Name: Kashish Chhimwal

UID: 22BCS14983

SECTION: 605(B)

ASSIGNMENT 3

Question1: Binary Tree Inorder Traversal :

CODE:

class Solution {

public:

    void inorder(TreeNode\* root, vector<int>& result) {

        if (!root) return;

        inorder(root->left, result);  // Left

        result.push\_back(root->val);  // Root

        inorder(root->right, result); // Right

    }

    vector<int> inorderTraversal(TreeNode\* root) {

        vector<int> result;

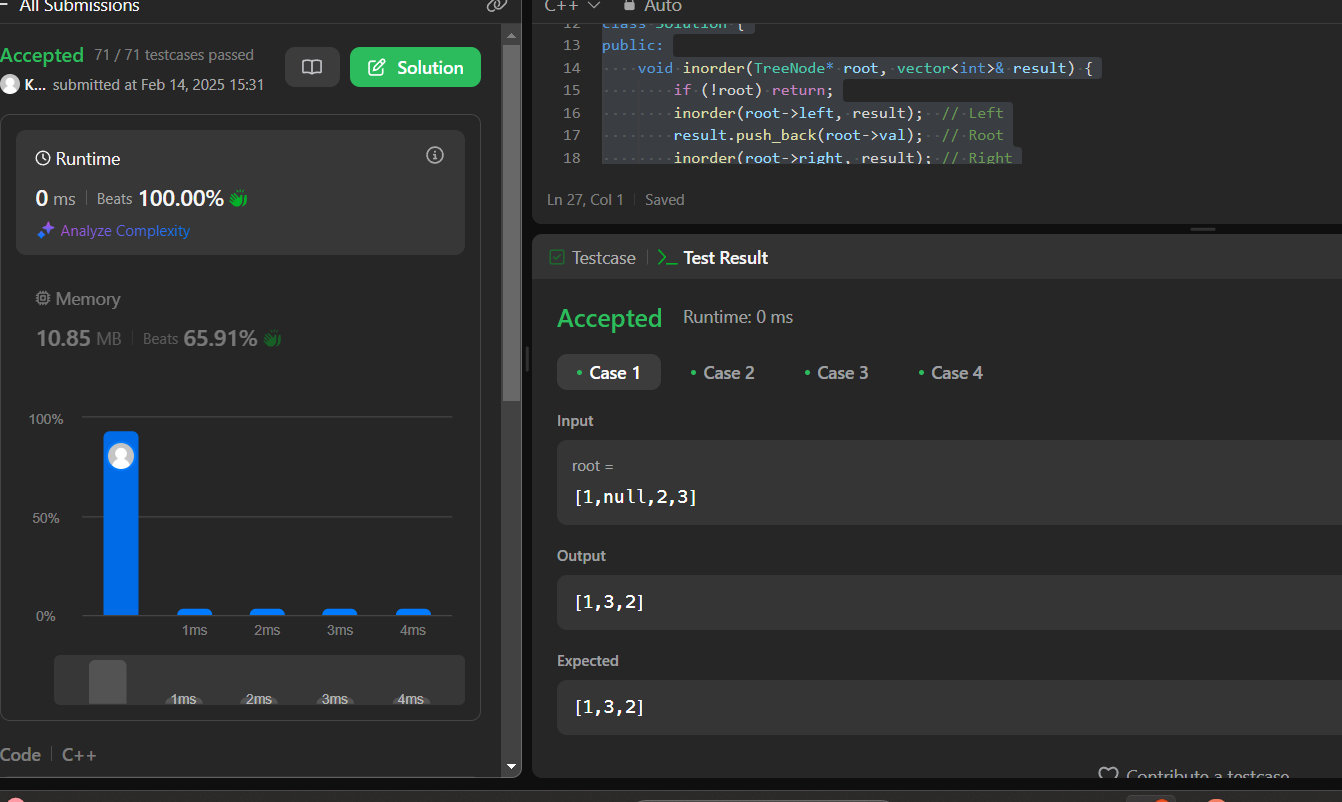
        inorder(root, result);

        return result;

    }

};

OUTPUT:



Question 2: Symmetric Tree :

CODE:

class Solution {

public:

    bool isMirror(TreeNode\* t1, TreeNode\* t2) {

        if (!t1 && !t2) return true; // Both are NULL

        if (!t1 || !t2) return false; // Only one is NULL

        return (t1->val == t2->val) &&

               isMirror(t1->left, t2->right) &&

               isMirror(t1->right, t2->left);

