Worksheet 3

Student Name: Palak **UID:** 22CS12960

Branch:CSE **Section/Group:**605-B

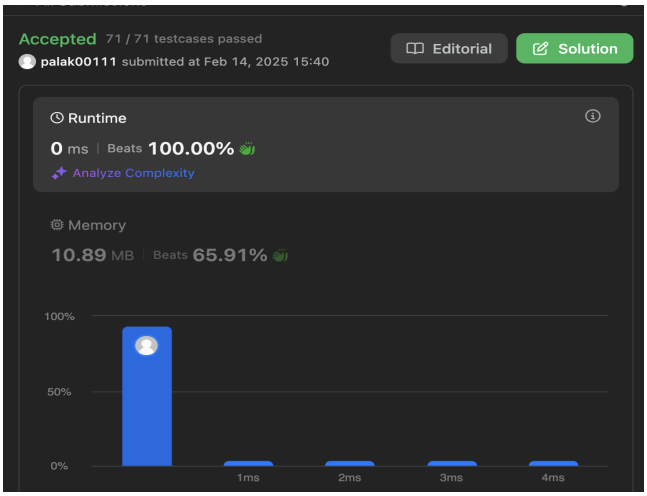
Semester: 5 **Date of Performance:** 14/02/25

Subject Name: AP **Subject Code:** 22CSP-351

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

```
class Solution {
public:
    void inorderTraversalHelper(TreeNode* root, vector<int>& result) {
    if (!root) return;
    inorderTraversalHelper(root->left, result);
    result.push_back(root->val);
    inorderTraversalHelper(root->right, result);
}

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

};

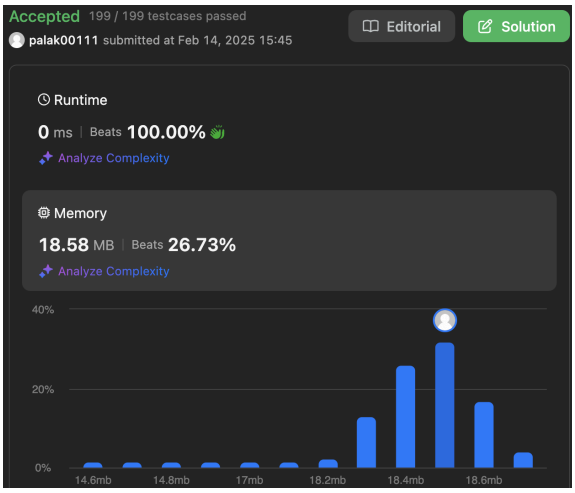


2.

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

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

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

```
}
```

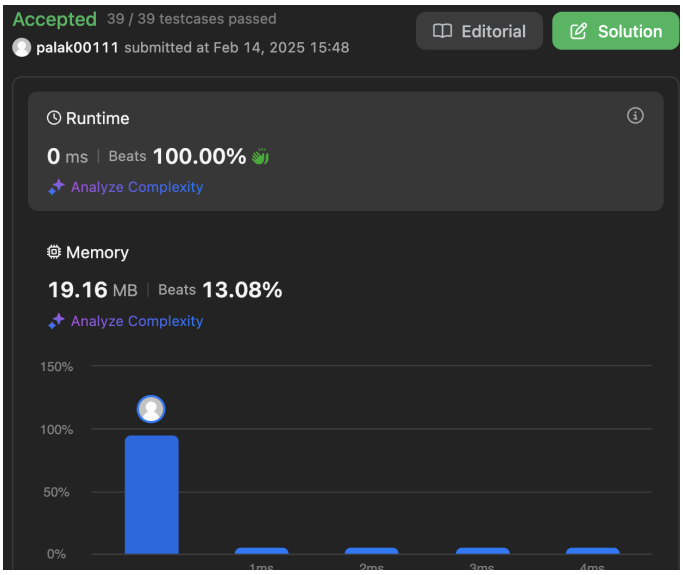


3.

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.

