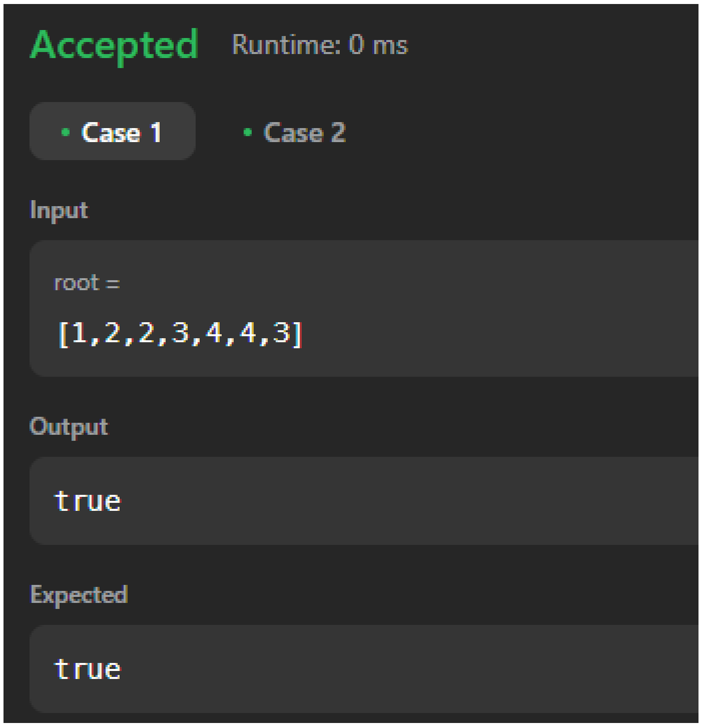
Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

Code:

```
class Solution {
public:
   bool isSymmetric(TreeNode* root) {
      if (!root) return true;
      return isMirror(root->left, root->right);
   }

   bool isMirror(TreeNode* t1, TreeNode* t2) {
      if (!t1 && !t2) return true;
      if (!t1 || !t2 || t1->val != t2->val) return false;
```

```
        return isMirror(t1->left, t2->right) && isMirror(t1->right, t2->left);
    }
};
```

Output:



Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

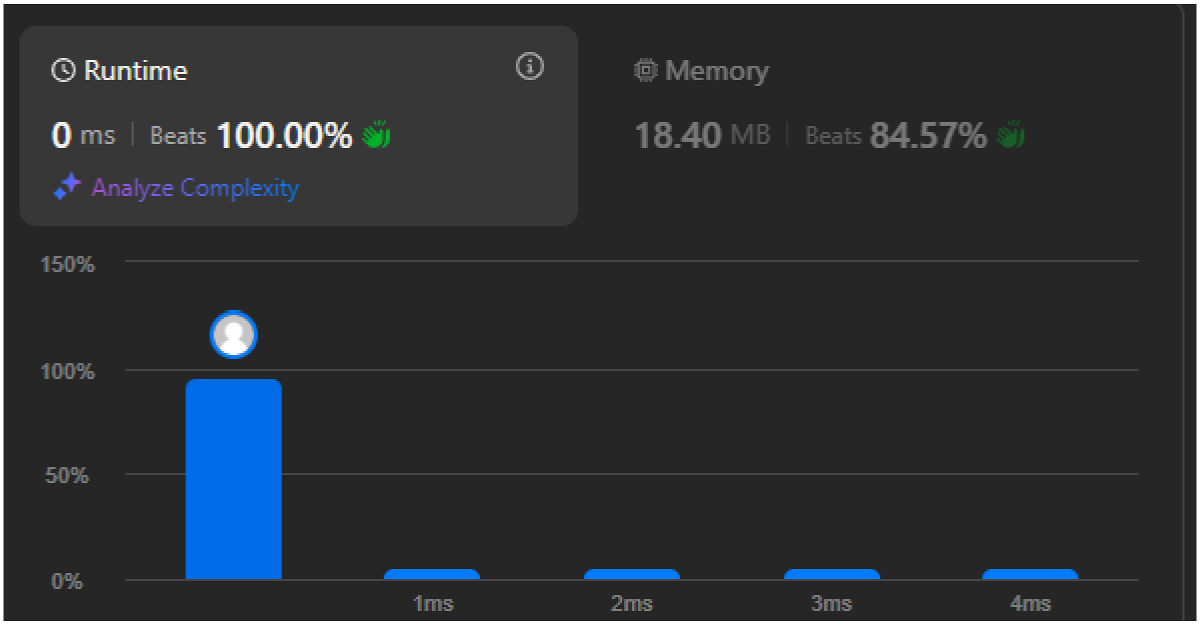
```
root =
[1,2,2,3,4,4,3]
```

Output

```
true
```

Expected

```
true
```



Runtime

0 ms | Beats 100.00%

Analyze Complexity

Memory

18.40 MB | Beats 84.57%

Q.4. Binary Tree Zigzag Level Order Traversal

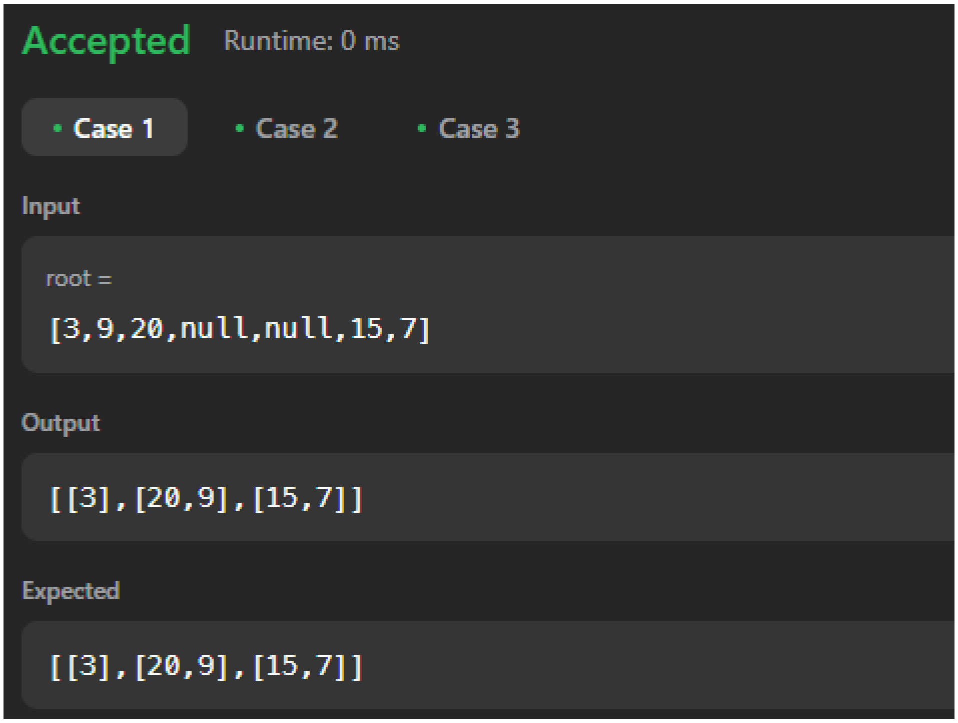
Given the root of a binary tree, return the zigzag level order traversal of its nodes' values. (i.e., from left to right, then right to left for the next level and alternate between).

Code:

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root)
    {
        vector<vector<int>> ans;

        if (root == NULL) {
            return ans;
        }

        queue<TreeNode*> q;
        q.push(root);
        bool rightToLeft = false;

        while (!q.empty())
        {
            int size = q.size();
            vector<int> level;

            for (int i = 0; i < size; i++)
            {
                TreeNode* temp = q.front();
                q.pop();

                level.push_back(temp->val);

                if (temp->left) q.push(temp->left);
```

```
                    if (temp->right) q.push(temp->right);
            }

            if (rightToLeft) {
                reverse(level.begin(), level.end());
            }

            ans.push_back(level);
            rightToLeft = !rightToLeft;
        }

        return ans;
    }
};
```

Output:



Accepted Runtime: 0 ms

Input

root =

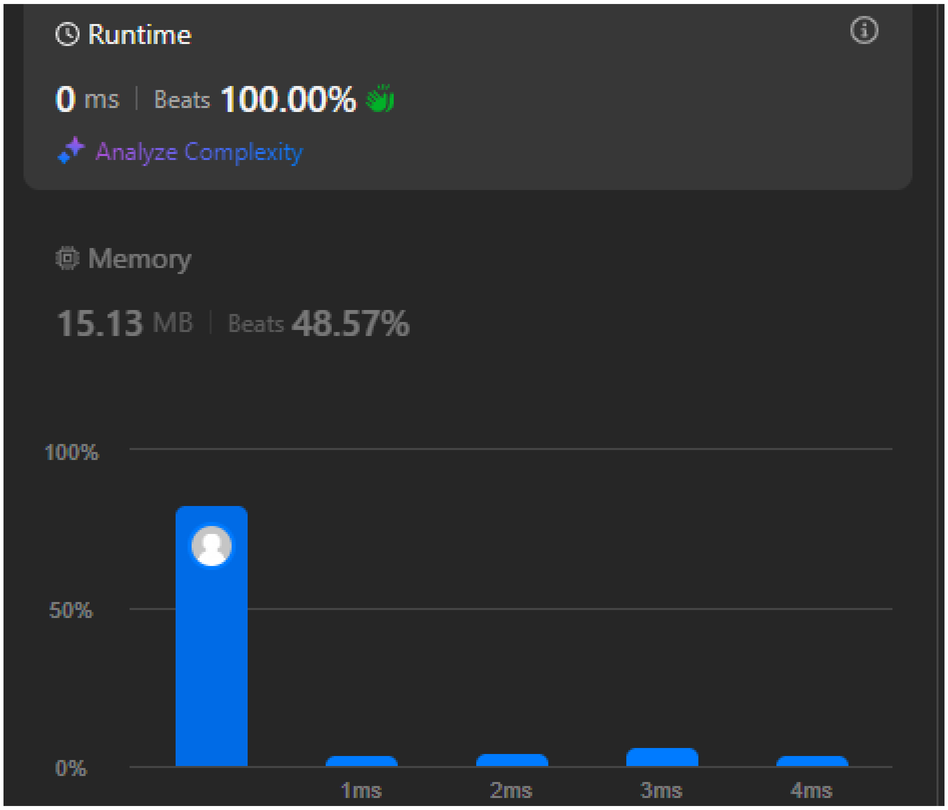
```
[3,9,20,null,null,15,7]
```

Output

