# Assignment 3

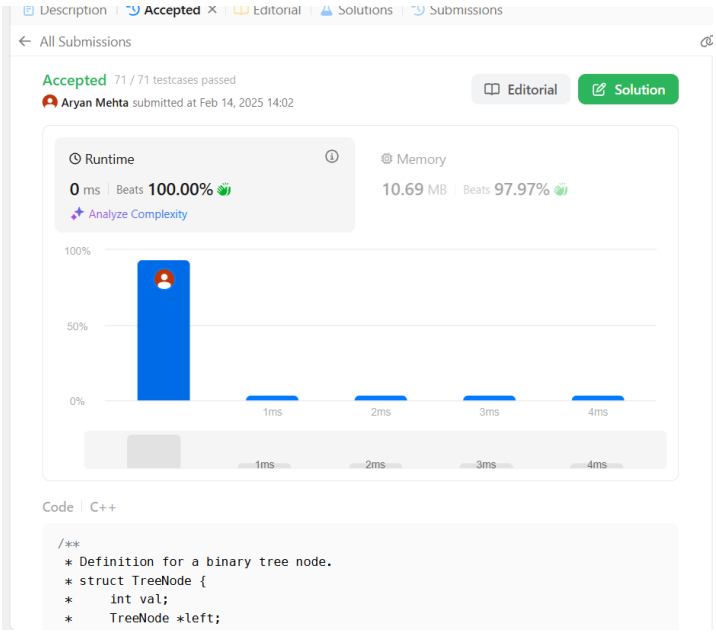
**Student Name: Aryan Mehta** **UID:22BCS12209**
**Branch: BE-CSE (General)** **Section/Group: FL_IOT-602 A**
**Semester:6th** **Date of Performance: 14-02-25**
**Subject Name: Advanced Programming Lab-2** **Subject Code: 22CSP-351**

## 1. Aim: 94. Binary tree In-order Traversal

### Implementation/ Code:

```
class Solution {
public:
void inorderTraversalHelper(TreeNode* root, vector<int>& result) {
if (root == nullptr) 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;
}
};
```

**Output:**

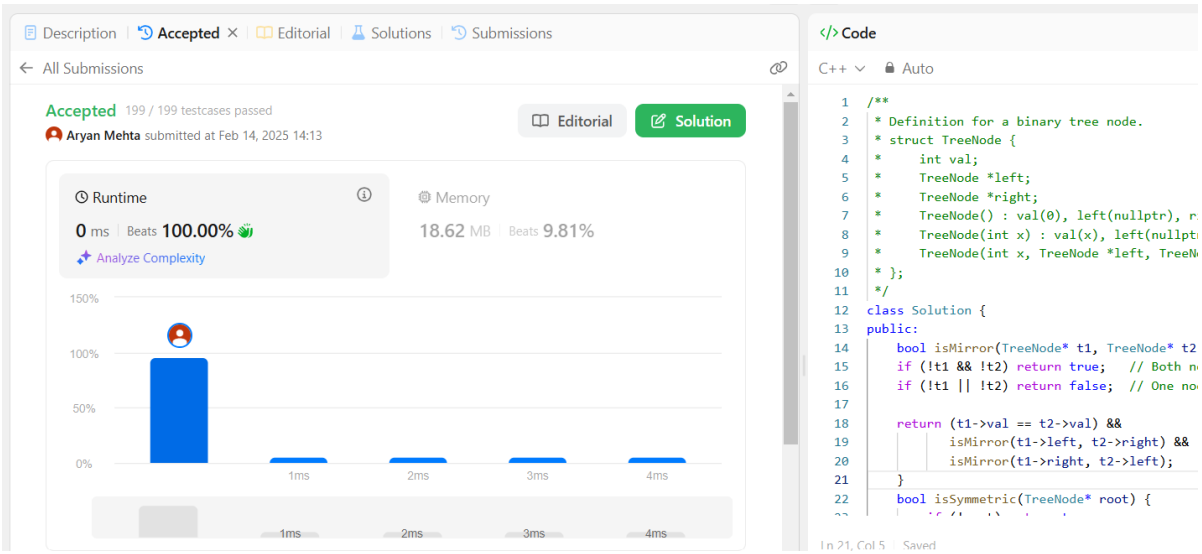



## 1. Aim: 101. Symmetric Tree

### Implementation/ Code:

```
class Solution {
    public:
    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);    }
};
```