```
class Solution {

public:

    int maxDepth(TreeNode* root) {

    if (!root) return 0;  // Base case: if the node is null, depth is 0

    int leftDepth = maxDepth(root->left);

    int rightDepth = maxDepth(root->right);

    return 1 + max(leftDepth, rightDepth);  // Add 1 for the current node

}

};
```



4. 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

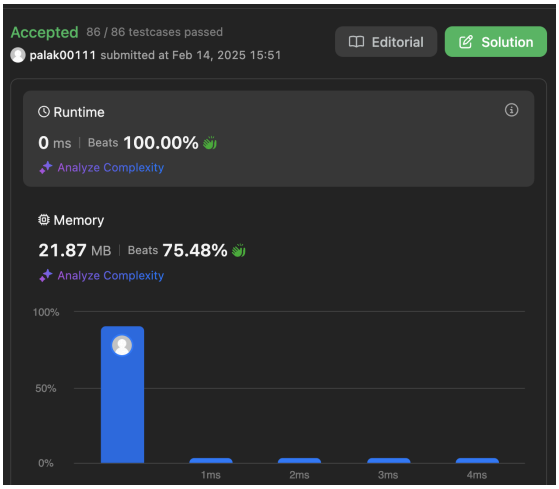
   of a node contains only nodes with keys less than the node's key.

   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.

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

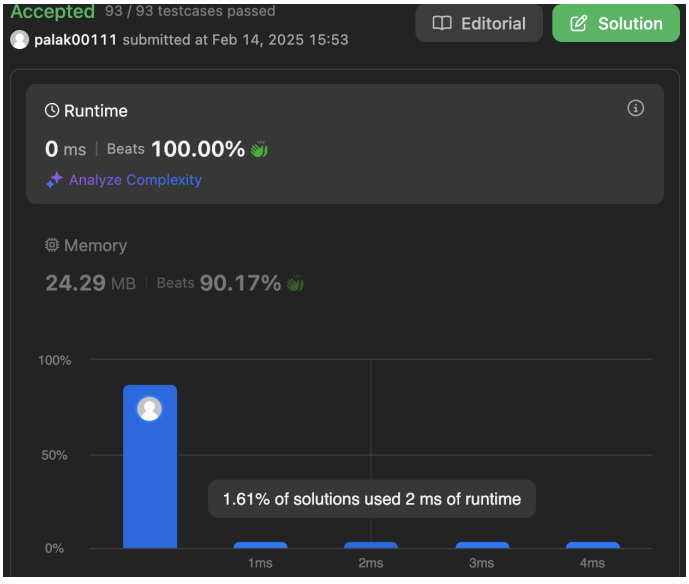
```
bool isValidBST(TreeNode* root) {
    return isValidBSTHelper(root, LONG_MIN, LONG_MAX);
}
```



5. 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.

```
class Solution {
public:
    int count = 0;
    int result = 0;

    void inorder(TreeNode* root, int k) {
        if (!root) return;
        inorder(root->left, k);
        count++;
        if (count == k) {
```

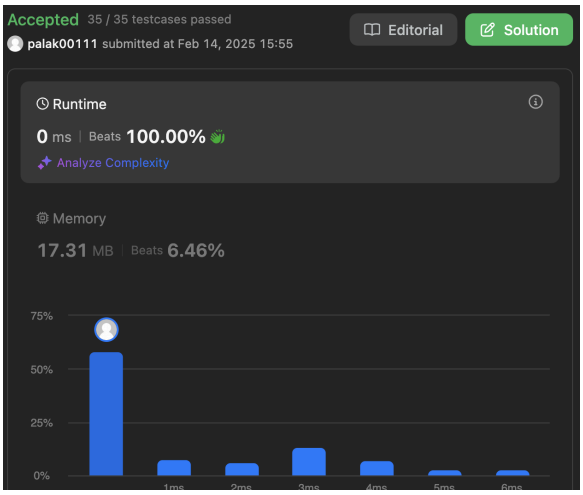
```
            result = root->val;
            return;
        }
        inorder(root->right, k);
    }

    int kthSmallest(TreeNode* root, int k) {
        inorder(root, k);
        return result;
    }
};
```



6. 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).

```
vector<vector<int>> levelOrder(TreeNode* root) {
```

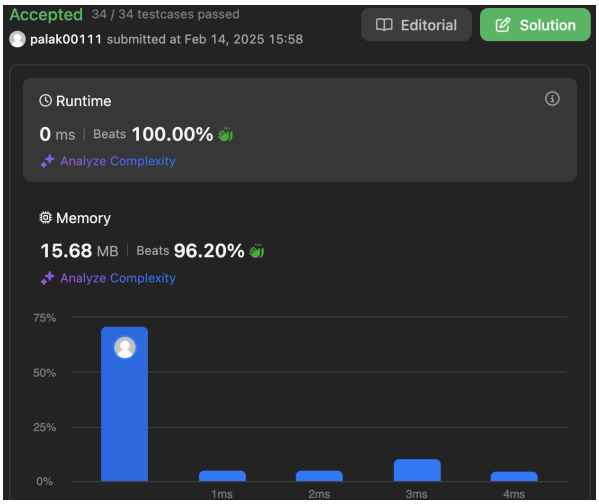
```
    vector<vector<int>> result;
    if (!root) return result;

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

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

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();
            currentLevel.push_back(node->val);

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
        result.push_back(currentLevel);
    }
    return result;
}
```



7. Given the root of a binary tree, return the bottom-up level order traversal of its nodes' values. (i.e., from left to right, level by level from leaf to root).