    }

    bool isSymmetric(TreeNode\* root) {

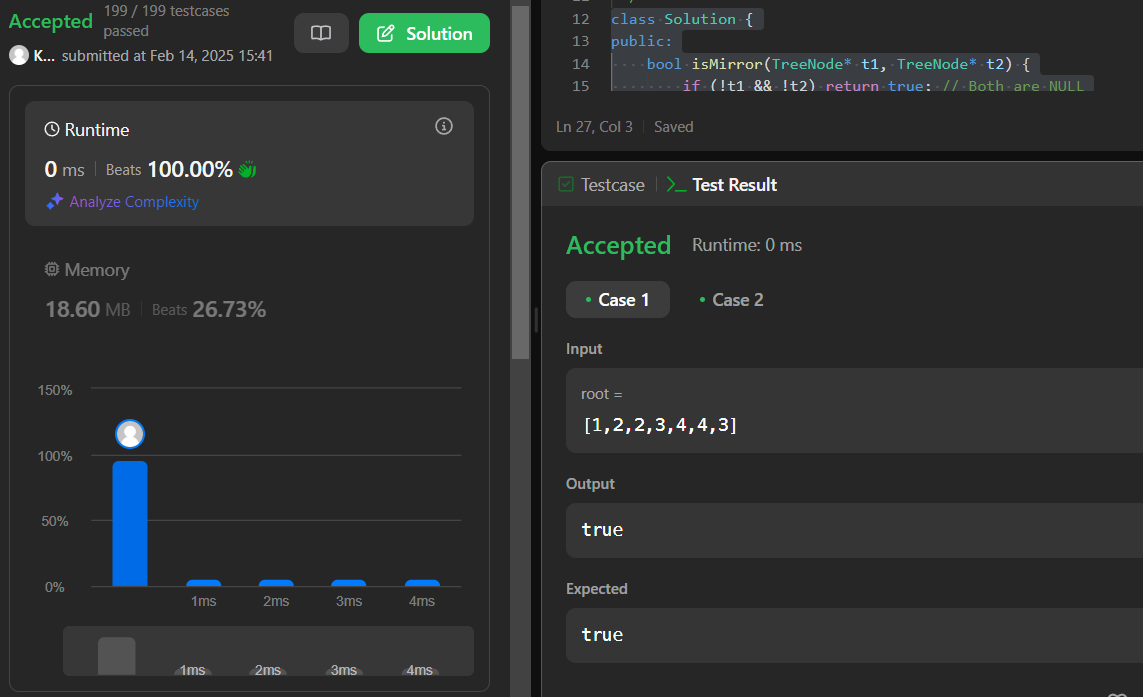
        if (!root) return true; // An empty tree is symmetric

        return isMirror(root->left, root->right);

    }

};

OUTPUT:



Question 3: Maximum Depth Of Binary Tree:

CODE:

class Solution {

public:

    int maxDepth(TreeNode\* root) {

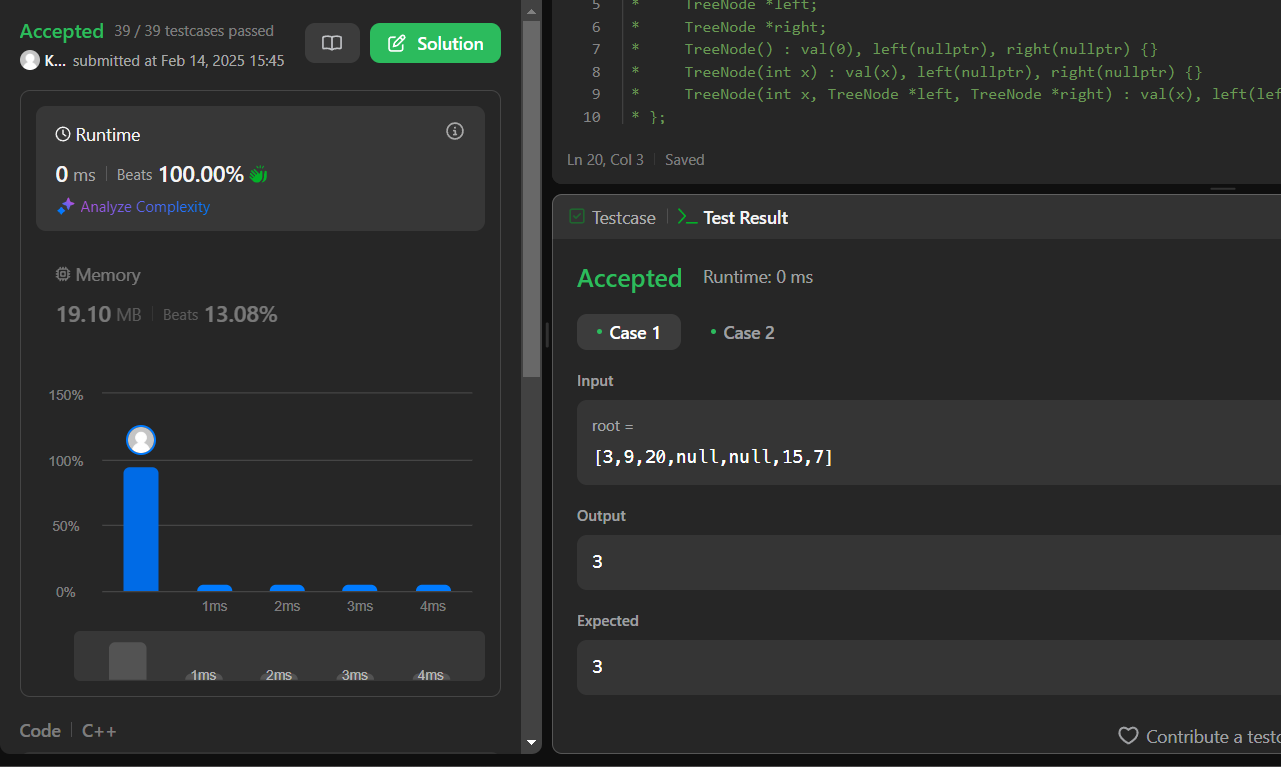
        if (!root) return 0;