**Output:**

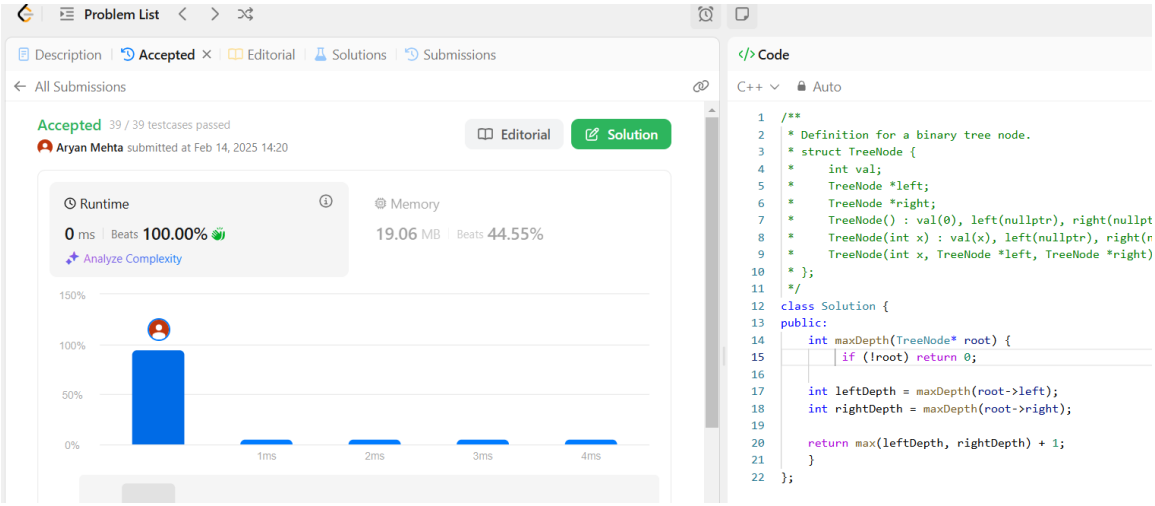



**2. Aim:** 104. Maximum Depth of Binary Tree

**Implementation/ Code:**

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (!root) return 0;
        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);
        return max(leftDepth, rightDepth) + 1;
    }
};
```

**Output:**

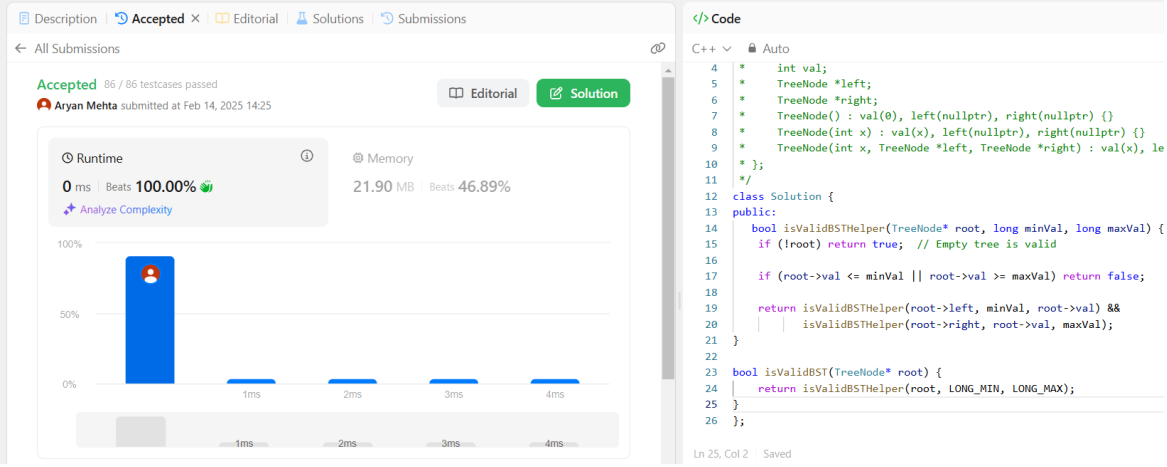



**3. Aim:** 98. Validate Binary Search Tree

## Implementation/ Code:

