**Name : Anjali**

**UID : 22BCS10665  
Batch : 22BCS\_IOT-614(B)**

**Assignment – 5**

**Problem-1 : Maximum Depth of Binary Tree**  
**Code:**

class Solution {

public:

    int maxDepth(TreeNode\* root) {

        if (!root) return 0;

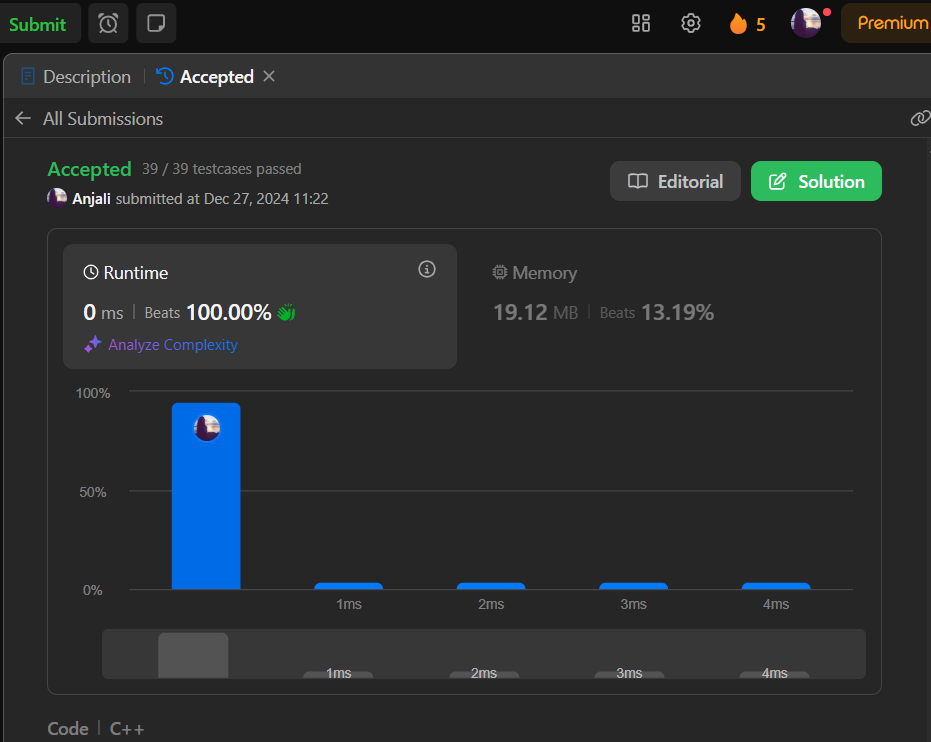
        int leftDepth = maxDepth(root->left);

        int rightDepth = maxDepth(root->right);

        return 1 + std::max(leftDepth, rightDepth);

    }

};

**Output SS:**

**Problem-2 : Validate Binary Search Tree**  
**Code:**

class Solution {

public:

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

        if (!root) return true;

        if (root->val <= minVal || root->val >= maxVal) return false;

