AP Assignment 7

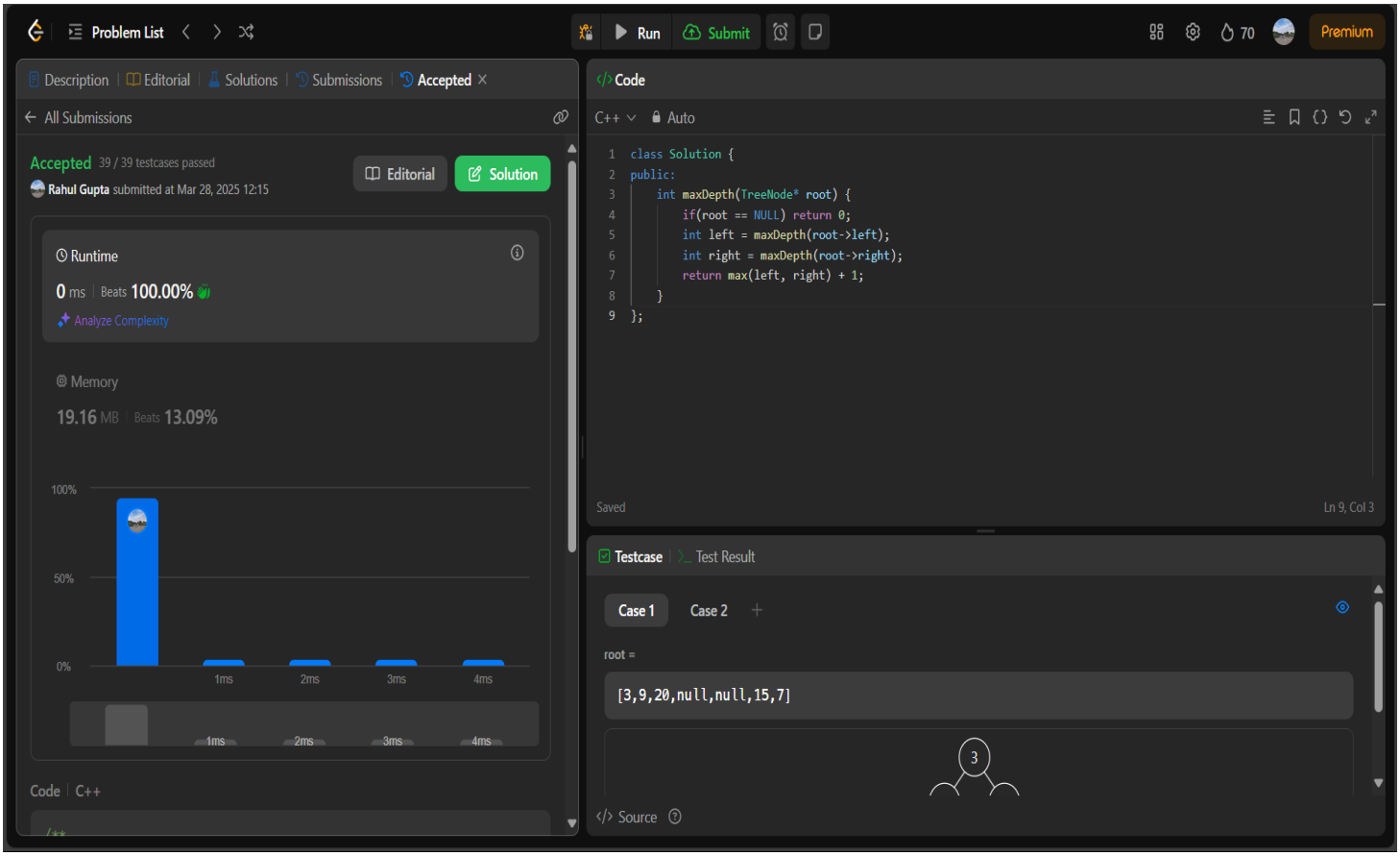
Name-> Ishu

UID-> 22BCS15695

Section-> 609-B

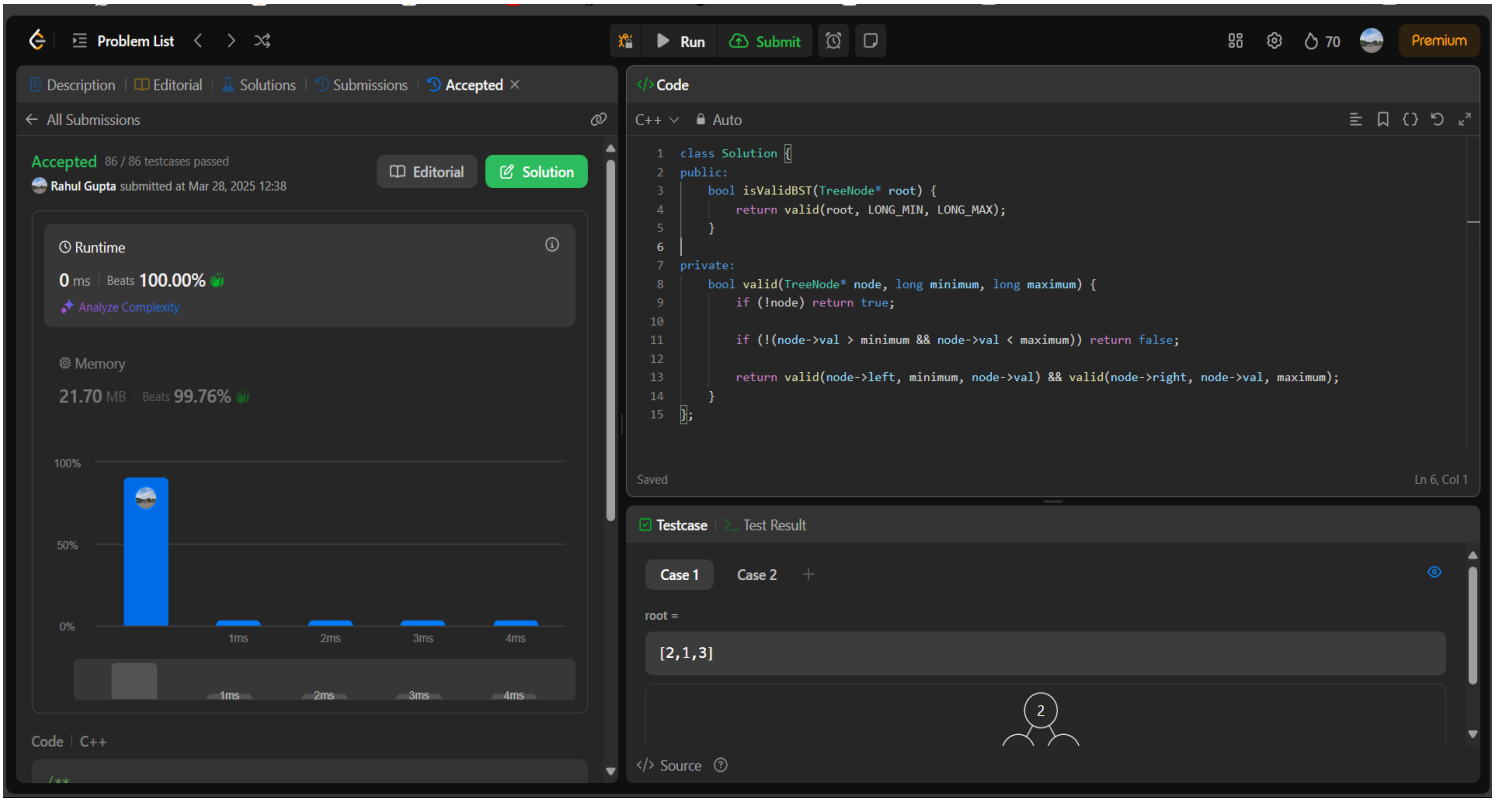
Q1.) Maximum Depth of Binary Tree

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