```
class Solution {
public:
  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);
}
};
```

## Output:

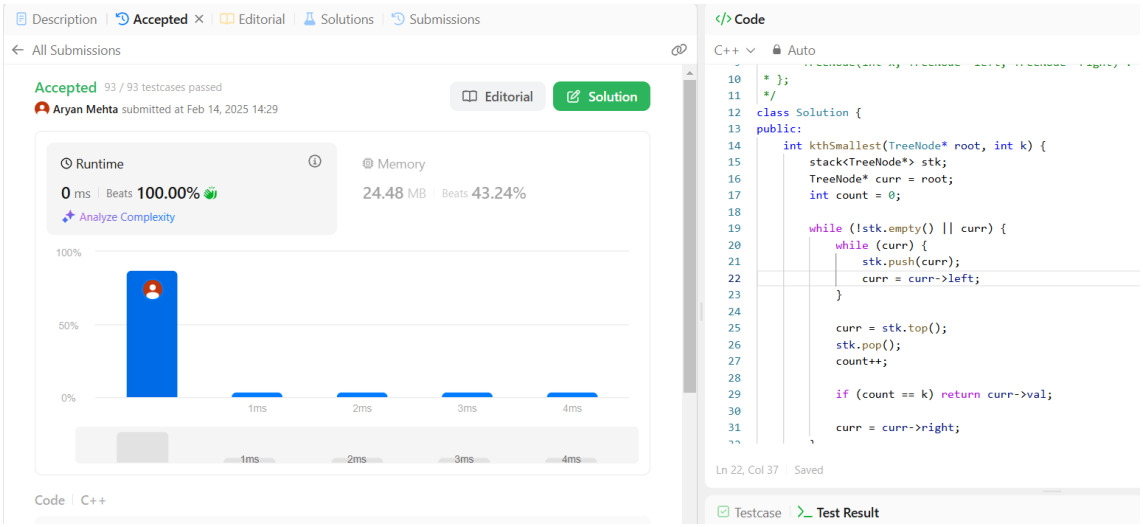



**4. Aim:** 230. Kth Smallest Element in a BST

## Implementation/ Code:

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        stack<TreeNode*> stk;
        TreeNode* curr = root;
        int count = 0;
        while (!stk.empty() || curr) {
            while (curr) {
                stk.push(curr);
                curr = curr->left;
            }
            curr = stk.top();
            stk.pop();
            count++;
            if (count == k) return curr->val;
            curr = curr->right;
        }
        return -1;
    }
};
```