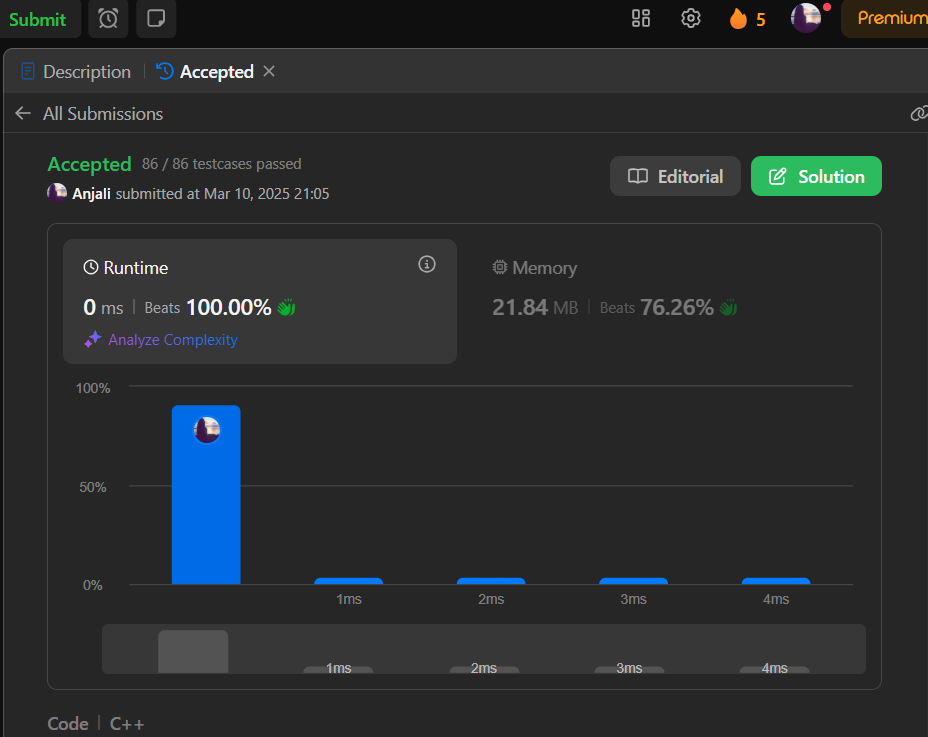
        return isValidBST(root->left, minVal, root->val) &&

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

    }

};

**Output SS:**



**Problem-3 : Symmetric Tree**  
**Code:**

class Solution {

public:

    bool isSymmetric(TreeNode\* root) {

        if (!root) return true;

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

    }

private:

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

        if (!t1 && !t2) return true;

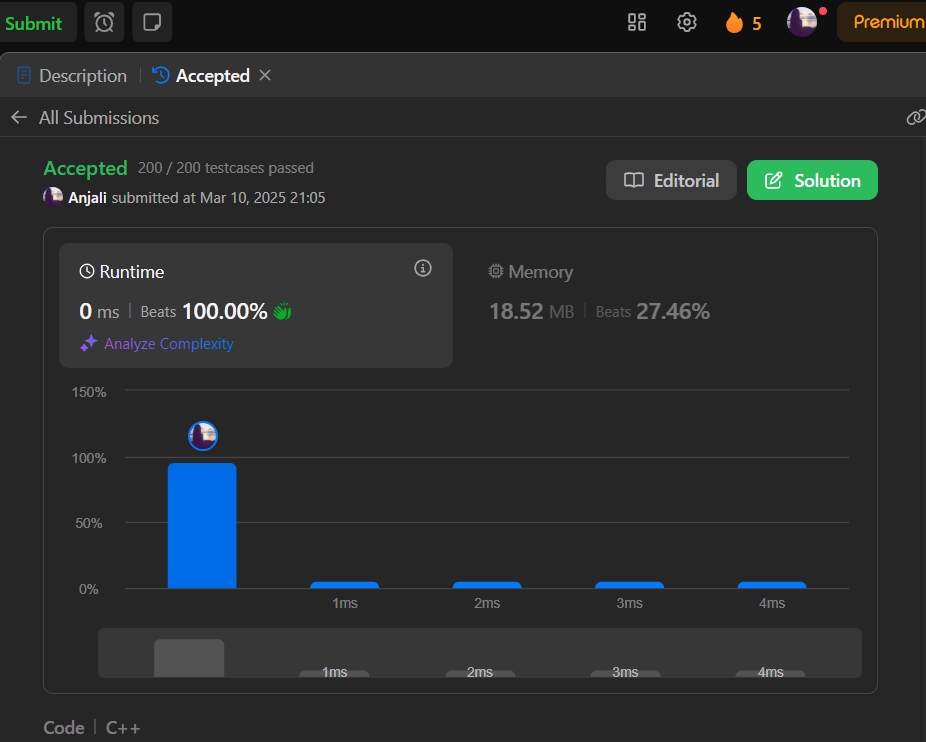
        if (!t1 || !t2 || t1->val != t2->val) return false;

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

    }

};

**Output SS:**



**Problem-4 : Binary Tree Zigzag Level Order Traversal**  
**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> levelValues;