```
[[3],[20,9],[15,7]]
```

Expected

```
[[3],[20,9],[15,7]]
```



Runtime

0 ms | Beats 100.00%

Memory

15.13 MB | Beats 48.57%

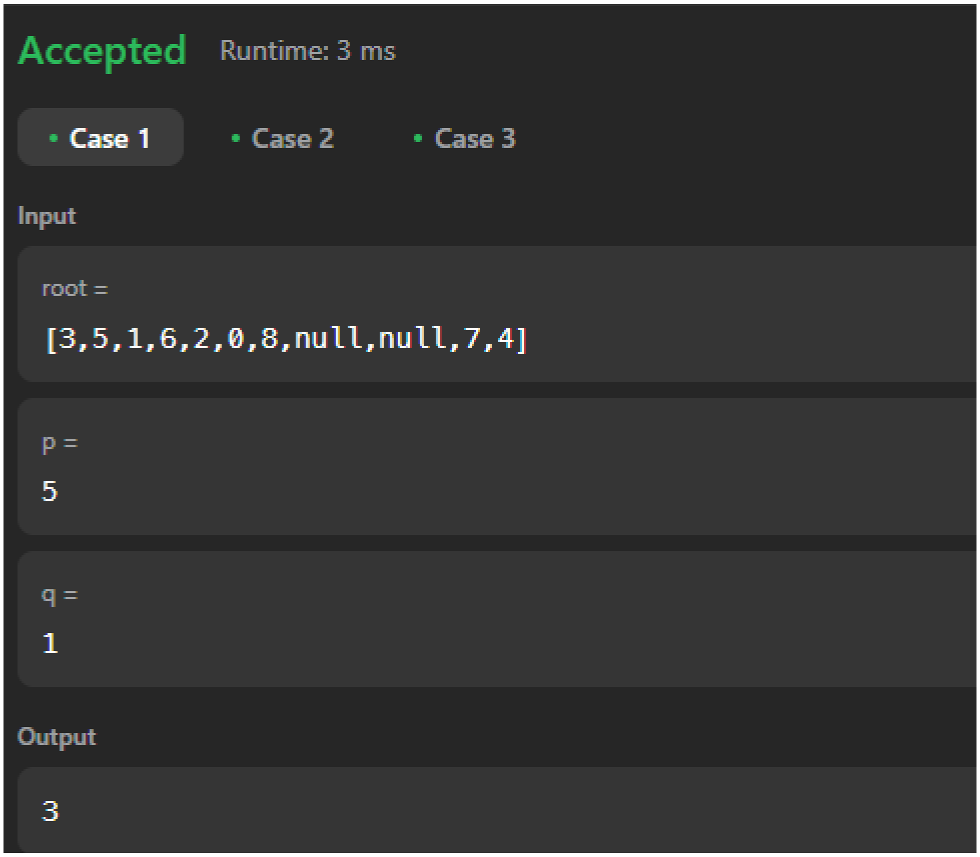
Q.5. Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Code:

```
class Solution {
public:
   TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
      if (!root || root == p || root == q) return root;
      TreeNode* left = lowestCommonAncestor(root->left, p, q);
      TreeNode* right = lowestCommonAncestor(root->right, p, q);
      if (left && right) return root;
      return left ? left : right;
   }};
```

Output:



Accepted Runtime: 3 ms

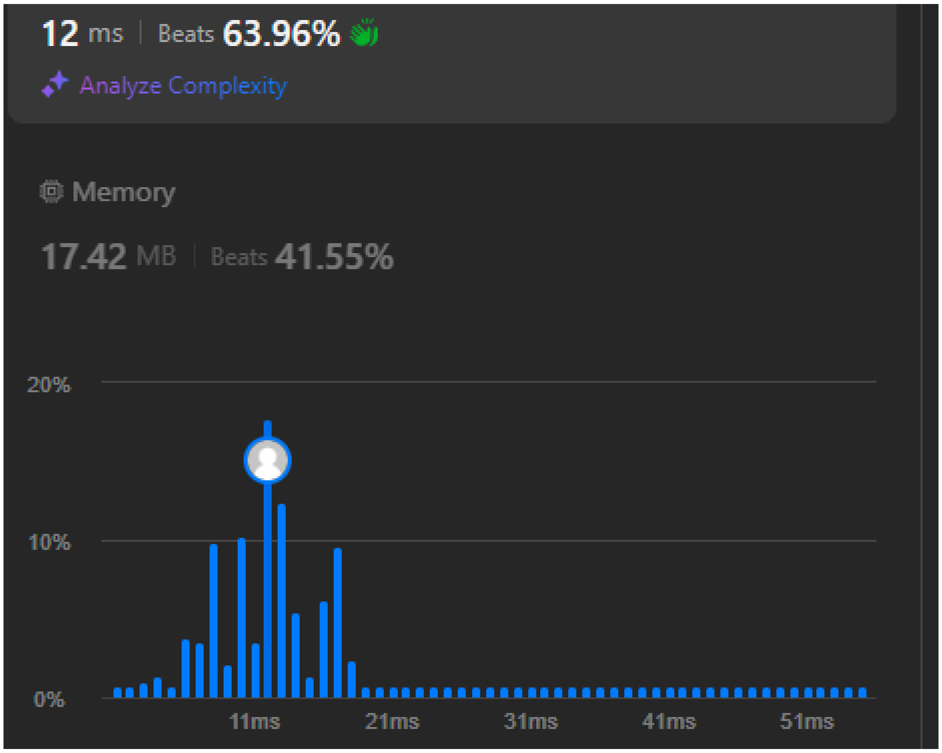
• Case 1 • Case 2 • Case 3

Input