## Output:

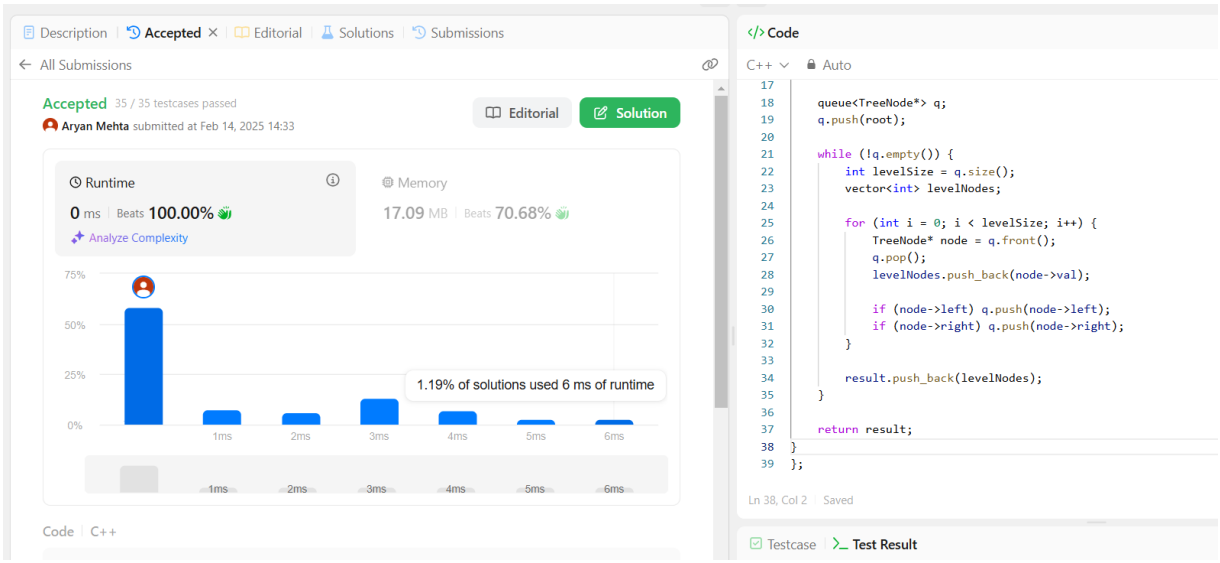



**5. Aim:** 102. Binary Tree Level Order Traversal

## Implementation/ Code:

```
class Solution {
public:
    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> levelNodes;
        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();
            levelNodes.push_back(node->val);
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
        result.push_back(levelNodes);
    }
    return result;
}
};
```