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

                TreeNode\* node = q.front();

                q.pop();

                if (leftToRight) {

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

                } else {

                    levelValues.push\_front(node->val);

                }

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

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

            }

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

            leftToRight = !leftToRight;

        }

        return result;

    }

};

void printVector(vector<vector<int>> vec) {

    cout << "[";

    for (size\_t i = 0; i < vec.size(); i++) {

        cout << "[";

        for (size\_t j = 0; j < vec[i].size(); j++) {

            cout << vec[i][j];

            if (j != vec[i].size() - 1) cout << ",";

        }

        cout << "]";

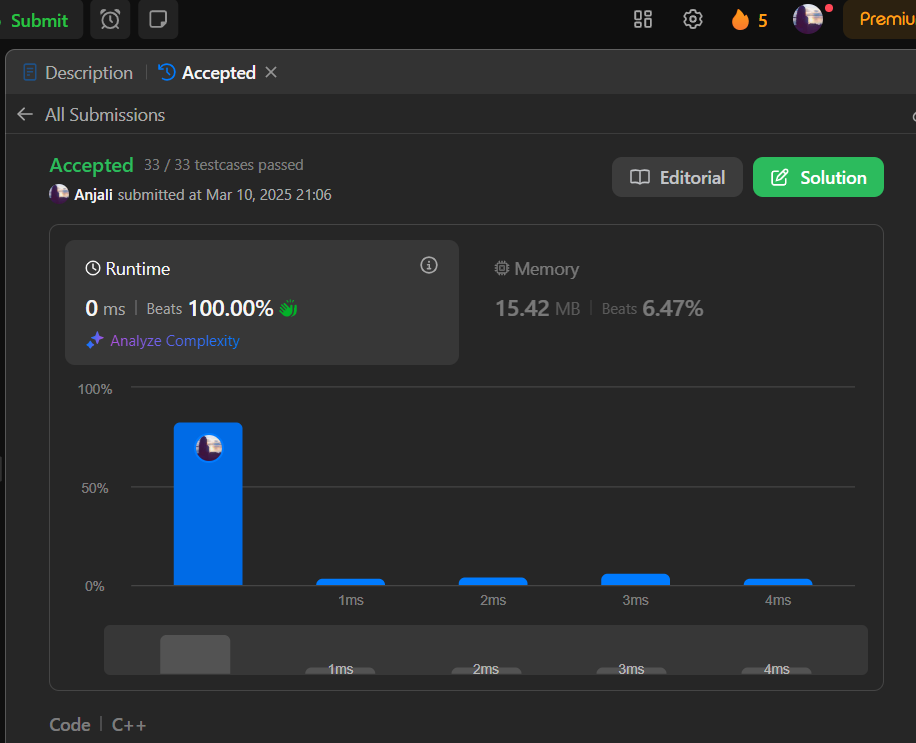
        if (i != vec.size() - 1) cout << ",";

    }

    cout << "]\n";

}

**Output SS:**



**Problem-5 : Lowest Common Ancestor of a Binary Tree**  
**Code:**

class Solution {

public:

    TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

        if (root == nullptr || root == p || root == q) {

            return root;

        }

        TreeNode\* left = lowestCommonAncestor(root->left, p, q);

        TreeNode\* right = lowestCommonAncestor(root->right, p, q);

        if (left != nullptr && right != nullptr) {

            return root;