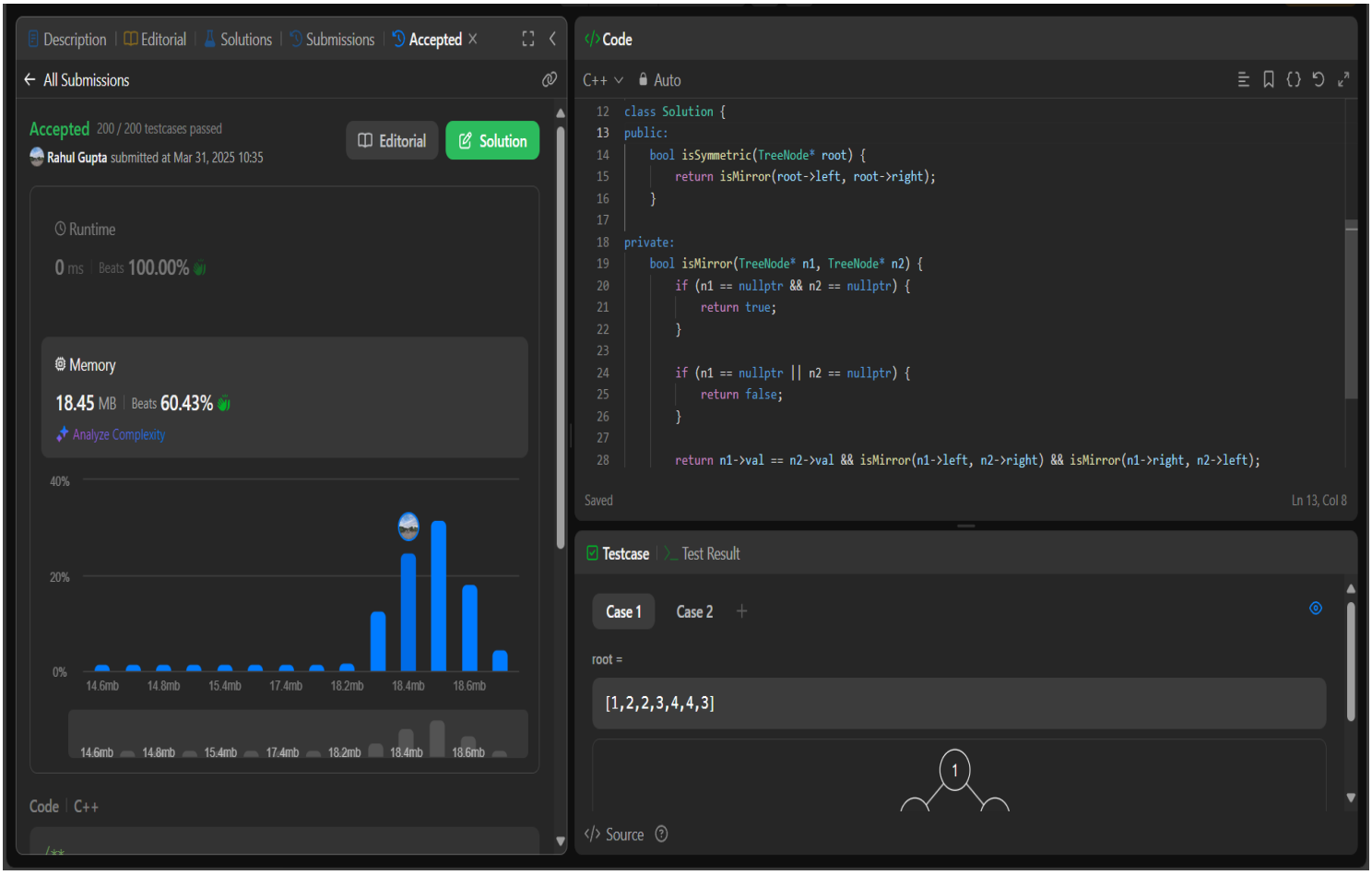
Q2.) Validate Binary Search Tree

```
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        return valid(root, LONG_MIN, LONG_MAX);
    }
private:
    bool valid(TreeNode* node, long minimum, long maximum) {
        if (!node) return true;
        if (!(node->val > minimum && node->val < maximum)) return false;
        return valid(node->left, minimum, node->val) && valid(node->right, node->val, maximum);
    }
};
```