        return 1 + max(maxDepth(root->left), maxDepth(root->right));

    }

};

OUTPUT:



Question 4: Validate Binary Search Tree:

CODE:

class Solution {

public:

    bool isValidBST(TreeNode\* root, long minVal = LONG\_MIN, long maxVal = LONG\_MAX) {

        if (!root) return true; // Empty tree is a BST

        if (root->val <= minVal || root->val >= maxVal) return false; // Out of valid range

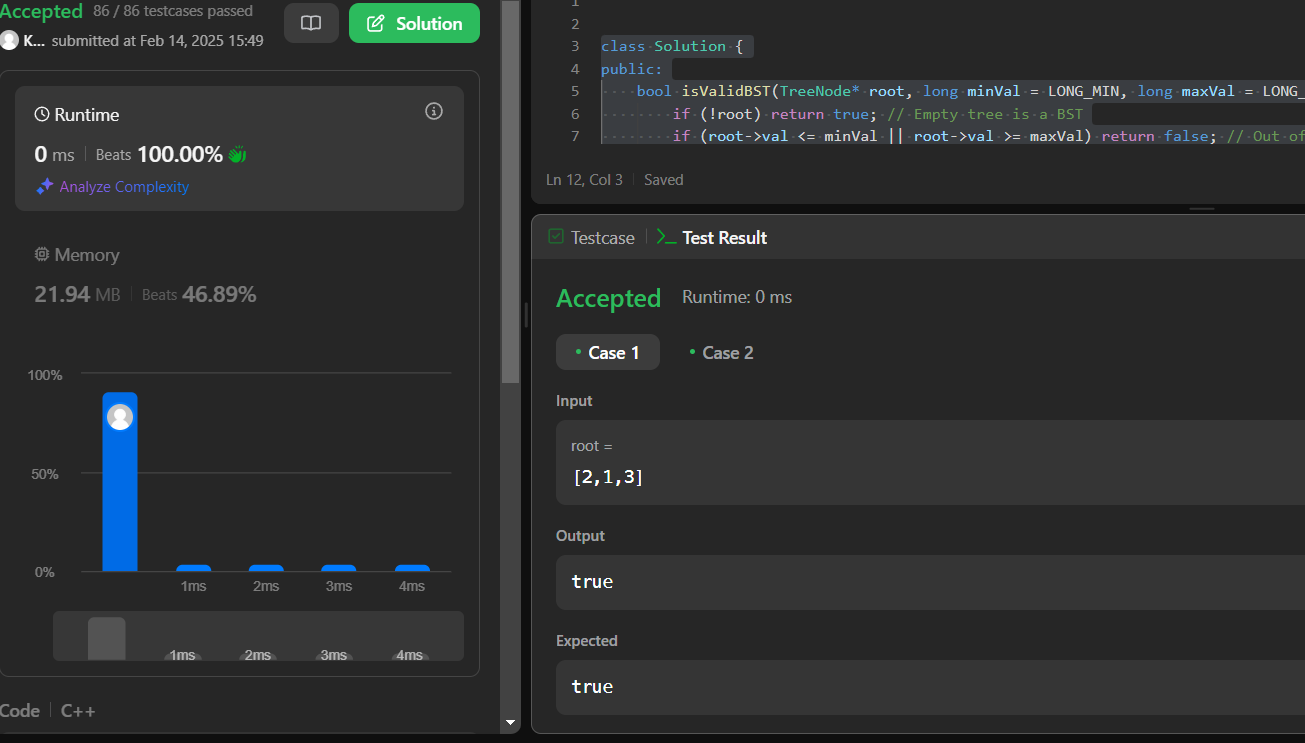
        return isValidBST(root->left, minVal, root->val) &&

               isValidBST(root->right, root->val, maxVal);

    }

};

OUTPUT:



Question 5: kth Smallest element in a BST:

CODE:

class Solution {

public:

    int kthSmallest(TreeNode\* root, int k) {

        stack<TreeNode\*> st;