```
root =
[3,5,1,6,2,0,8,null,null,7,4]

p =
5

q =
1
```

Output

```
3
```



12 ms | Beats 63.96%

Analyze Complexity

Memory

17.42 MB | Beats 41.55%

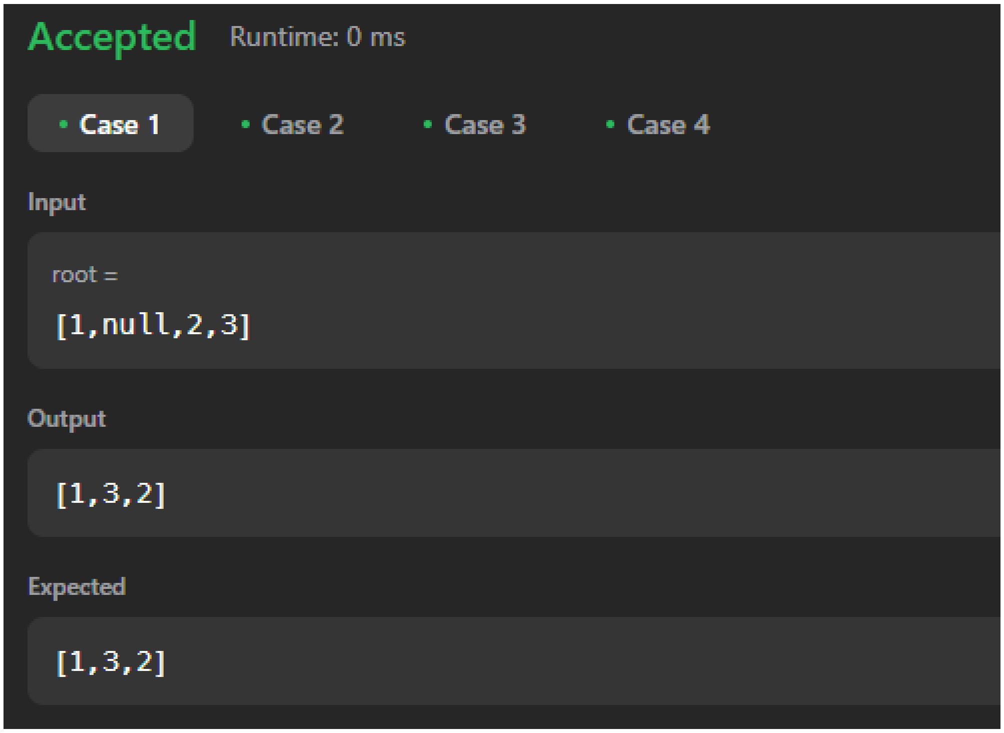
Q.6. Binary Tree Inorder Traversal

Given the root of a binary tree, return the inorder traversal of its nodes' values.

Code:

```
class Solution {
 public:
    void traverse(TreeNode* root, vector<int> &ans)
    {
        if (root == NULL)
        {
            return ;
        }

        traverse(root->left, ans) ;
        ans.push_back(root->val) ;
        traverse(root->right, ans) ;
    }