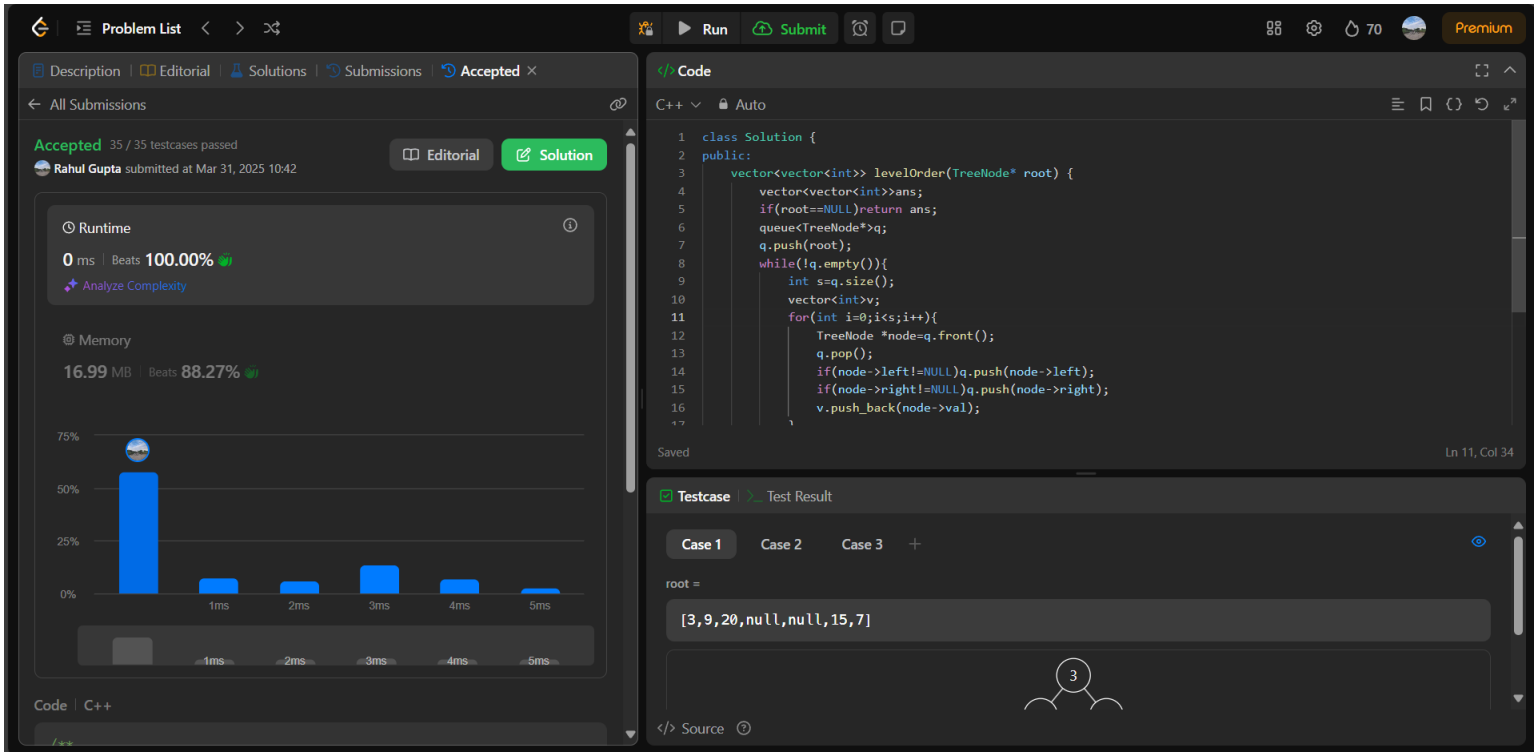
Q3.) Symmetric Tree

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        return isMirror(root->left, root->right);
private:
    bool isMirror(TreeNode* n1, TreeNode* n2) {
        if (n1 == nullptr && n2 == nullptr) {
            return true;
        }
        if (n1 == nullptr || n2 == nullptr) {
            return false;
        }
        return n1->val == n2->val && isMirror(n1->left, n2->right) && isMirror(n1->right, n2->left);
    }
};
```



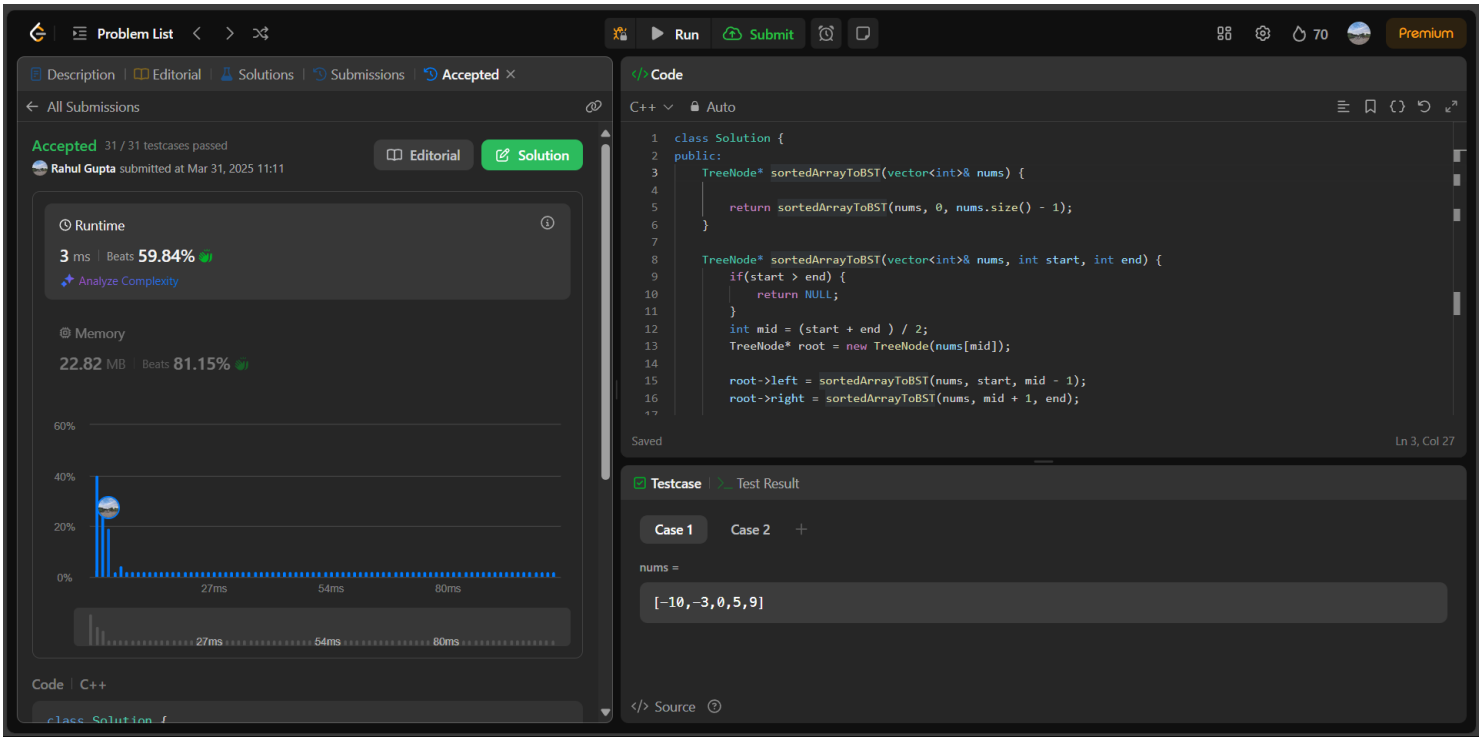
Q4.) [Binary Tree Level Order Traversal]