## Output:

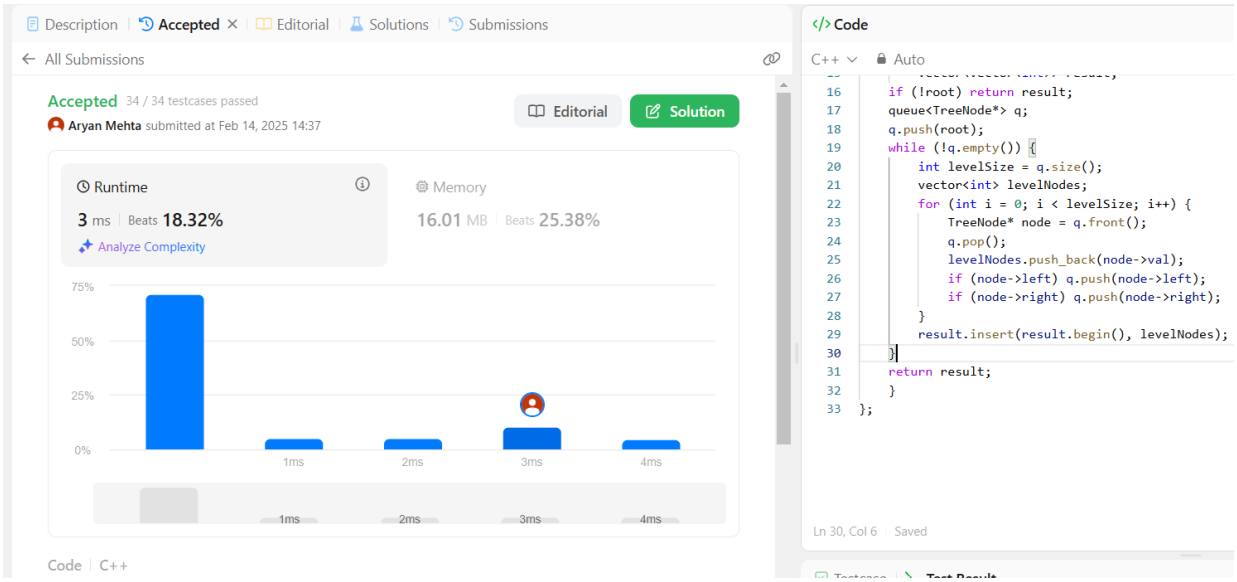



**6. Aim:** 107. Binary Tree Level Order Traversal II

## Implementation/ Code:

```
class Solution {
public:
    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> levelNodes;
        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();
            levelNodes.push_back(node->val);
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
        result.insert(result.begin(), levelNodes); // Insert at beginning
    }
    return result;
    }
     };
```

## Output:



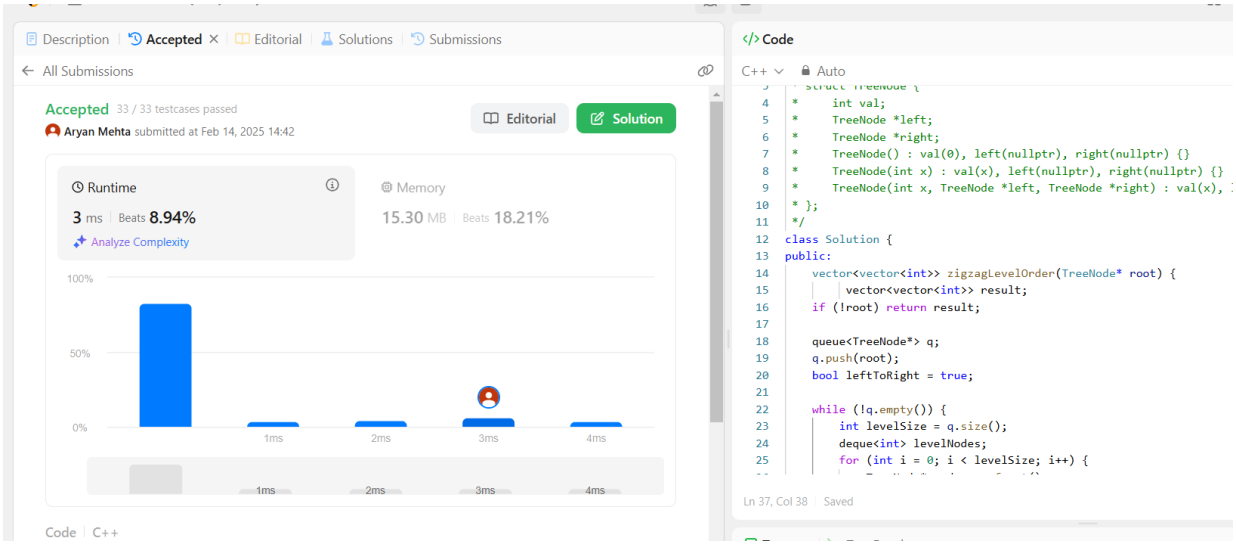

**7. Aim:** Binary Tree ZigZag Level Order Traversal

## Implementation/ Code:

```
class Solution {
public:
    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();
        deque<int> levelNodes;
        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();
            if (leftToRight) {
                levelNodes.push_back(node->val);
            } else {
                levelNodes.push_front(node->val);
            }
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
```

```
    }
    result.push_back(vector<int>(levelNodes.begin(), levelNodes.end()));
    leftToRight = !leftToRight;
  }
 return result;
 }
 };
```

