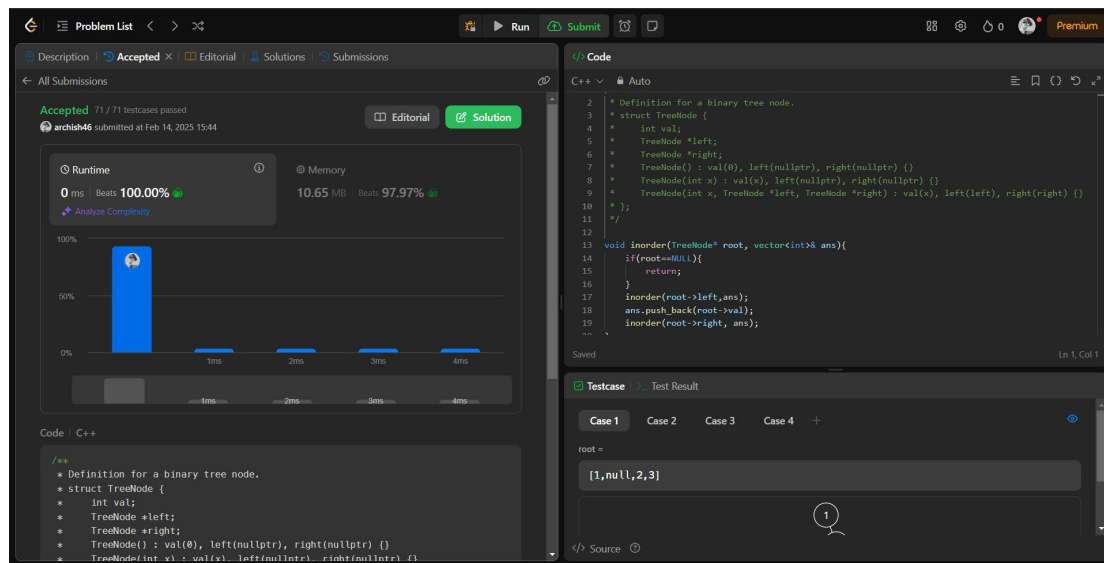


NAME: Archishman Mukherjee uid: 22BCS11671 Sec- 605(B)

1. Binary Tree Inorder Traversal



Runtime 0 ms | Beats 100.00% | **Memory** 10.65 MB | Beats 97.97%

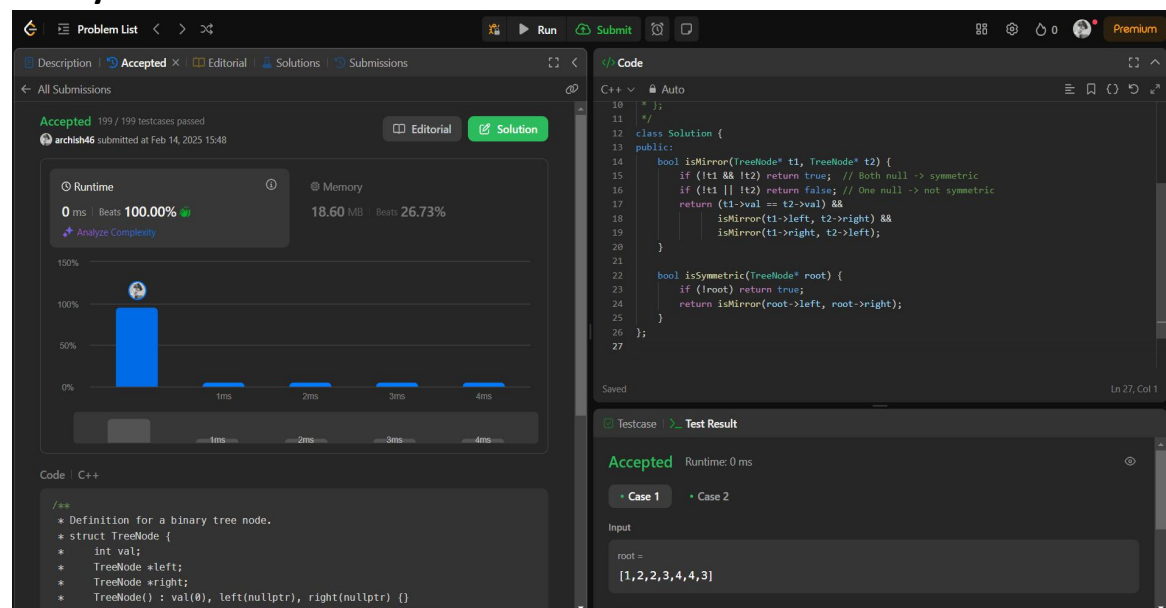
```
2 * Definition for a binary tree node.
3 * struct TreeNode {
4 *     int val;
5 *     TreeNode *left;
6 *     TreeNode *right;
7 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
9 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
10 * };
11
12
13 void inorder(TreeNode* root, vector<int>& ans){
14     if(root==NULL){
15         return;
16     }
17     inorder(root->left,ans);
18     ans.push_back(root->val);
19     inorder(root->right, ans);
20 }
```