    vector<int> inorderTraversal(TreeNode* root)
    {
        vector<int> ans ;
        traverse(root, ans) ;
        return ans ;
    }
 };
```

Output:



Accepted Runtime: 0 ms

Input

root =

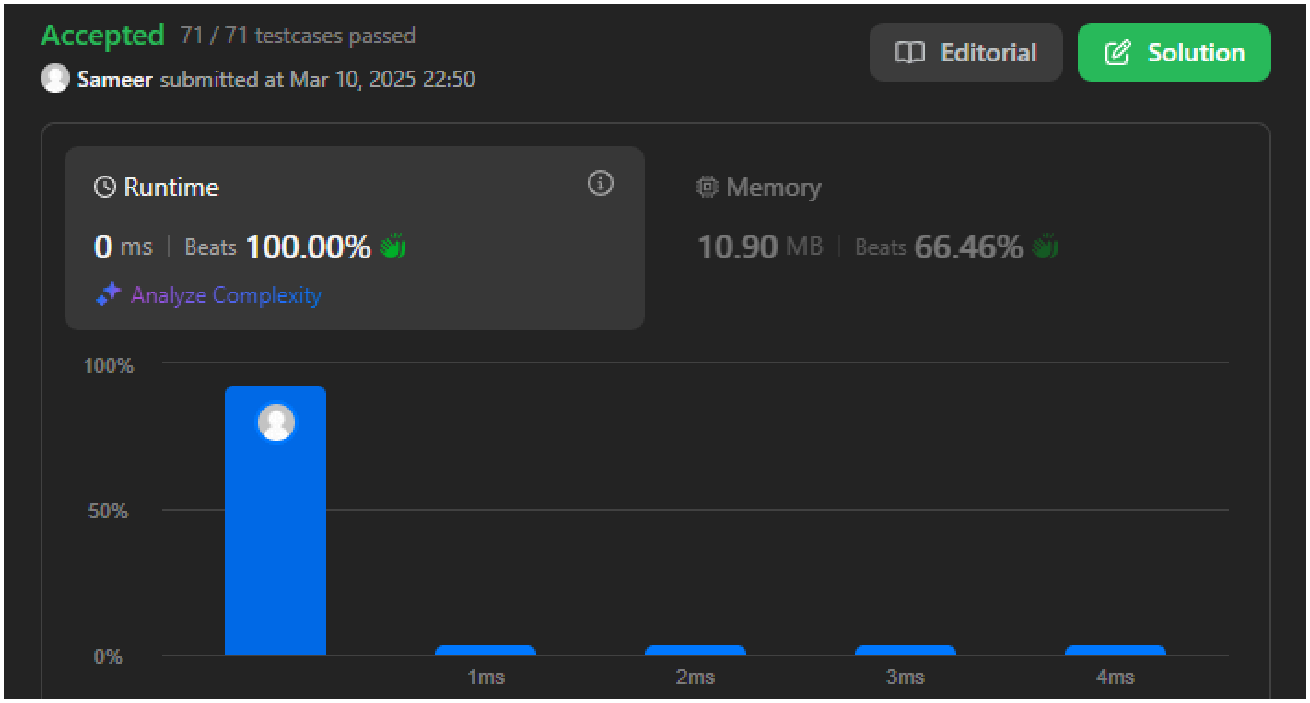
```
[1,null,2,3]
```

Output

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

Expected

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



Accepted 71 / 71 testcases passed

Sameer submitted at Mar 10, 2025 22:50

Runtime

0 ms | Beats 100.00%

Analyze Complexity

Memory

10.90 MB | Beats 66.46%

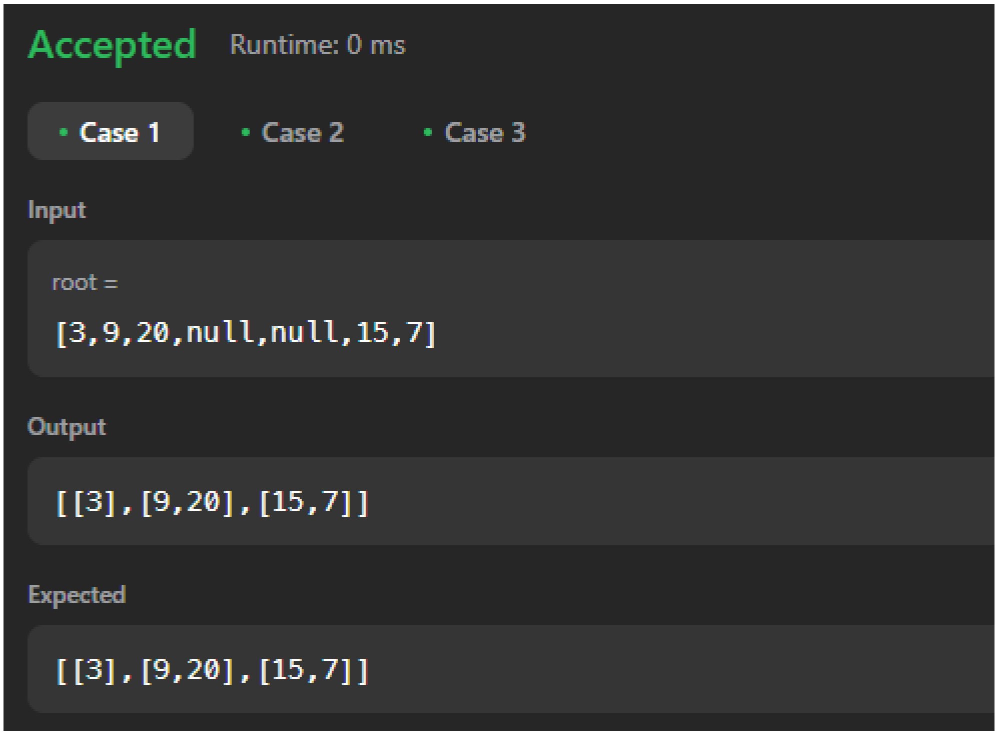
Q.7. Binary Tree Level Order Traversal

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).

Code:

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root)
    {
        vector<vector<int>> ans ;
        vector<int> level ;

        if (root == NULL)
        {
            return ans ;
        }

        queue<TreeNode*> q ;
        q.push(root) ;
        q.push(NULL) ;

        while(!q.empty())
        {
            TreeNode* temp = q.front() ;
            q.pop() ;

            if (temp == NULL)
            {
                ans.push_back(level) ;
                level.clear() ;
                if (!q.empty())
                {
                    q.push(NULL) ;
                }
            }

            else
            {
                level.push_back(temp->val) ;

                if (temp->left != NULL)
                {
                    q.push(temp->left) ;
                }

                if (temp->right != NULL)
                {
                    q.push(temp->right) ;
                }
            }
        }

        return ans ;
    }
};
```

Output:



Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

root =

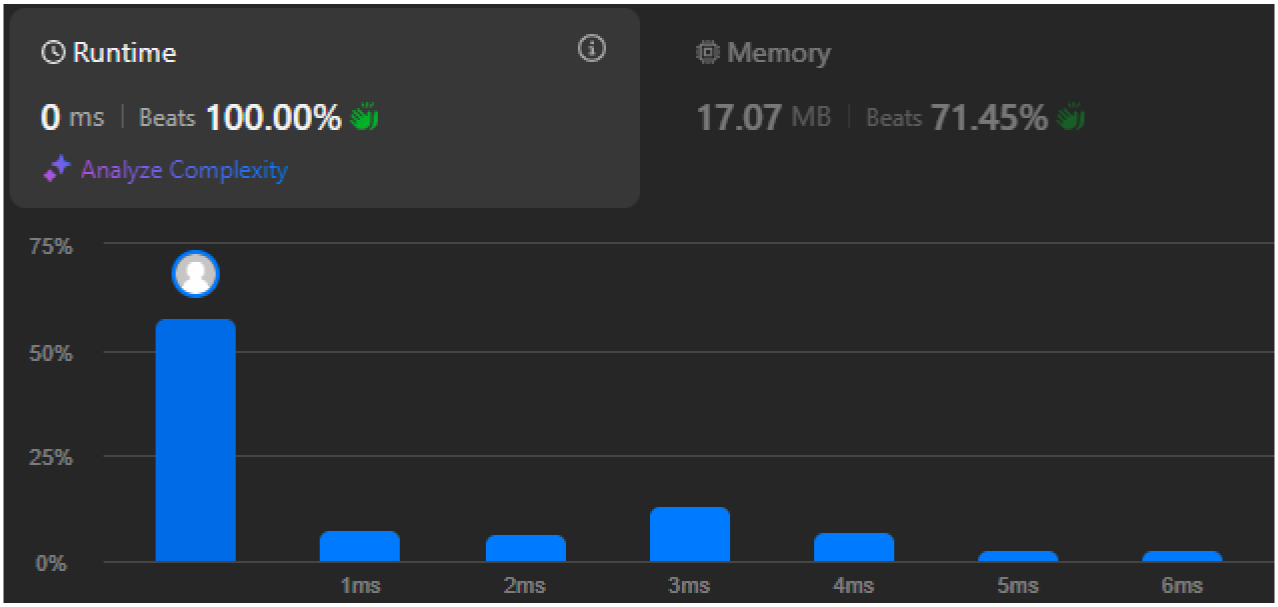
```
[3,9,20,null,null,15,7]
```

Output

```
[[3],[9,20],[15,7]]
```

Expected

```
[[3],[9,20],[15,7]]
```



Runtime

0 ms | Beats 100.00%

Analyze Complexity

Memory

17.07 MB | Beats 71.45%

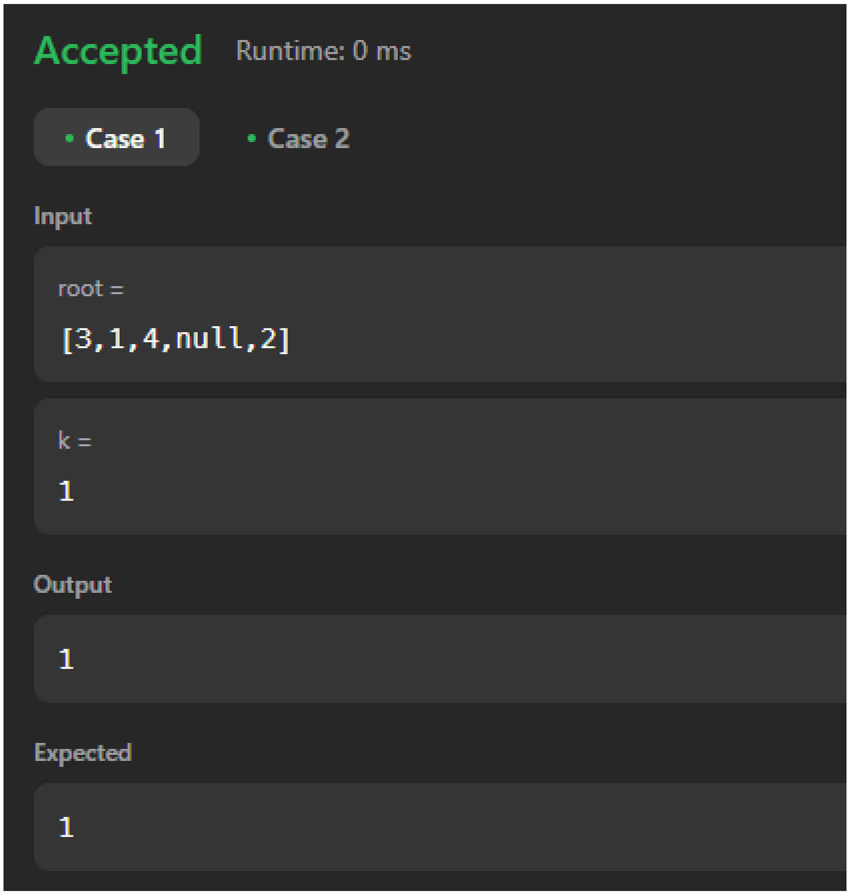
Q.8. Kth Smallest Element in a BST

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.

Code:

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        stack<TreeNode*> s;
        while (true) {
            while (root) {
                s.push(root);
                root = root->left;
            }
            root = s.top();
            s.pop();
            if (--k == 0) return root->val;
            root = root->right;
        }
    }
};
```

Output:



Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

root =

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

k =

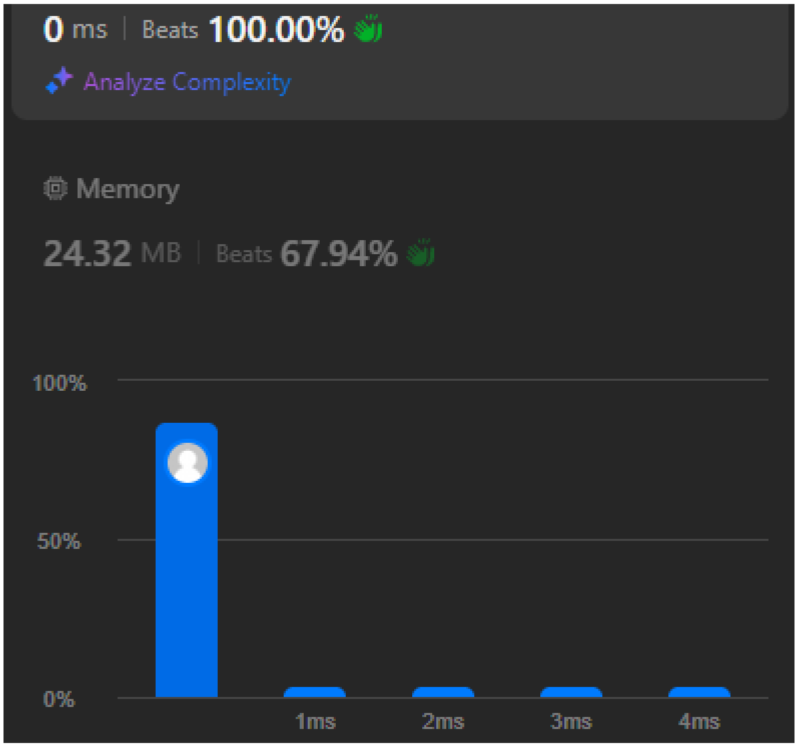
```
1
```

Output