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



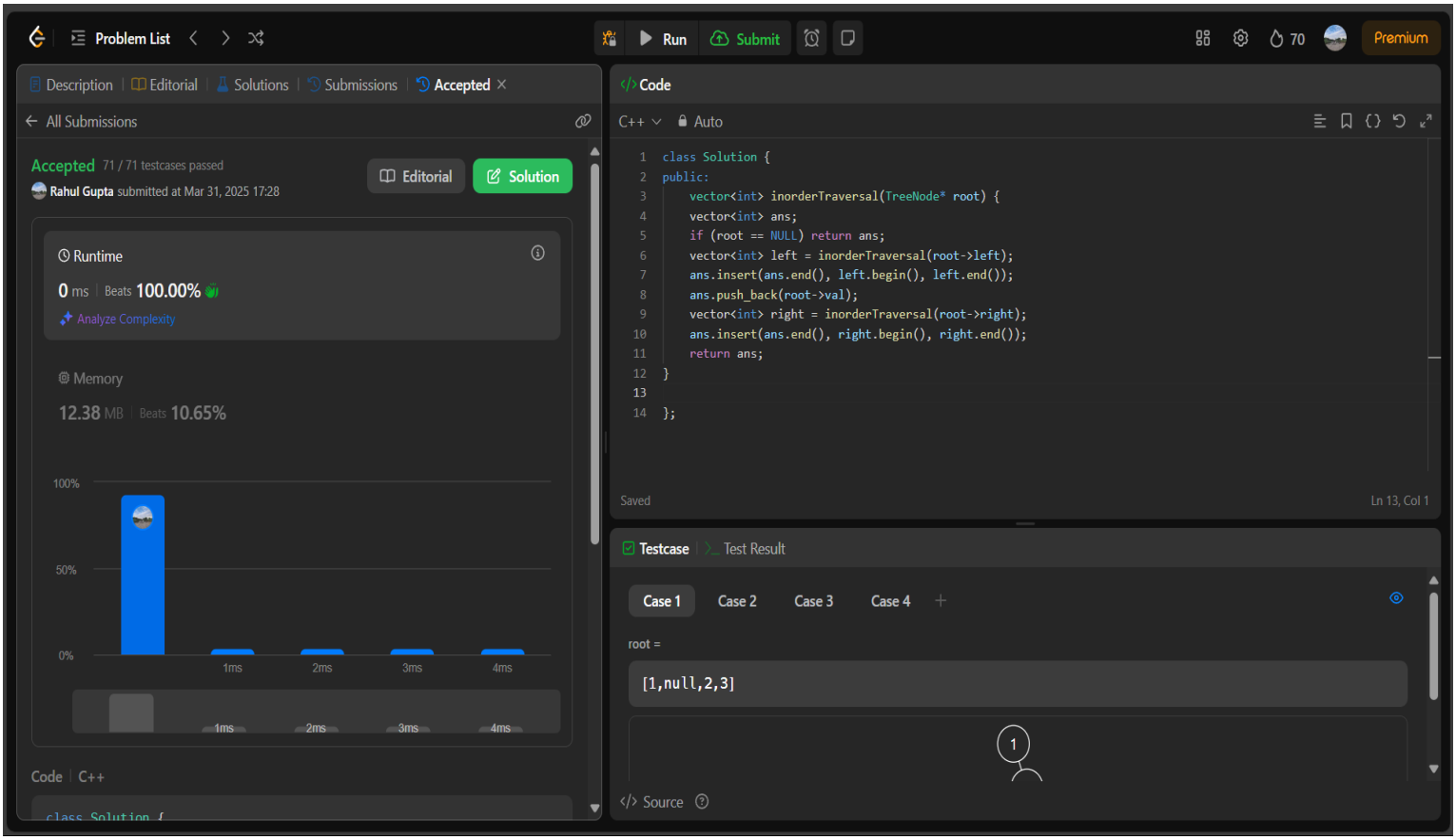
Q6.) [Convert Sorted Array to Binary Search Tree]

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return sortedArrayToBST(nums, 0, nums.size() - 1);
    }
    TreeNode* sortedArrayToBST(vector<int>& nums, int start, int end) {
        if(start > end) {
            return NULL;
        }
        int mid = (start + end ) / 2;
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = sortedArrayToBST(nums, start, mid - 1);
        root->right = sortedArrayToBST(nums, mid + 1, end);
        return root;
    }
};
```