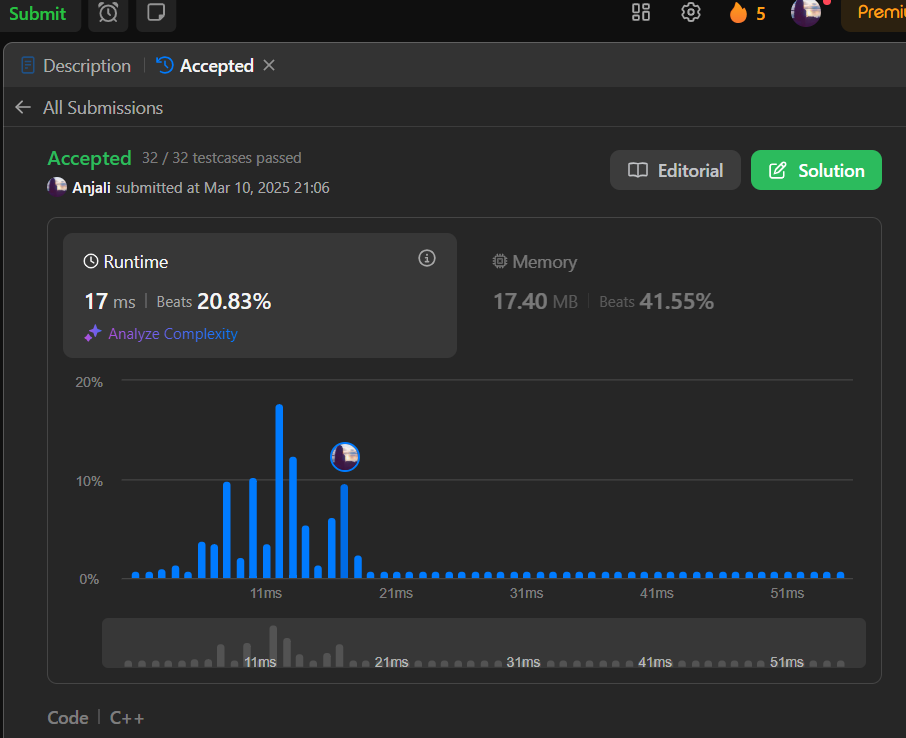
        }

        return left != nullptr ? left : right;

    }

};

**Output SS:**



**Problem-6 : Binary Tree Inorder Traversal**  
**Code:**

class Solution {

public:

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

        if (!root) return;

        inorder(root->left, result);

        result.push\_back(root->val);

        inorder(root->right, result);

    }

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

        vector<int> result;

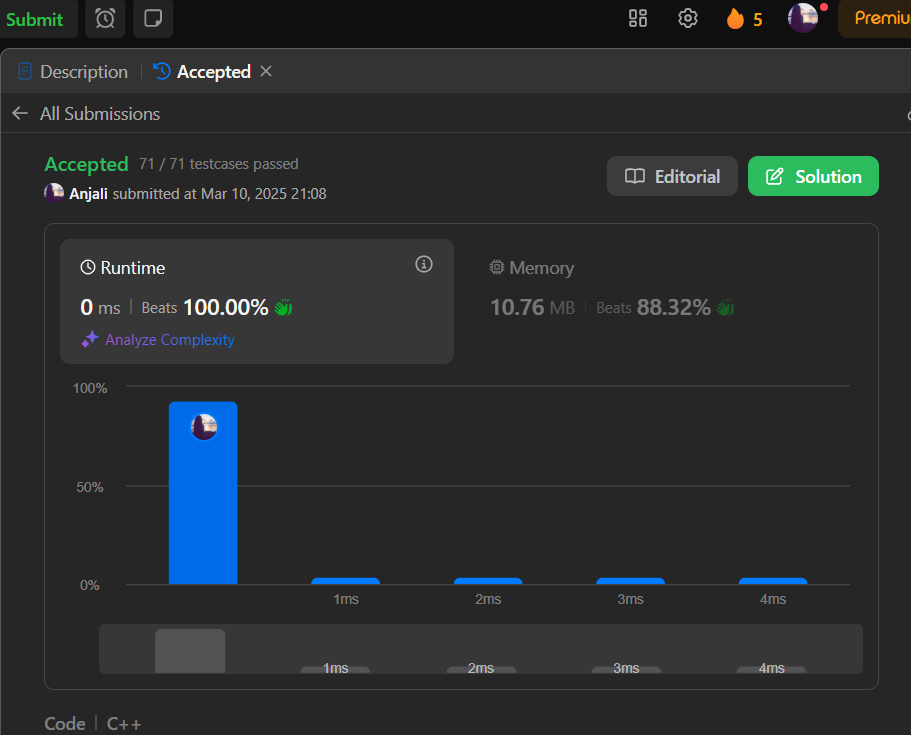
        inorder(root, result);

        return result;

    }

};