## Output:



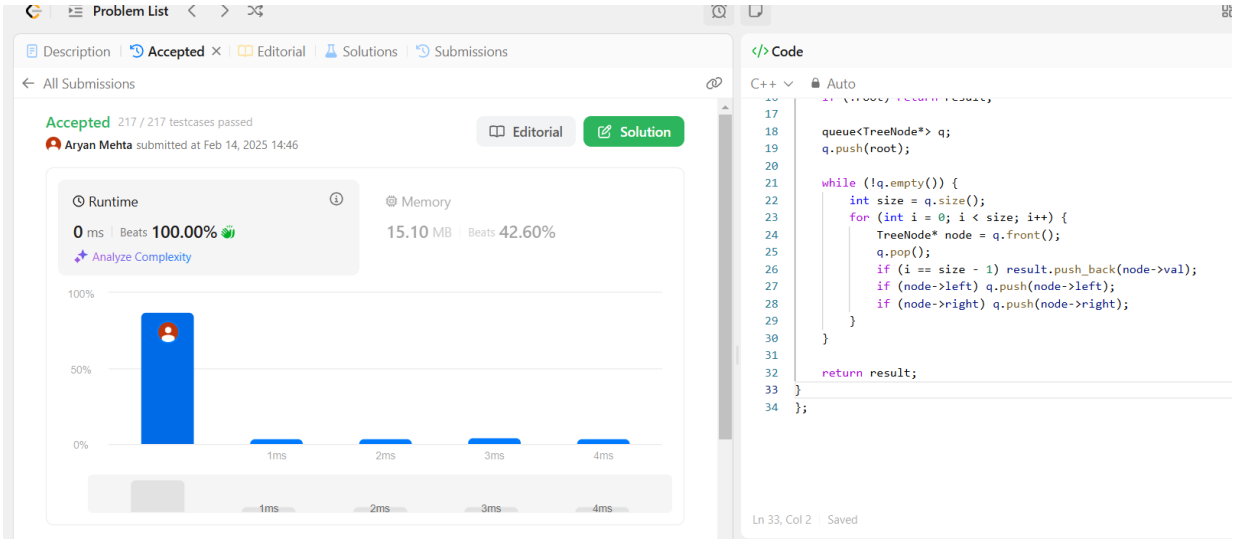

## 8. Aim: 199.Binary Tree Right Side View

## Implementation/ Code:

```
class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
    vector<int> result;
    if (!root) return result;
    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        int size = q.size();
        for (int i = 0; i < size; i++) {
            TreeNode* node = q.front();
            q.pop();
            if (i == size - 1) result.push_back(node->val);
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
```

```
    }
    return result;
}
};
```