```
1
```

Expected

```
1
```



0 ms | Beats 100.00%

Analyze Complexity

Memory

24.32 MB | Beats 67.94%

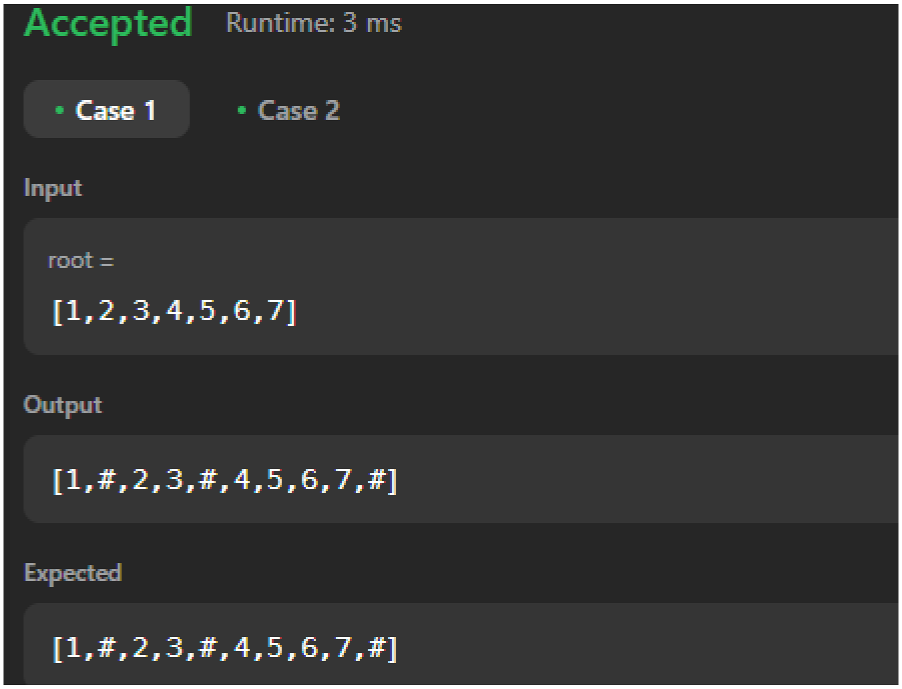
Q.9. Populating Next Right Pointers in Each Node

You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL. Initially, all next pointers are set to NULL.

Code:

```
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return root;
        Node* leftmost = root;
        while (leftmost->left) {
            Node* head = leftmost;
            while (head) {
                head->left->next = head->right;
                if (head->next) head->right->next = head->next->left;
                head = head->next;
            }
            leftmost = leftmost->left;
        }
        return root;
    }
};
```

Output:



Accepted Runtime: 3 ms

• Case 1 • Case 2

Input

root =

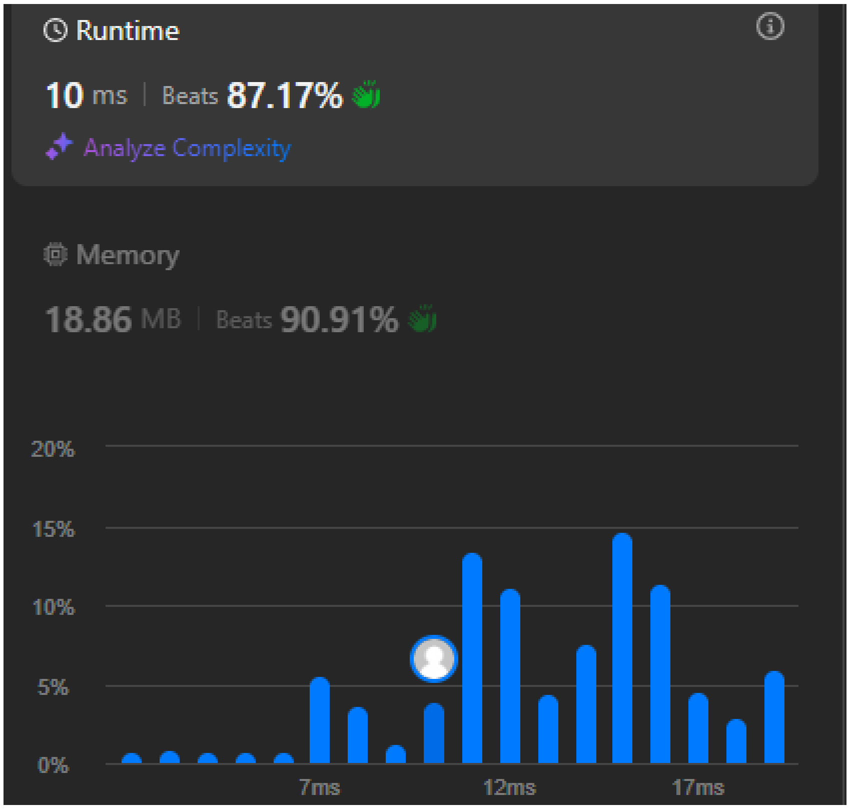
```
[1,2,3,4,5,6,7]
```

Output