**Output SS:**



**Problem-7 : Binary Tree Level Order Traversal**  
**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 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);

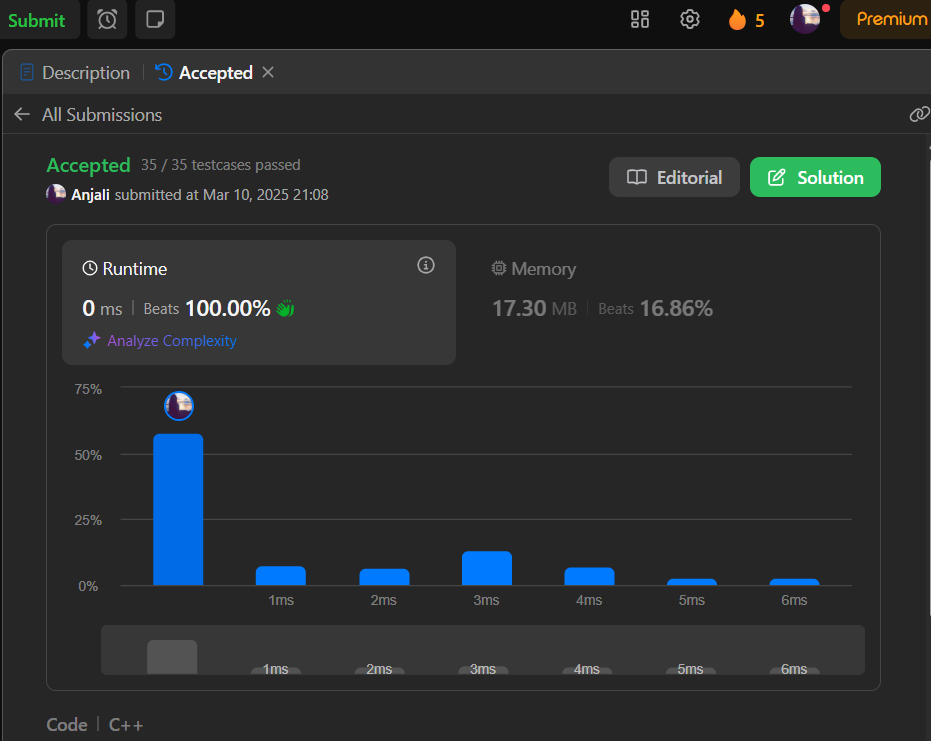
        }

        return result;

    }

};

**Output SS:**



**Problem-8 : Kth Smallest Element in a BST**  
**Code:**

class Solution {

public:

    int count = 0, result = -1;

    void inorder(TreeNode\* root, int k) {

        if (!root || count >= k) return;

        inorder(root->left, k);

        count++;

        if (count == k) {

            result = root->val;

            return;

        }

        inorder(root->right, k);

    }

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

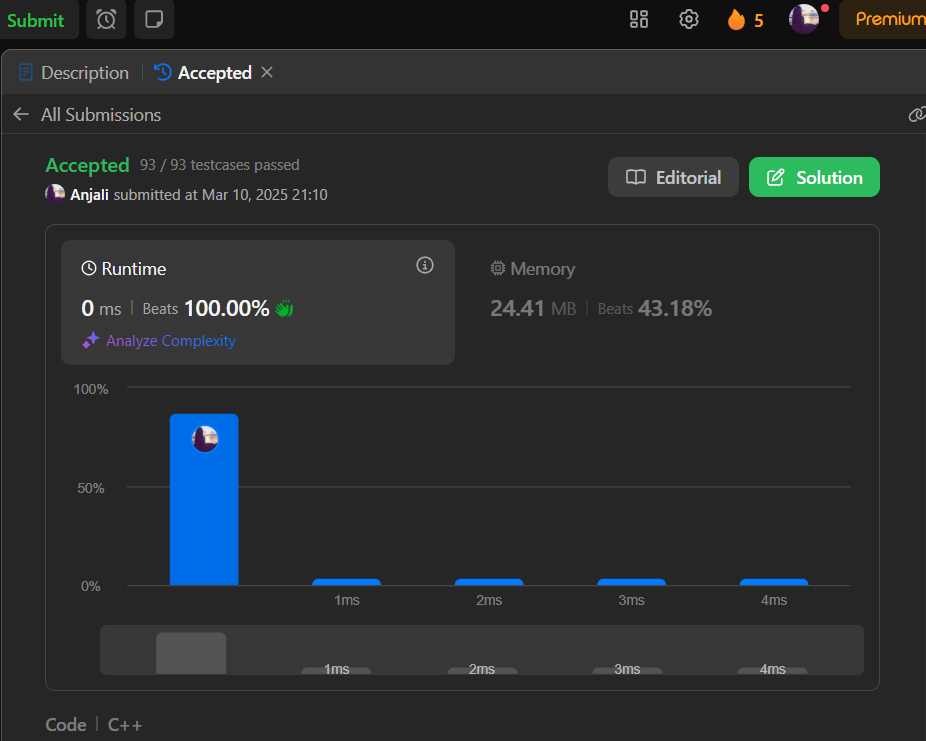
        inorder(root, k);