## Output:



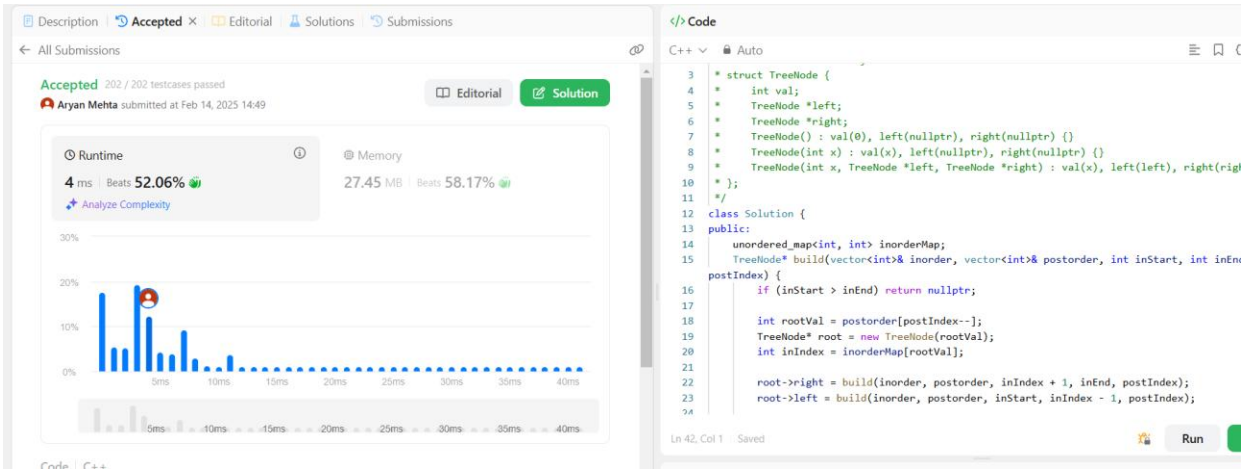

**9. Aim:** 106.Construct Binary Tree from Inorder and Postorder Traversal

## Implementation/ Code:

```
class Solution {
public:
    unordered_map<int, int> inorderMap;
    TreeNode* build(vector<int>& inorder, vector<int>& postorder, int inStart, int inEnd, int&
postIndex) {
        if (inStart > inEnd) return nullptr;

        int rootVal = postorder[postIndex--];
        TreeNode* root = new TreeNode(rootVal);
        int inIndex = inorderMap[rootVal];

        root->right = build(inorder, postorder, inIndex + 1, inEnd, postIndex);
        root->left = build(inorder, postorder, inStart, inIndex - 1, postIndex);

        return root;
    }

    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        for (int i = 0; i < inorder.size(); i++) {
            inorderMap[inorder[i]] = i;
```

```
        }
        int postIndex = postorder.size() - 1;
        return build(inorder, postorder, 0, inorder.size() - 1, postIndex);
    }
};
```

**Output:**

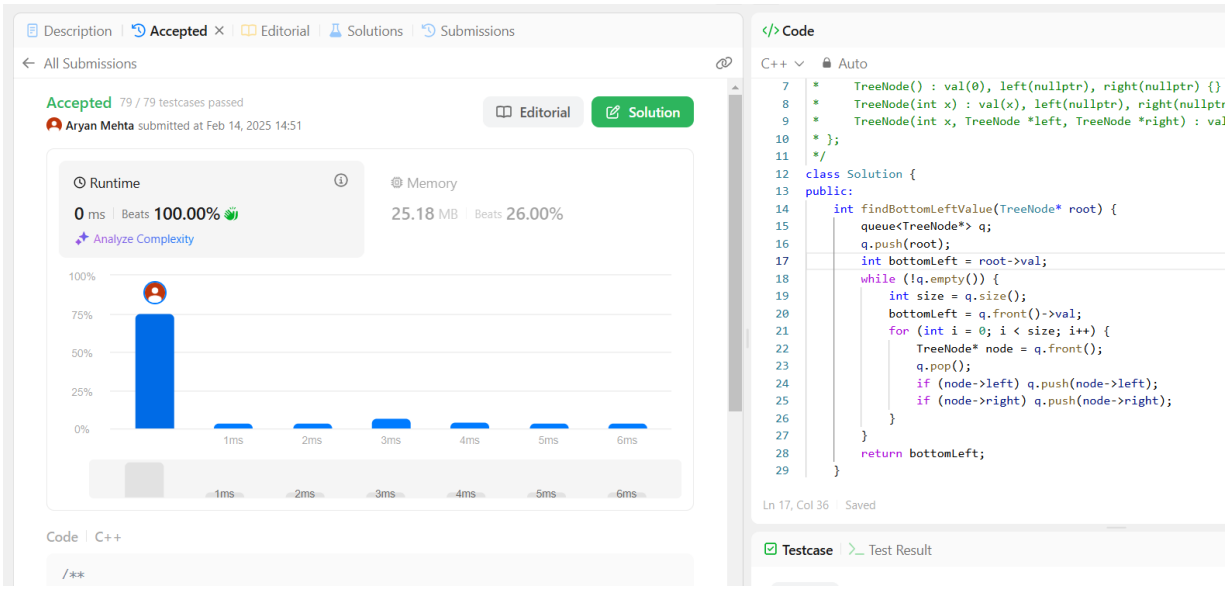



**10. Aim:** 513.Find Bottom Left Tree Value

**Implementation/ Code:**

```
class Solution {
public:
    int findBottomLeftValue(TreeNode* root) {
        queue<TreeNode*> q;
        q.push(root);
        int bottomLeft = root->val;
        while (!q.empty()) {
            int size = q.size();
            bottomLeft = q.front()->val;
            for (int i = 0; i < size; i++) {
                TreeNode* node = q.front();
                q.pop();
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
        }
        return bottomLeft;
    }
};
```