        TreeNode\* curr = root;

        while (!st.empty() || curr) {

            while (curr) {

                st.push(curr);

                curr = curr->left; // Move left

            }

            curr = st.top();

            st.pop();

            k--; // Process node

            if (k == 0) return curr->val; // kth smallest found

            curr = curr->right; // Move right

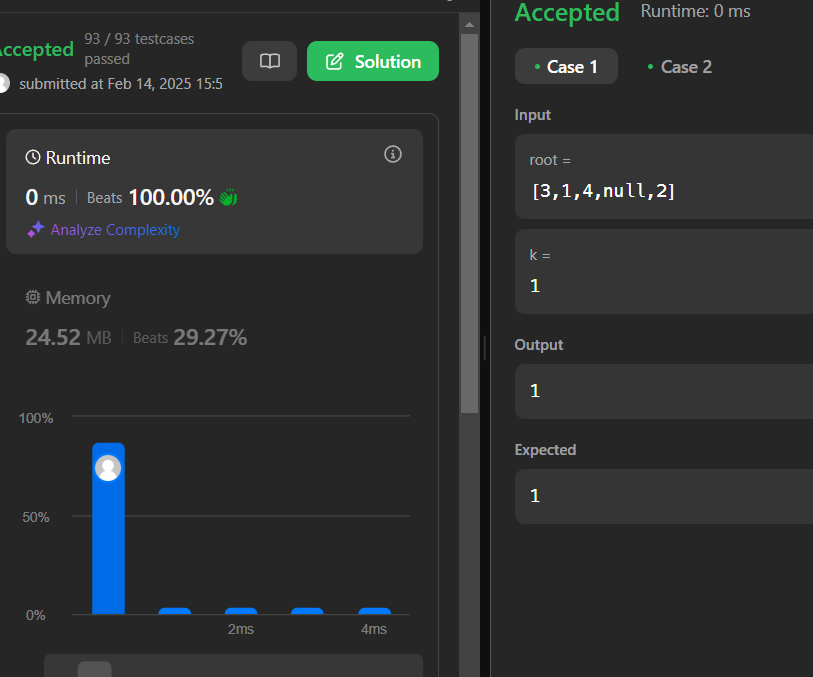
        }

        return -1; // Should never reach here

    }

};

OUTPUT:



Question 6: Binary Tree level order Traversal:

CODE:

class Solution {

public:

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

        vector<vector<int>> result;

        if (!root) return result; // If tree is empty

queue<TreeNode\*> q;

        q.push(root);

while (!q.empty()) {

            int size = q.size();

            vector<int> level;

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

                TreeNode\* node = q.front();

                q.pop();

                level.push\_back(node->val);

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

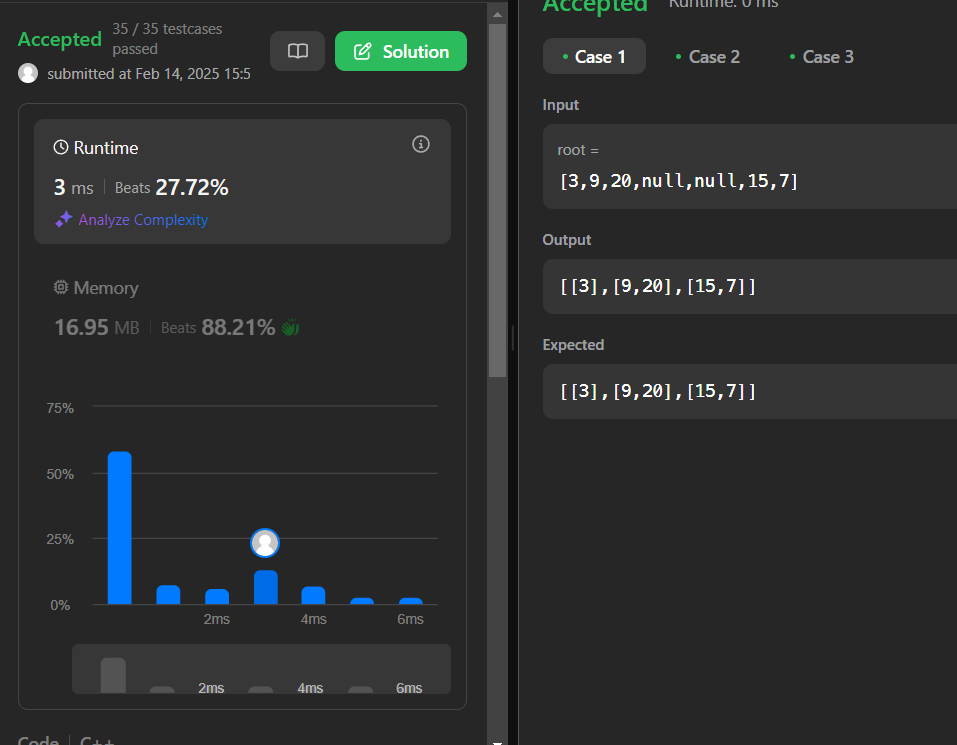
                if (node->right) q.push(node->right);}

           result.push\_back(level);}

return result;}

};

OUTPUT:



Question 7: Binary Tree level order Traversal II :