Testcase: Case 1 Case 2 Case 3 Case 4 +

root = [1,null,2,3]

Output: 1

2. Symmetric Tree



Runtime 0 ms | Beats 100.00% | **Memory** 18.60 MB | Beats 26.73%

```
10 * };
11 */
12 class Solution {
13 public:
14     bool isMirror(TreeNode* t1, TreeNode* t2) {
15         if (!t1 && !t2) return true; // Both null -> symmetric
16         if (!t1 || !t2) return false; // One null -> not symmetric
17         return (t1->val == t2->val) &&
18             isMirror(t1->left, t2->right) &&
19             isMirror(t1->right, t2->left);
20     }
21
22     bool isSymmetric(TreeNode* root) {
23         if (!root) return true;
24         return isMirror(root->left, root->right);
25     }
26 };
27
```

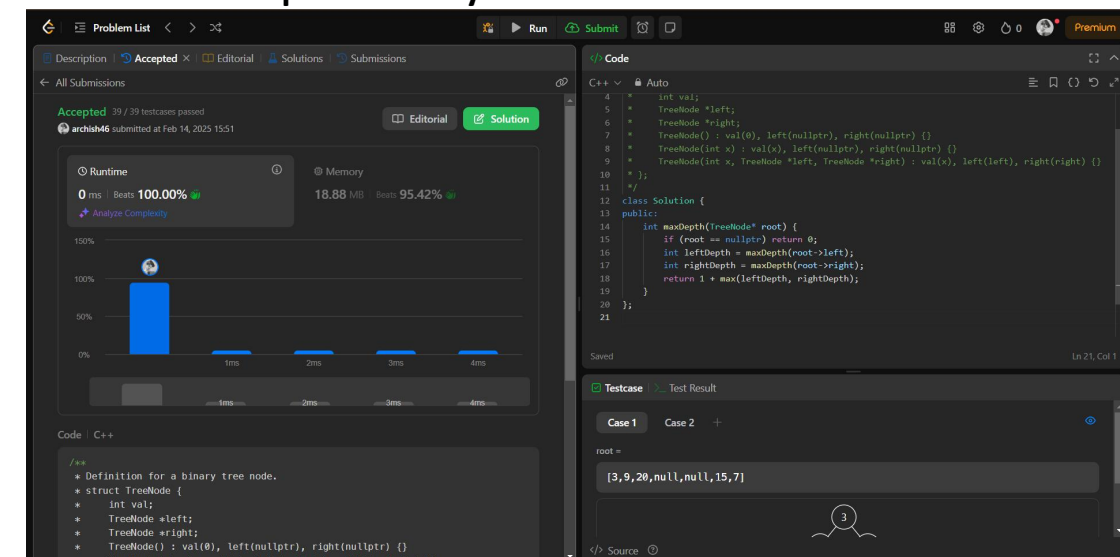
Testcase: Accepted Runtime: 0 ms

Case 1 Case 2

Input: root = [1,2,2,3,4,4,3]

Output:

3. Maximum Depth of binary tree



Runtime 0 ms | Beats 100.00% | **Memory** 18.88 MB | Beats 95.42%

```
4 * int val;
5 *     TreeNode *left;
6 *     TreeNode *right;
7 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
8 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
9 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
10 * };
11
12
13 class Solution {
14 public:
15     int maxDepth(TreeNode* root) {
16         if (root == nullptr) return 0;
17         int leftDepth = maxDepth(root->left);
18         int rightDepth = maxDepth(root->right);
19         return 1 + max(leftDepth, rightDepth);
20     }
21 }
```

Testcase: Case 1 Case 2 +

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

Output: 3

4. Validate binary search tree

The screenshot shows a C++ solution for the 'Validate binary search tree' problem. The code defines a `TreeNode` struct and a `isValidBST` function. The function uses a recursive approach to check if the tree is a valid BST by comparing node values with a range of `minVal` and `maxVal`. The test result shows the solution is 'Accepted' with a runtime of 0 ms and memory usage of 21.91 MB. The input for the test case is `root = [2,1,3]`.

```
11 //
12 #include <limits.h>
13
14 class Solution {
15 public:
16     bool isValidBST(TreeNode* root, long minVal, long maxVal) {
17         if (!root) return true; // Base case: empty tree is a valid BST
18         if (root->val <= minVal || root->val >= maxVal) return false; // Violation of BST
19         return isValidBST(root->left, minVal, root->val) && isValidBST(root->right, root->val, maxVal);
20     }
21
22     bool isValidBST(TreeNode* root) {
23         return isValidBST(root, LONG_MIN, LONG_MAX);
24     }
25 };
26
```

Testcase: Accepted Runtime: 0 ms

Case 1 Case 2

Input

root = [2,1,3]

Output

5. Kth Smallest Element in a BST

The screenshot shows a C++ solution for the 'Kth Smallest Element in a BST' problem. The code defines a `TreeNode` struct and a `kthSmallest` function. The function uses an in-order traversal to find the kth smallest element. The test result shows the solution is 'Accepted' with a runtime of 0 ms and memory usage of 24.40 MB. The input for the test case is `root = [3,1,4,null,2]`.

```
16 void inorder(TreeNode* root, int k) {
17     if (!root || count >= k) return;
18     inorder(root->left, k); // Traverse left subtree
19     count++;
20     if (count == k) {
21         result = root->val;
22         return;
23     }
24     inorder(root->right, k); // Traverse right subtree
25 }
26
27
28 int kthSmallest(TreeNode* root, int k) {
29     inorder(root, k);
30     return result;
31 }
32
33
```

Testcase: Accepted Runtime: 0 ms

Case 1 Case 2

Input

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

Output

6. Binary Tree Level Order Traversal

The screenshot shows a C++ solution for the 'Binary Tree Level Order Traversal' problem. The code defines a `TreeNode` struct and a `levelOrder` function. The function uses a breadth-first search (BFS) approach to traverse the tree level by level. The test result shows the solution is 'Accepted' with a runtime of 7 ms and memory usage of 17.22 MB. The input for the test case is `root = [3,9,20,null,null,15,7]`.

```
15
16 class Solution {
17 public:
18     vector<vector<int>> levelOrder(TreeNode* root) {
19         vector<vector<int>> result;
20         if (!root) return result; // Return empty list if tree is empty
21
22         queue<TreeNode*> q;
23         q.push(root);
24
25         while (!q.empty()) {
26             int size = q.size();
27             vector<int> level;
28
29             for (int i = 0; i < size; i++) {
30                 TreeNode* node = q.front();
31                 q.pop();
32                 level.push_back(node->val);
33                 if (node->left) q.push(node->left);
34                 if (node->right) q.push(node->right);
35             }
36             result.push_back(level);
37         }
38         return result;
39     }
40 };

```

Testcase: Accepted Runtime: 7 ms

Case 1 Case 2 Case 3

Input

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

Output

7. Binary Tree Level Order Traversal II

The screenshot shows a C++ solution for the problem "Binary Tree Level Order Traversal II". The code is implemented using a queue to perform a level-order traversal from bottom to top. The runtime is 1 ms, and the memory usage is 16.09 MB. The test result is "Accepted" with a runtime of 0 ms.

```
class Solution {
public:
    vector<vector<int>> levelOrderBottom(TreeNode* root) {
        vector<vector<int>> result;
        if (!root) return result; // Return empty 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();
                level.push_back(node->val);
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
                q.pop();
            }
            result.push_back(level);
        }
        reverse(result.begin(), result.end());
        return result;
    }
};
```

8. Binary tree zigzag level order traversal

The screenshot shows a C++ solution for the problem "Binary tree zigzag level order traversal". The code uses a queue to perform a level-order traversal, alternating the direction of the result vector for each level. The runtime is 0 ms, and the memory usage is 15.09 MB. The test result is "Accepted" with a runtime of 0 ms.