```
[1,#,2,3,#,4,5,6,7,#]
```

Expected

```
[1,#,2,3,#,4,5,6,7,#]
```



Runtime

10 ms | Beats 87.17%

Analyze Complexity

Memory

18.86 MB | Beats 90.91%

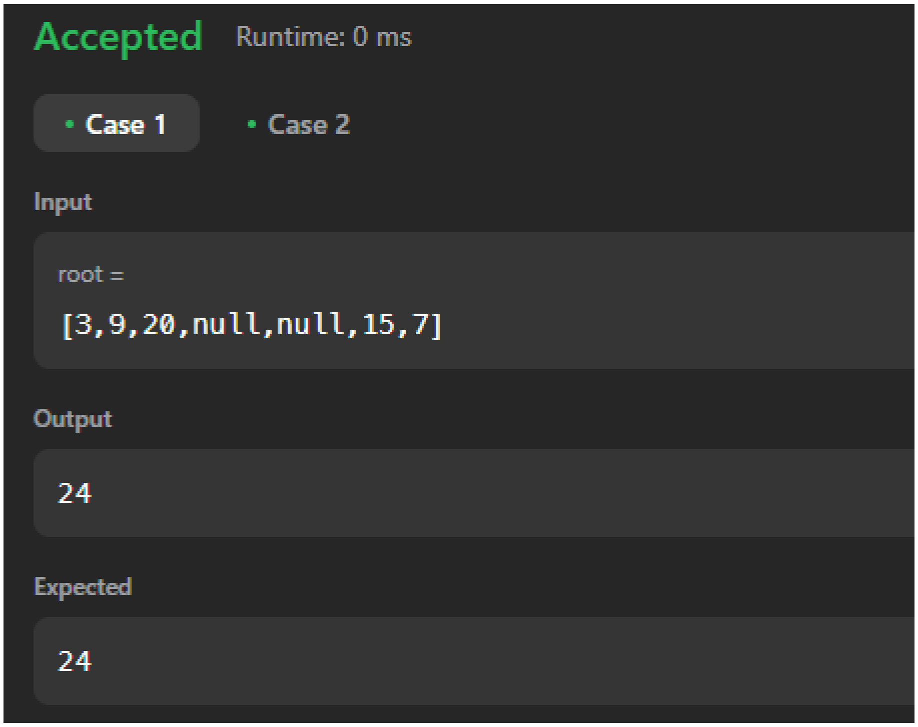
Q.10. Sum of Left Leaves

Given the root of a binary tree, return the sum of all left leaves. A leaf is a node with no children. A left leaf is a leaf that is the left child of another node.

Code:

```
class Solution {
public:
    int sumOfLeftLeaves(TreeNode* root) {
        if (!root) return 0;
        int sum = 0;
        if (root->left && !root->left->left && !root->left->right)
            sum += root->left->val;
        return sum + sumOfLeftLeaves(root->left) + sumOfLeftLeaves(root->right);
    }
};
```

Output:



Accepted Runtime: 0 ms

Input

root =

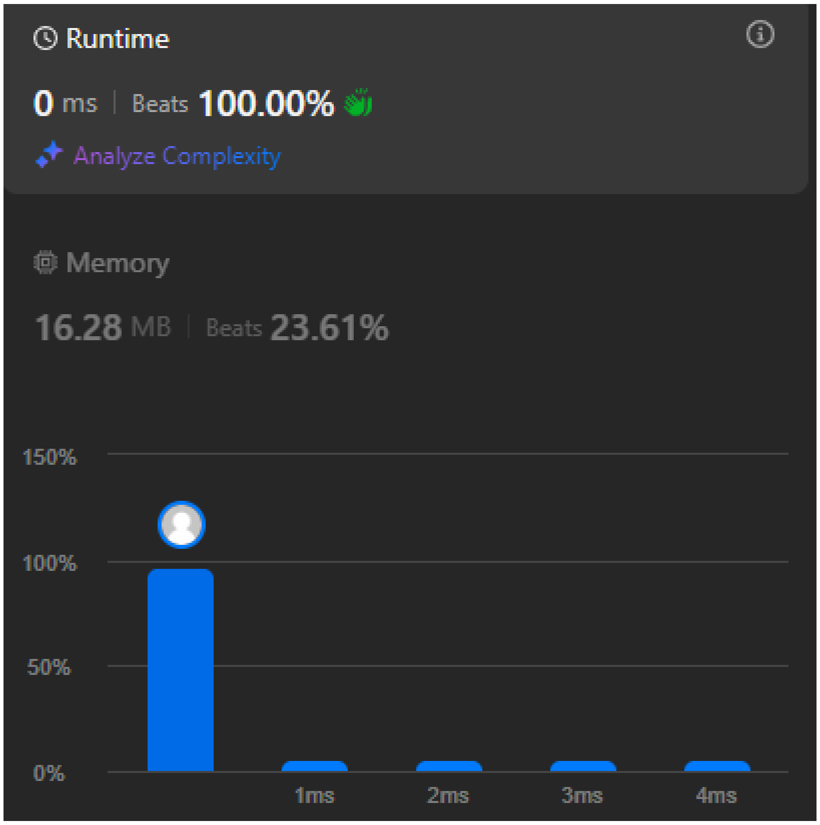
```
[3,9,20,null,null,15,7]
```

Output

```
24
```

Expected

```
24
```



Runtime

0 ms | Beats 100.00%

Analyze Complexity

Memory

16.28 MB | Beats 23.61%