CODE:

class Solution {

public:

    vector<vector<int>> levelOrderBottom(TreeNode\* root) {

        vector<vector<int>> result;

        if (!root) return result; // If tree is empty

        queue<TreeNode\*> q;

        q.push(root);

        while (!q.empty()) {

            int size = q.size();

            vector<int> level;

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

                TreeNode\* node = q.front();

                q.pop();

                level.push\_back(node->val);

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

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

            }

            // Insert the level at the beginning instead of pushing at the end

            result.insert(result.begin(), level);

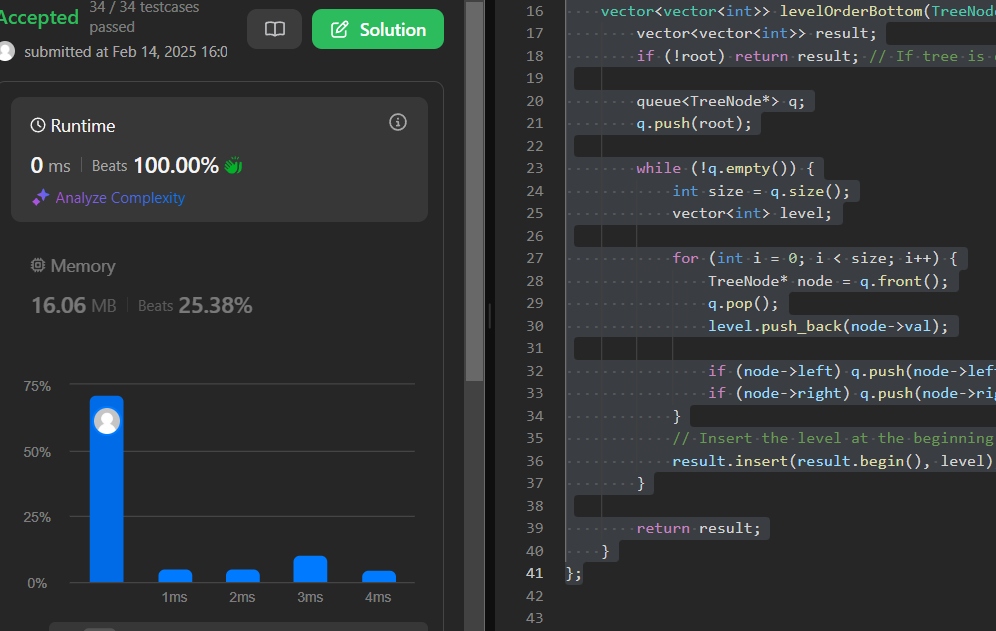
        }

        return result;

    }

};

OUTPUT:



Question 8: Binary Tree Zigzag Inorder Traversal:

CODE:

class Solution {

public:

    vector<vector<int>> zigzagLevelOrder(TreeNode\* root) {

        vector<vector<int>> result;

        if (!root) return result; // If tree is empty

        queue<TreeNode\*> q;

        q.push(root);

        bool leftToRight = true;

        while (!q.empty()) {

            int size = q.size();

            deque<int> level;  // Use deque to reverse order when needed

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

                TreeNode\* node = q.front();

                q.pop();

                if (leftToRight)

                    level.push\_back(node->val); // Normal order

                else

                    level.push\_front(node->val); // Reverse order

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

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

            }

            result.push\_back(vector<int>(level.begin(), level.end()));