```
vector<vector<int>> levelOrderBottom(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;
    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        int levelSize = q.size();
        vector<int> currentLevel;

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();
            currentLevel.push_back(node->val);
```

```
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
        result.push_back(currentLevel);
    }
    reverse(result.begin(), result.end());  // Reverse the result for bottom-up order
    return result;
}
```



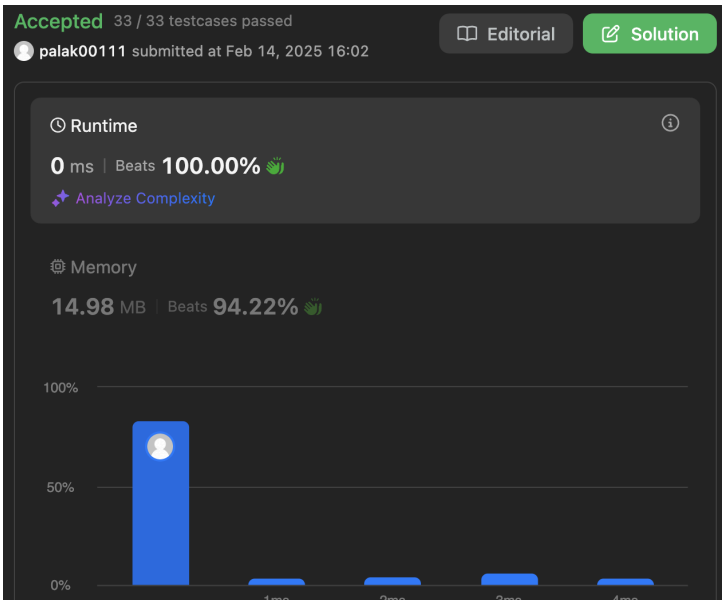
8.

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).

```
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;
    queue<TreeNode*> q;
    q.push(root);
```

```
bool leftToRight = true;

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

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

        // Determine position based on direction
        int index = leftToRight ? i : (levelSize - 1 - i);
        currentLevel[index] = node->val;

        if (node->left) q.push(node->left);
        if (node->right) q.push(node->right);
    }

    result.push_back(currentLevel);
    leftToRight = !leftToRight;  // Switch direction for the next level
}

return result;
```

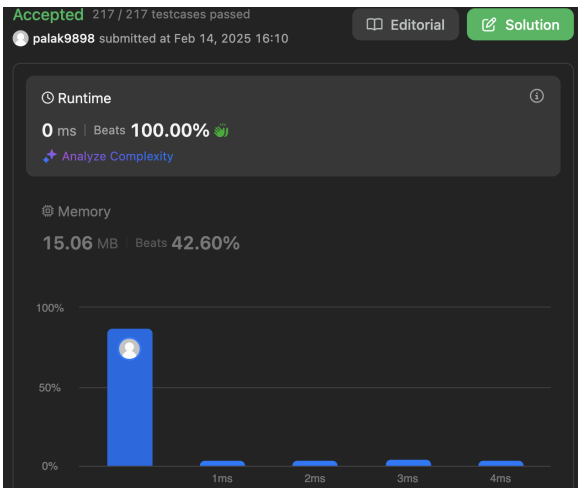
```
}
```



9. Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

```
vector<int> rightSideView(TreeNode* root) {
    vector<int> result;
    if (!root) return result;

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

    while (!q.empty()) {
        int levelSize = q.size();
        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
```

```
            q.pop();

            // Capture the last node of each level
            if (i == levelSize - 1) {
                result.push_back(node->val);
            }