## Output:

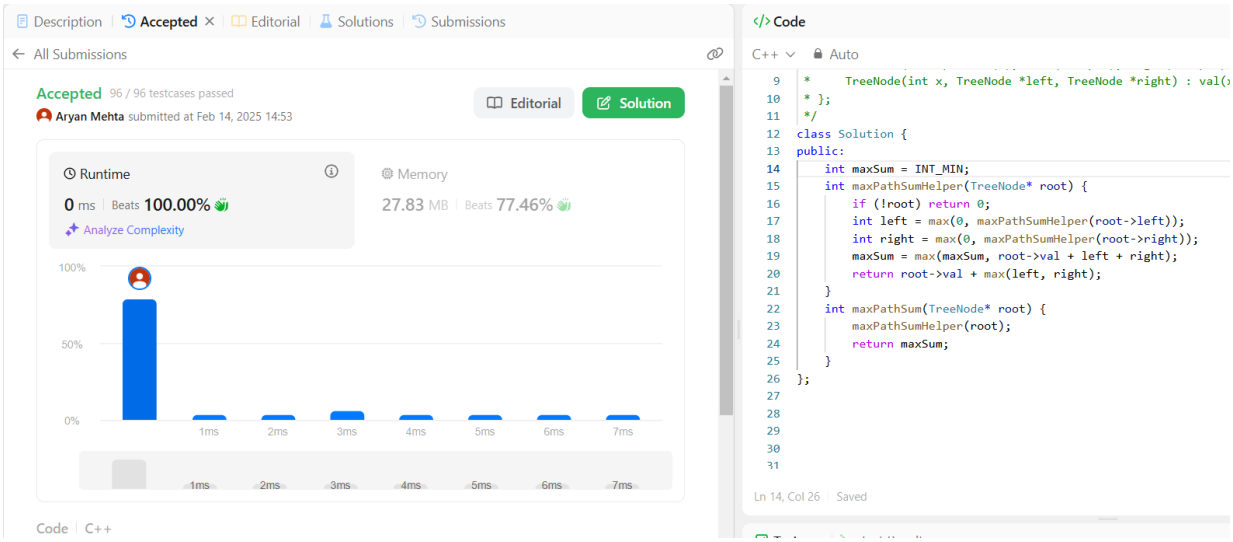



**11. Aim:** 124. Binary Tree Maximum Path Sum

## Implementation/ Code:

```
class Solution {
public:
    int maxSum = INT_MIN;
    int maxPathSumHelper(TreeNode* root) {
        if (!root) return 0;
        int left = max(0, maxPathSumHelper(root->left));
        int right = max(0, maxPathSumHelper(root->right));
        maxSum = max(maxSum, root->val + left + right);
        return root->val + max(left, right);
    }
    int maxPathSum(TreeNode* root) {
        maxPathSumHelper(root);
        return maxSum;
    }
     };
```

## Output:




**12. Aim:** 75. Sort Colors

## Implementation/ Code:

```
class Solution {
public:
    vector<vector<int>> verticalTraversal(TreeNode* root) {
        map<int, map<int, multiset<int>>> nodes;
        queue<tuple<TreeNode*, int, int>> q;
        q.push({root, 0, 0});

        while (!q.empty()) {
            auto [node, x, y] = q.front();
            q.pop();
            nodes[x][y].insert(node->val);
            if (node->left) q.push({node->left, x - 1, y + 1});
            if (node->right) q.push({node->right, x + 1, y + 1});
        }

        vector<vector<int>> result;
        for (auto& [x, yMap] : nodes) {
            vector<int> col;
            for (auto& [y, values] : yMap) {
                col.insert(col.end(), values.begin(), values.end());
            }
            result.push_back(col);
        }
```

```
        return result;
    }
};
```

**Output:**