            leftToRight = !leftToRight; // Toggle direction

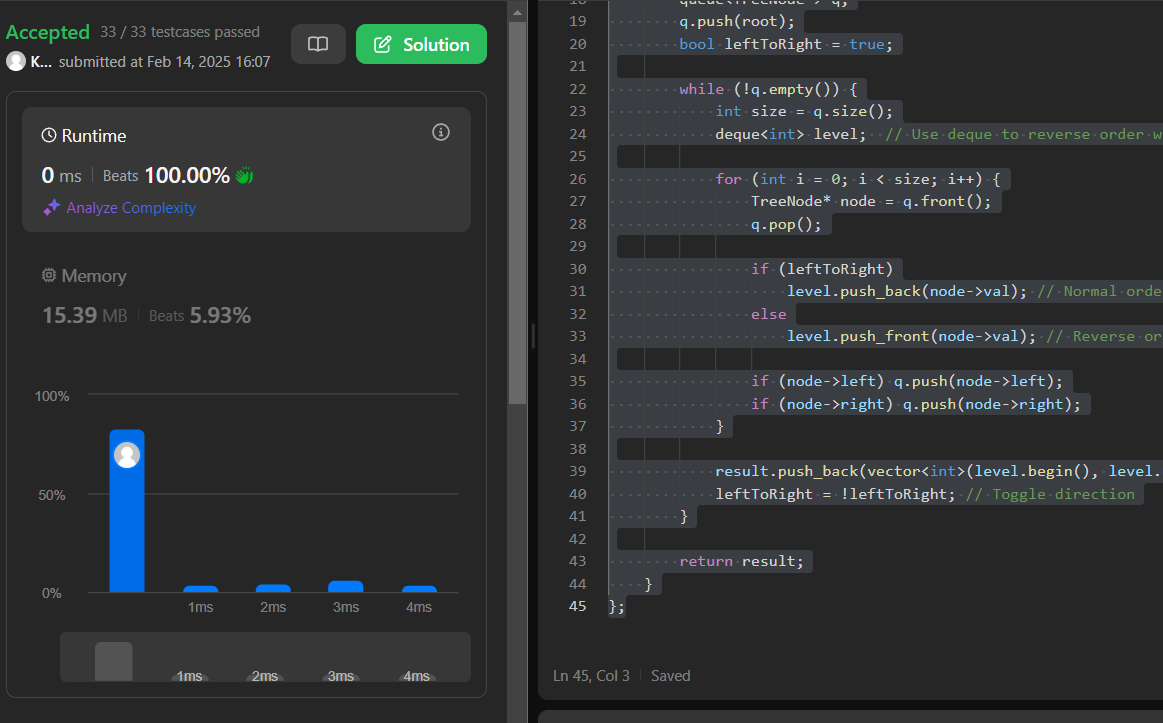
        }

        return result;

    }

};

OUTPUT:



Question 9: Binary Tree Right Side View

CODE:

class Solution {

public:

    vector<int> rightSideView(TreeNode\* root) {

        vector<int> result;

        if (!root) return result; // If tree is empty

        queue<TreeNode\*> q;

        q.push(root);

        while (!q.empty()) {

            int size = q.size();

            int rightmostValue = 0;

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

                TreeNode\* node = q.front();

                q.pop();

                rightmostValue = node->val;

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

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

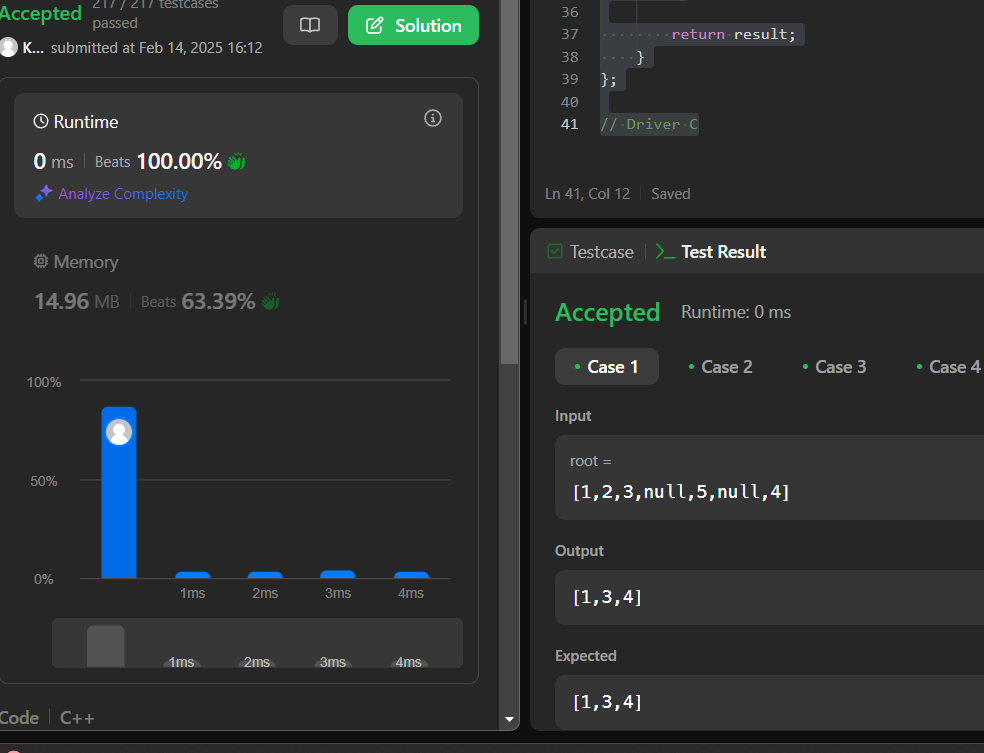
result.push\_back(rightmostValue);}

        return result;

    }

};

OUTPUT:



Question 10: Construct Binary Tree from inorder and postorder Traversal:

CODE:

class Solution {

public:

    unordered\_map<int, int> inorderIndex; // Map for quick index lookup

    int postIndex; // Index tracker for postorder array

    TreeNode\* buildTreeHelper(vector<int>& inorder, vector<int>& postorder, int left, int right) {

        if (left > right) return nullptr; // Base case

        // Pick postorder[postIndex] as root and decrease postIndex

        int rootVal = postorder[postIndex--];

        TreeNode\* root = new TreeNode(rootVal);

        // Find the root in inorder

        int index = inorderIndex[rootVal];

        // Build right and left subtrees (right first because postorder processes left last)

        root->right = buildTreeHelper(inorder, postorder, index + 1, right);

        root->left = buildTreeHelper(inorder, postorder, left, index - 1);

        return root;