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }

    return result;
}
```

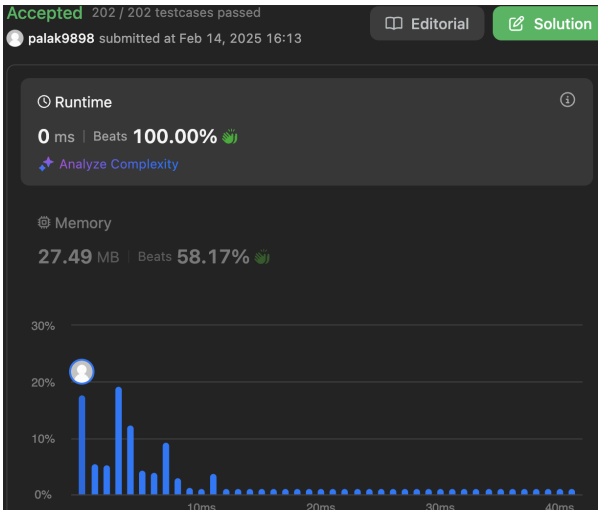


10.

Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return the binary tree.

```
 class Solution {

public:

    unordered_map<int, int> inorderIndexMap;  // To store the index of each value in the inorder array

    TreeNode* buildTreeHelper(vector<int>& inorder, vector<int>& postorder, int inorderStart, int inorderEnd, int& postorderIndex) {

        if (inorderStart > inorderEnd) return nullptr;

        int rootVal = postorder[postorderIndex--];  // Get the root value from postorder

        TreeNode* root = new TreeNode(rootVal);

        int inorderIndex = inorderIndexMap[rootVal];  // Find the index of the root in inorder array

        // Build the right subtree first (since we are traversing postorder from end to start)

        root->right = buildTreeHelper(inorder, postorder, inorderIndex + 1, inorderEnd, postorderIndex);

        // Build the left subtree

        root->left = buildTreeHelper(inorder, postorder, inorderStart, inorderIndex - 1, postorderIndex);

        return root;

    }



    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {

        // Map each value in the inorder array to its index for quick lookup

        for (int i = 0; i < inorder.size(); i++) {
```

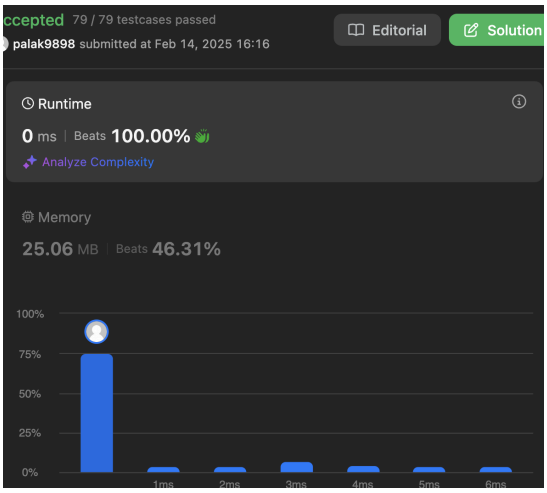
```
            inorderIndexMap[inorder[i]] = i;
        }
        int postorderIndex = postorder.size() - 1;  // Start from the end of the postorder array
        return buildTreeHelper(inorder, postorder, 0, inorder.size() - 1, postorderIndex);
    }
};
```



11. Given the root of a binary tree, return the leftmost value in the last row of the tree.

```
int findBottomLeftValue(TreeNode* root) {
    queue<TreeNode*> q;
    q.push(root);
    int leftmostValue = root->val;
```

```
    while (!q.empty()) {
        int size = q.size();  // Number of nodes at the current level
        for (int i = 0; i < size; i++) {
            TreeNode* node = q.front();
            q.pop();
            // The first node at each level is the leftmost one
            if (i == 0) {
                leftmostValue = node->val;
            }
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }

    return leftmostValue;
}
```



12.

Given the root of a binary tree, calculate the vertical order traversal of the binary tree.

For each node at position (row, col), its left and right children will be at positions (row + 1, col - 1) and (row + 1, col + 1) respectively. The root of the tree is at (0, 0).

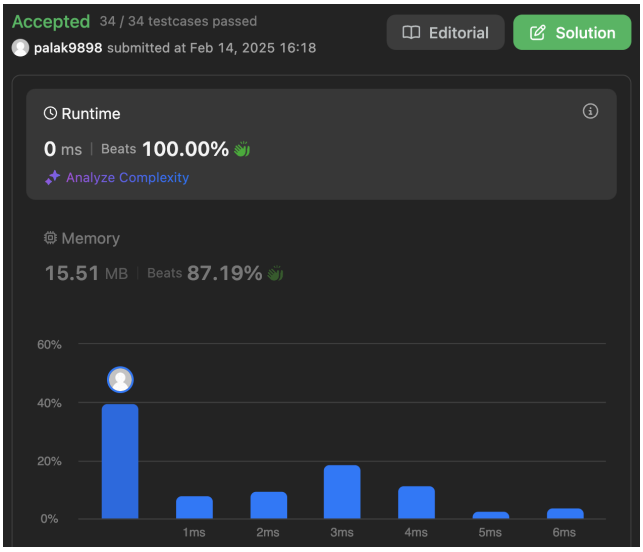
The vertical order traversal of a binary tree is a list of top-to-bottom orderings for each column index starting from the leftmost column and ending on the rightmost column. There may be multiple nodes in the same row and same column. In such a case, sort these nodes by their values.

Return the vertical order traversal of the binary tree.

```
class Solution {

public:

    vector<vector<int>> verticalTraversal(TreeNode* root) {

        // Map to store nodes by column -> (row, value)

        map<int, vector<pair<int, int>>> columnTable;

        queue<tuple<TreeNode*, int, int>> q;  // (node, row, column)
```

```
q.push({root, 0, 0});

while (!q.empty()) {
    auto [node, row, col] = q.front();
    q.pop();
    columnTable[col].emplace_back(row, node->val);

    if (node->left) {
        q.push({node->left, row + 1, col - 1});
    }
    if (node->right) {
        q.push({node->right, row + 1, col + 1});
    }
}

vector<vector<int>> result;
for (auto& [col, nodes] : columnTable) {
    // Sort nodes first by row, then by value
    sort(nodes.begin(), nodes.end(), [](const pair<int, int>& a, const pair<int, int>& b) {
        if (a.first == b.first) return a.second < b.second;
        return a.first < b.first;
    });
    // Extract the sorted values
```

```
            vector<int> sortedColumn;
            for (auto& [row, value] : nodes) {
                sortedColumn.push_back(value);
            }
            result.push_back(sortedColumn);
        }

        return result;
    }
};
```



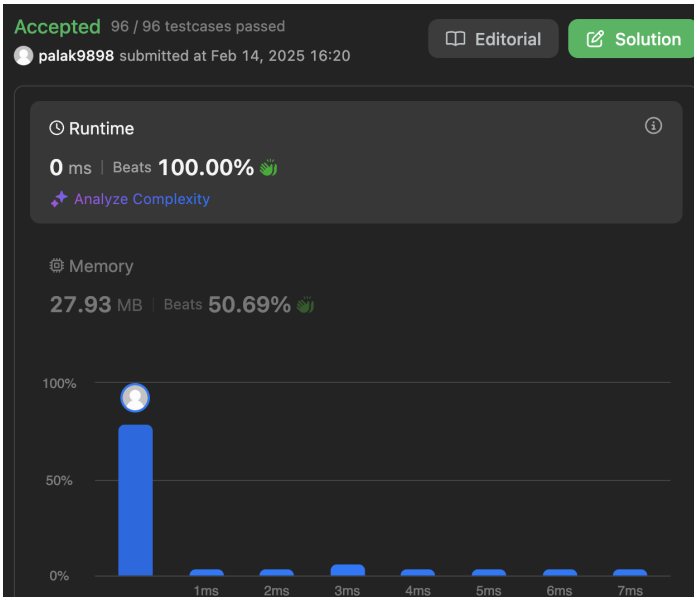
13. A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence at most once. Note that the path does not need to pass through the root.

    The path sum of a path is the sum of the node's values in the path.

Given the root of a binary tree, return the maximum path sum of any non-empty path.

```
class Solution {
public:
    int maxPathSum(TreeNode* root) {
        int maxSum = INT_MIN;
        calculateMaxPathSum(root, maxSum);
        return maxSum;
    }

private:
    int calculateMaxPathSum(TreeNode* node, int& maxSum) {
        if (!node) return 0;

        // Calculate max path sum for left and right subtrees
        int leftSum = max(0, calculateMaxPathSum(node->left, maxSum));
        int rightSum = max(0, calculateMaxPathSum(node->right, maxSum));

        // Update maxSum with the current node as the highest node in the path
        maxSum = max(maxSum, leftSum + rightSum + node->val);

        // Return the maximum path sum including the current node and one of its subtrees
        return node->val + max(leftSum, rightSum);
```

```
    }
};
```


Accepted 96 / 96 testcases passed
palak9898 submitted at Feb 14, 2025 16:20
Editorial
Solution
Runtime
0 ms | Beats 100.00%
Analyze Complexity
Memory
27.93 MB | Beats 50.69%
100%
50%
0%
1ms
2ms
3ms
4ms
5ms
6ms
7ms