        return result;

    }

};

**Output SS:**



**Problem-9 : Populating Next Right Pointers in Each Node**  
**Code:**

class Solution {

public:

    Node\* connect(Node\* root) {

        if (!root) return nullptr;

        queue<Node\*> q;

        q.push(root);

        while (!q.empty()) {

            int size = q.size();

            Node\* prev = nullptr;

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

                Node\* curr = q.front();

                q.pop();

                if (prev) prev->next = curr;

                prev = curr;

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

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

            }

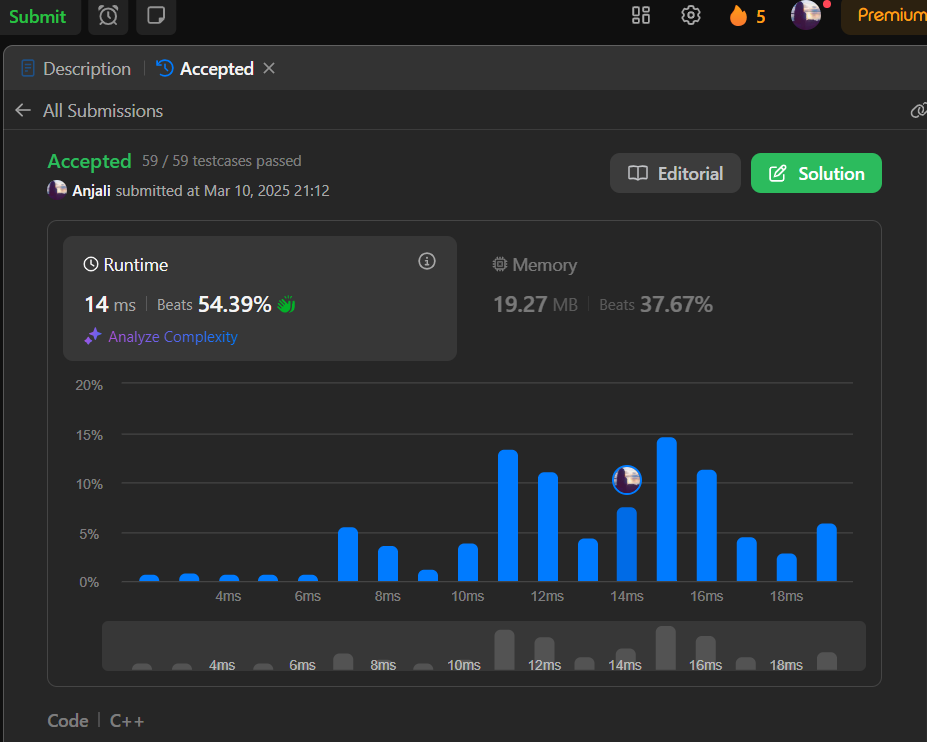
        }

        return root;

    }

};

**Output SS:**



**Problem-10 : Sum of Left Leaves**

**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;

        }

        sum += sumOfLeftLeaves(root->left);

        sum += sumOfLeftLeaves(root->right);

        return sum;

    }

};

**Output SS:**