    }

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

        postIndex = postorder.size() - 1; // Start from the last element in postorder

        // Store inorder indices in a hashmap for O(1) lookup

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

            inorderIndex[inorder[i]] = i;

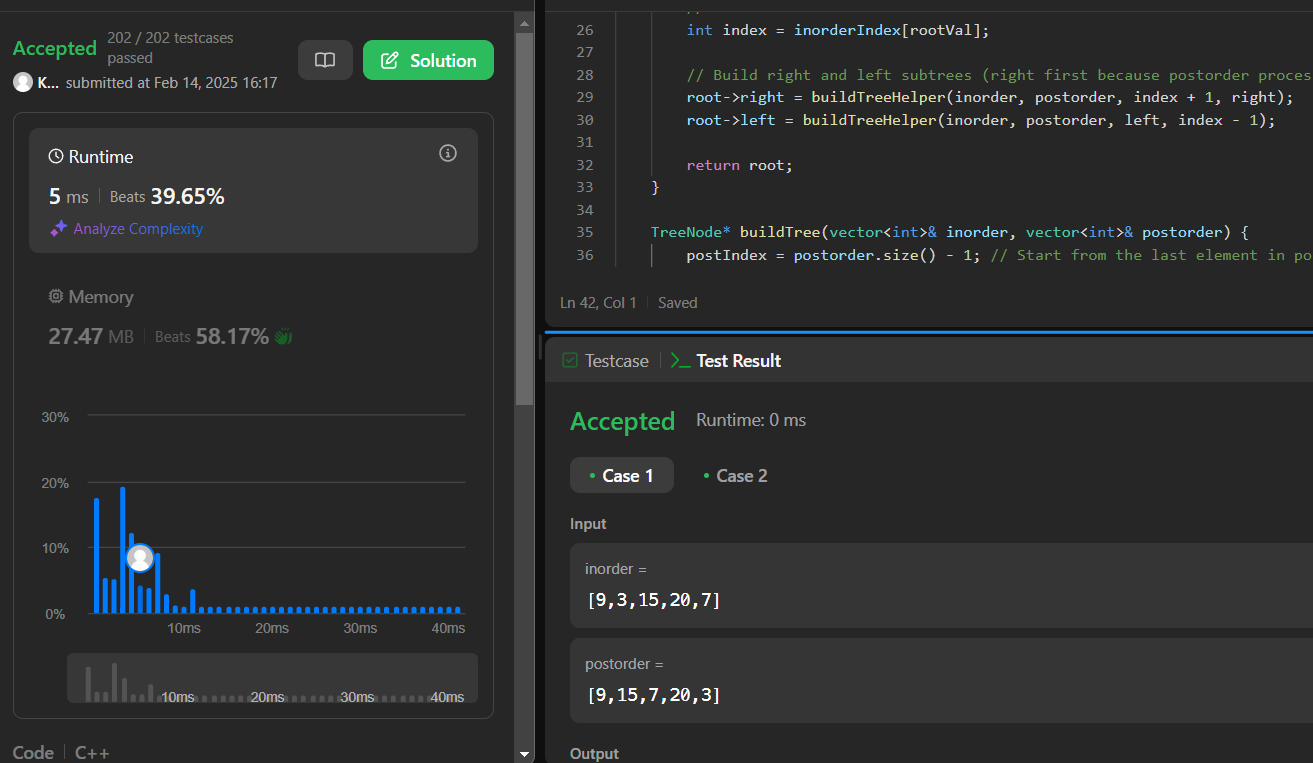
        }

        return buildTreeHelper(inorder, postorder, 0, inorder.size() - 1);

    }

};

OUTPUT:



Question 11:Find Bottom left Tree value

CODE:

class Solution {

public:

    int findBottomLeftValue(TreeNode\* root) {

        queue<TreeNode\*> q;

        q.push(root);

        int leftmost = 0;

        while (!q.empty()) {

            int size = q.size();

            leftmost = q.front()->val; // First node in this level

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

                TreeNode\* node = q.front();

                q.pop();

                // Push left child first (so leftmost is processed first)