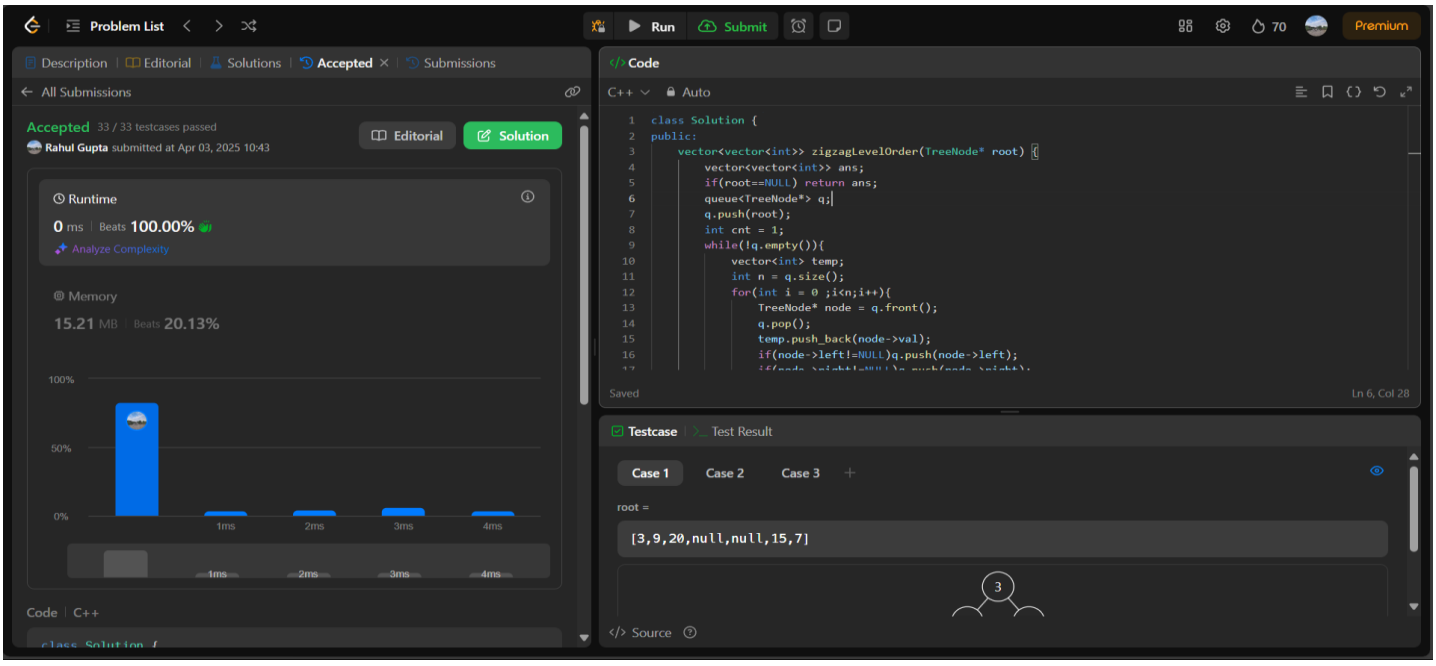
Q7.) [Binary Tree Inorder Traversal]

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
    vector<int> ans;
    if (root == NULL) return ans;
    vector<int> left = inorderTraversal(root->left);
    ans.insert(ans.end(), left.begin(), left.end());
    ans.push_back(root->val);
    vector<int> right = inorderTraversal(root->right);
    ans.insert(ans.end(), right.begin(), right.end());
    return ans;
}
};
```



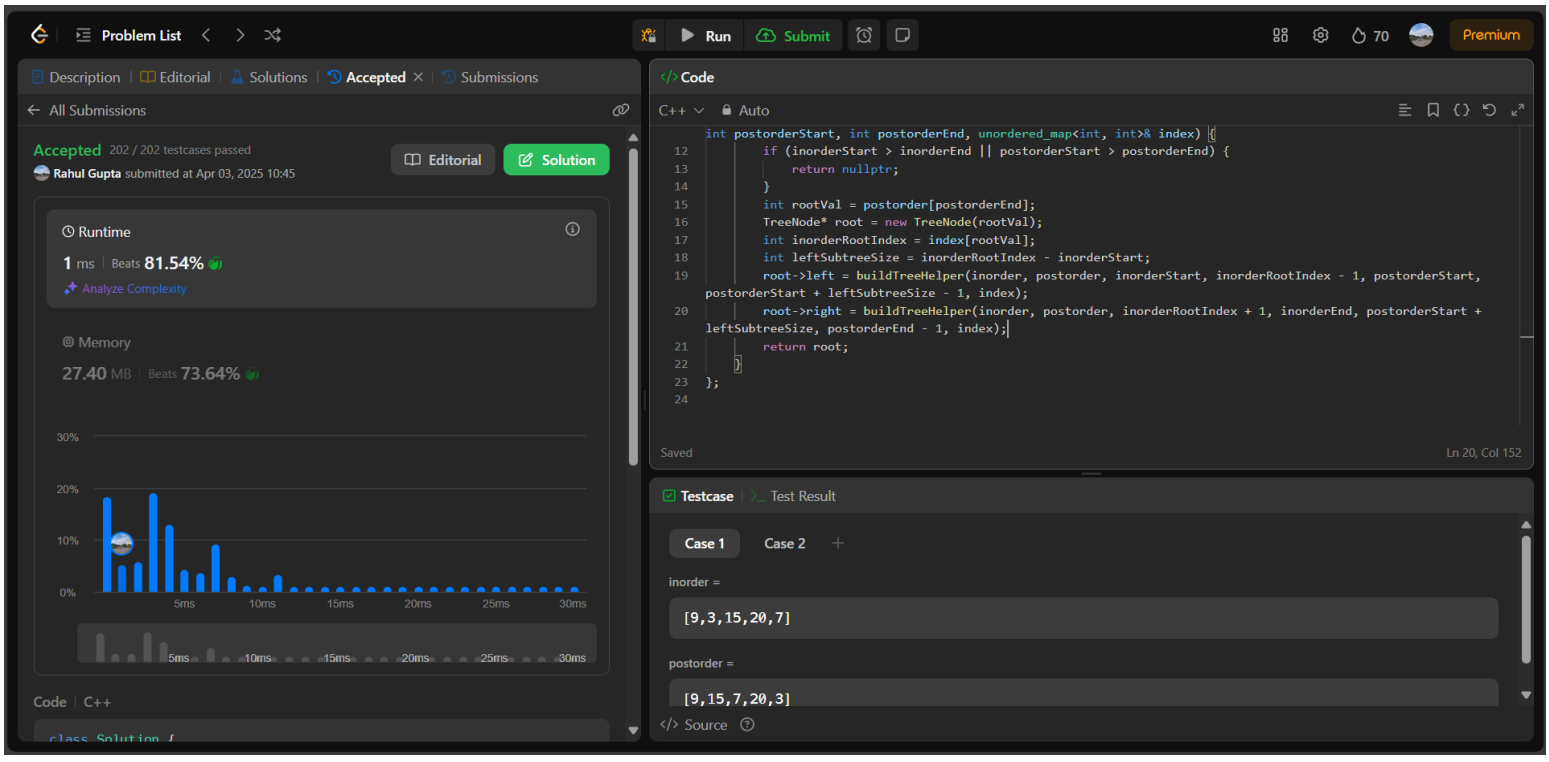
Q8.) [Binary Tree Zigzag Level Order Traversal]

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        if(root==NULL) return ans;
        queue<TreeNode*> q;
        q.push(root);
        int cnt = 1;
        while(!q.empty()){
            vector<int> temp;
            int n = q.size();
            for(int i = 0 ;i<n;i++){
                TreeNode* node = q.front();
                q.pop();
                temp.push_back(node->val);
                if(node->left!=NULL)q.push(node->left);
                if(node->right!=NULL)q.push(node->right);          }
            if(cnt%2==0) reverse(temp.begin() ,temp.end());
            ans.push_back(temp);
            cnt++;}
        return ans; }};
```



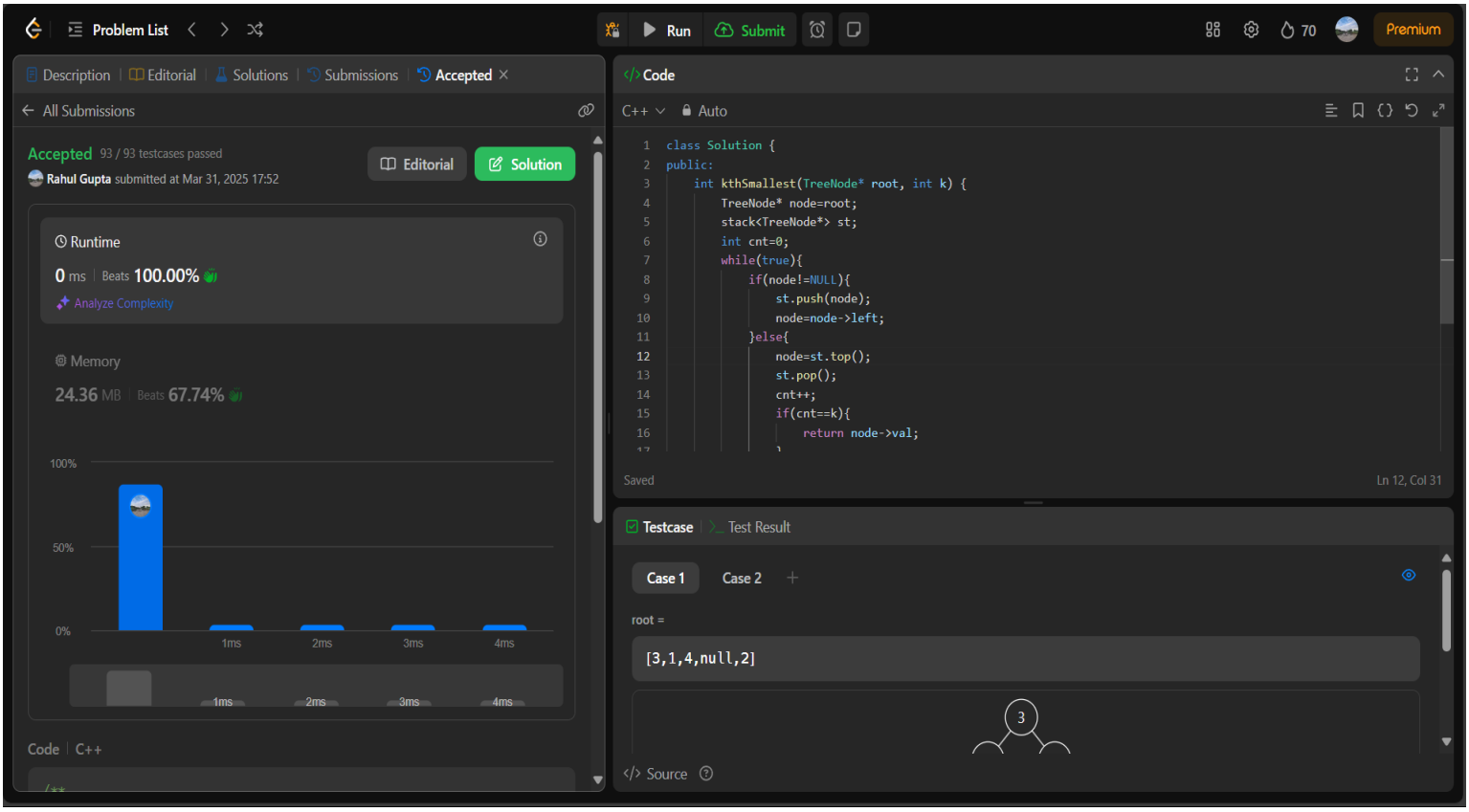
Q9.) Construct Binary Tree from Inorder and Postorder Traversal

```
class Solution {

public:

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

      unordered_map<int, int> index;

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

         index[inorder[i]] = i;

      }

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

   TreeNode* buildTreeHelper(vector<int>& inorder, vector<int>& postorder, int inorderStart, int inorderEnd, int postorderStart, int postorderEnd, unordered_map<int, int>& index) {

      if (inorderStart > inorderEnd || postorderStart > postorderEnd) {

         return nullptr; }

      int rootVal = postorder[postorderEnd];

      TreeNode* root = new TreeNode(rootVal);

      int inorderRootIndex = index[rootVal];

      int leftSubtreeSize = inorderRootIndex - inorderStart;

      root->left = buildTreeHelper(inorder, postorder, inorderStart, inorderRootIndex - 1, postorderStart, postorderStart + leftSubtreeSize - 1, index);

      root->right = buildTreeHelper(inorder, postorder, inorderRootIndex + 1, inorderEnd, postorderStart + leftSubtreeSize, postorderEnd - 1, index);

      return root; } };
```



Q10.) [Kth Smallest Element in a BST]

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        TreeNode* node=root;
        stack<TreeNode*> st;
        int cnt=0;
        while(true){
            if(node!=NULL){
                st.push(node);
                node=node->left;
            }else{
                node=st.top();
                st.pop();
                cnt++;
                if(cnt==k){
                    return node->val;  }
                node=node->right;  }  }
        return -1; } };
```



Q11.) [Populating Next Right Pointers in Each Node]

```
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return nullptr;
        Node* leftMost = root;

        while (leftMost->left) {
            Node* currNode = leftMost;

            while (currNode) {
                currNode->left->next = currNode->right;
                if (currNode->next) {
                    currNode->right->next = currNode->next->left;
                }
                currNode = currNode->next;  }
            leftMost = leftMost->left;  }
        return root; } };
```