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> result;
        if (!root) return result; // If tree is empty, return empty result
        queue<TreeNode*> q;
        q.push(root);
        bool leftToRight = true; // Toggle for direction
        while (!q.empty()) {
            int size = q.size();
            vector<int> level(size);
            for (int i = 0; i < size; i++) {
                TreeNode* node = q.front();
                level[i] = node->val;
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
                q.pop();
            }
            if (!leftToRight) reverse(level.begin(), level.end());
            result.push_back(level);
            leftToRight = !leftToRight;
        }
        return result;
    }
};
```

9. Binary Tree Right Side View

The screenshot shows a C++ solution for the problem "Binary Tree Right Side View". The code uses a queue to perform a level-order traversal, storing the value of the rightmost node in each level. The runtime is 0 ms, and the memory usage is 14.88 MB. The test result is "Accepted" with a runtime of 0 ms.

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

10. Construct binary tree from inorder and post order traversal

The screenshot shows a LeetCode submission for the problem "Construct binary tree from inorder and post order traversal". The submission is accepted, with a runtime of 0 ms and memory usage of 27.46 MB. The code is written in C++ and implements a recursive solution to build the binary tree from the given inorder and postorder traversals.

Runtime: 0 ms | Beats 100.00%
Memory: 27.46 MB | Beats 58.17%

Code (C++):

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 * };
 */
class Solution {
public:
    unordered_map<int, int> inorderMap; // Stores index of values in inorder traversal
    int postIndex; // Tracks the current root in postorder array

    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        postIndex = postorder.size() - 1; // Last element is the root
        for (int i = 0; i < inorder.size(); i++) {
            inorderMap[inorder[i]] = i; // Store index of each element in inorder
        }
        return buildTreeHelper(postorder, 0, inorder.size() - 1);
    }

    TreeNode* buildTreeHelper(vector<int>& postorder, int left, int right) {
        if (left > right) return nullptr; // Base case

        // ... (rest of the code) ...
    }
};
```

11. Find Bottom Left Tree Value

The screenshot shows a LeetCode submission for the problem "Find Bottom Left Tree Value". The submission is accepted, with a runtime of 0 ms and memory usage of 24.96 MB. The code is written in C++ and implements a breadth-first search (BFS) solution to find the value of the bottom-left node in a binary tree.

Runtime: 0 ms | Beats 100.00%
Memory: 24.96 MB | Beats 63.66%

Code (C++):

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 * };
 */
class Solution {
public:
    int findBottomLeftValue(TreeNode* root) {
        queue<TreeNode*> q;
        q.push(root);
        int bottomLeft = root->val;

        while (!q.empty()) {
            int levelSize = q.size();
            bottomLeft = q.front()->val; // First element of the level

            // ... (rest of the code) ...
        }
        return bottomLeft;
    }
};
```

12. Binary Tree Maximum Path Sum

The screenshot shows a LeetCode submission for the problem "Binary Tree Maximum Path Sum". The submission is accepted, with a runtime of 0 ms and memory usage of 27.84 MB. The code is written in C++ and implements a recursive solution to find the maximum path sum in a binary tree.

Runtime: 0 ms | Beats 100.00%
Memory: 27.84 MB | Beats 77.46%

Code (C++):

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 * };
 */
class Solution {
public:
    int maxSum = INT_MIN; // Global max path sum

    int maxPathHelper(TreeNode* root) {
        if (!root) return 0;

        // Compute max path sum from left and right subtrees
        int left = max(0, maxPathHelper(root->left)); // Ignore negative sums
        int right = max(0, maxPathHelper(root->right));

        // ... (rest of the code) ...
    }
};
```

13. Vertical Order Traversal of a Binary Tree