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

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

            }

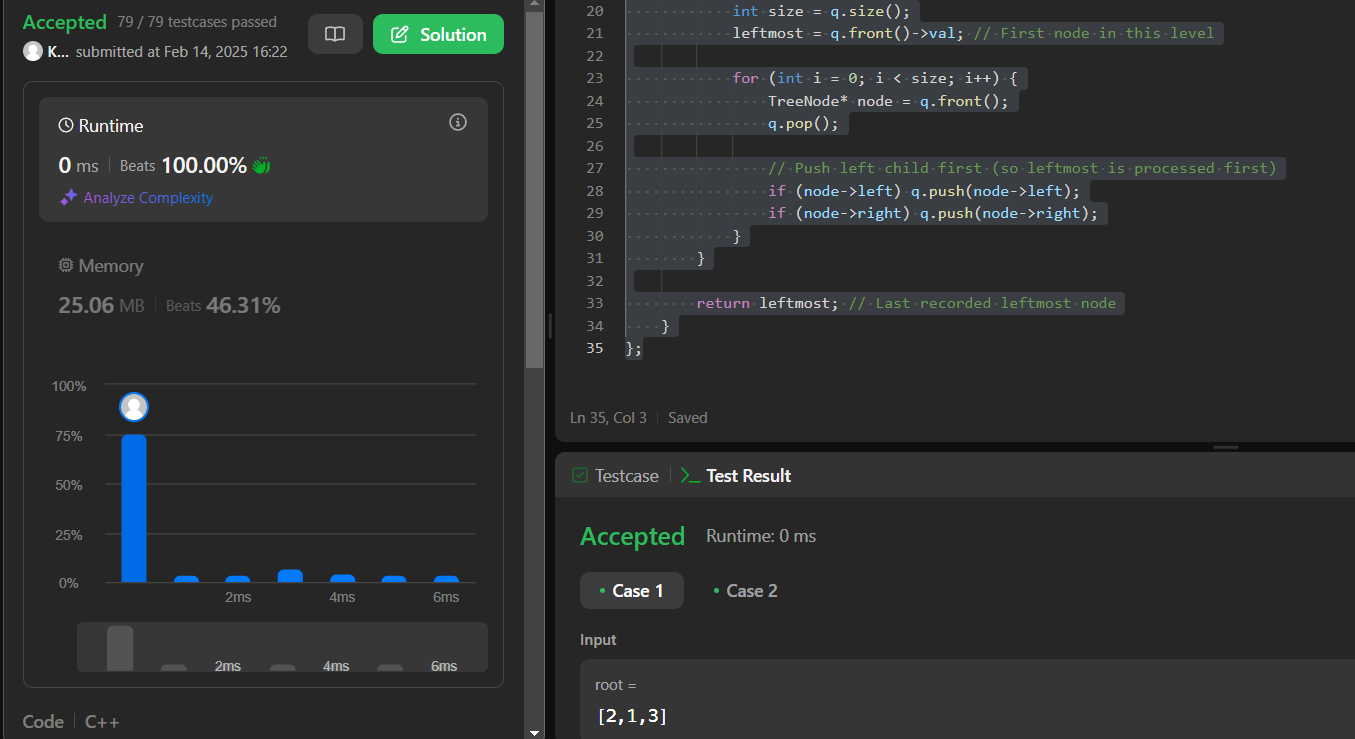
        }

        return leftmost; // Last recorded leftmost node

    }

};

OUTPUT:



Question 12: Binary Tree Maximum Path Sum:

CODE:

class Solution {

public:

    int maxSum = INT\_MIN; // Global variable to store the max path sum

    int maxPathSumHelper(TreeNode\* root) {

        if (!root) return 0; // Base case

        // Recursively get the max path sum of left and right subtrees

        int leftMax = max(0, maxPathSumHelper(root->left));

        int rightMax = max(0, maxPathSumHelper(root->right));

        // Update the global max sum considering the current node as the highest root

        maxSum = max(maxSum, leftMax + rightMax + root->val);

        // Return the maximum sum path that can be extended to the parent

        return root->val + max(leftMax, rightMax);

    }

    int maxPathSum(TreeNode\* root) {

        maxPathSumHelper(root);

        return maxSum;

    }

};

OUTPUT:

class Solution {

public:

    int maxSum = INT\_MIN; // Global variable to store the max path sum

    int maxPathSumHelper(TreeNode\* root) {

        if (!root) return 0; // Base case

        // Recursively get the max path sum of left and right subtrees

        int leftMax = max(0, maxPathSumHelper(root->left));

        int rightMax = max(0, maxPathSumHelper(root->right));

        // Update the global max sum considering the current node as the highest root

        maxSum = max(maxSum, leftMax + rightMax + root->val);

        // Return the maximum sum path that can be extended to the parent

        return root->val + max(leftMax, rightMax);

    }

    int maxPathSum(TreeNode\* root) {

        maxPathSumHelper(root);

        return maxSum;

    }

};

Question 13: Vertical Order Traversal of a Binary Tree:

CODE:

class Solution {

public:

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

        map<int, vector<pair<int, int>>> columnMap; // {col -> [(row, value)]}

        queue<pair<TreeNode\*, pair<int, int>>> q;  // {node, (row, col)}

        q.push({root, {0, 0}});

        while (!q.empty()) {

            auto [node, pos] = q.front(); q.pop();

            int row = pos.first, col = pos.second;

            columnMap[col].push\_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] : columnMap) {

            sort(nodes.begin(), nodes.end()); // Sort by (row, value)

            vector<int> sortedVals;

            for (auto& [row, value] : nodes) {

                sortedVals.push\_back(value);

            }

            result.push\_back(sortedVals);

        }

        return result;

    }

};

OUTPUT:

