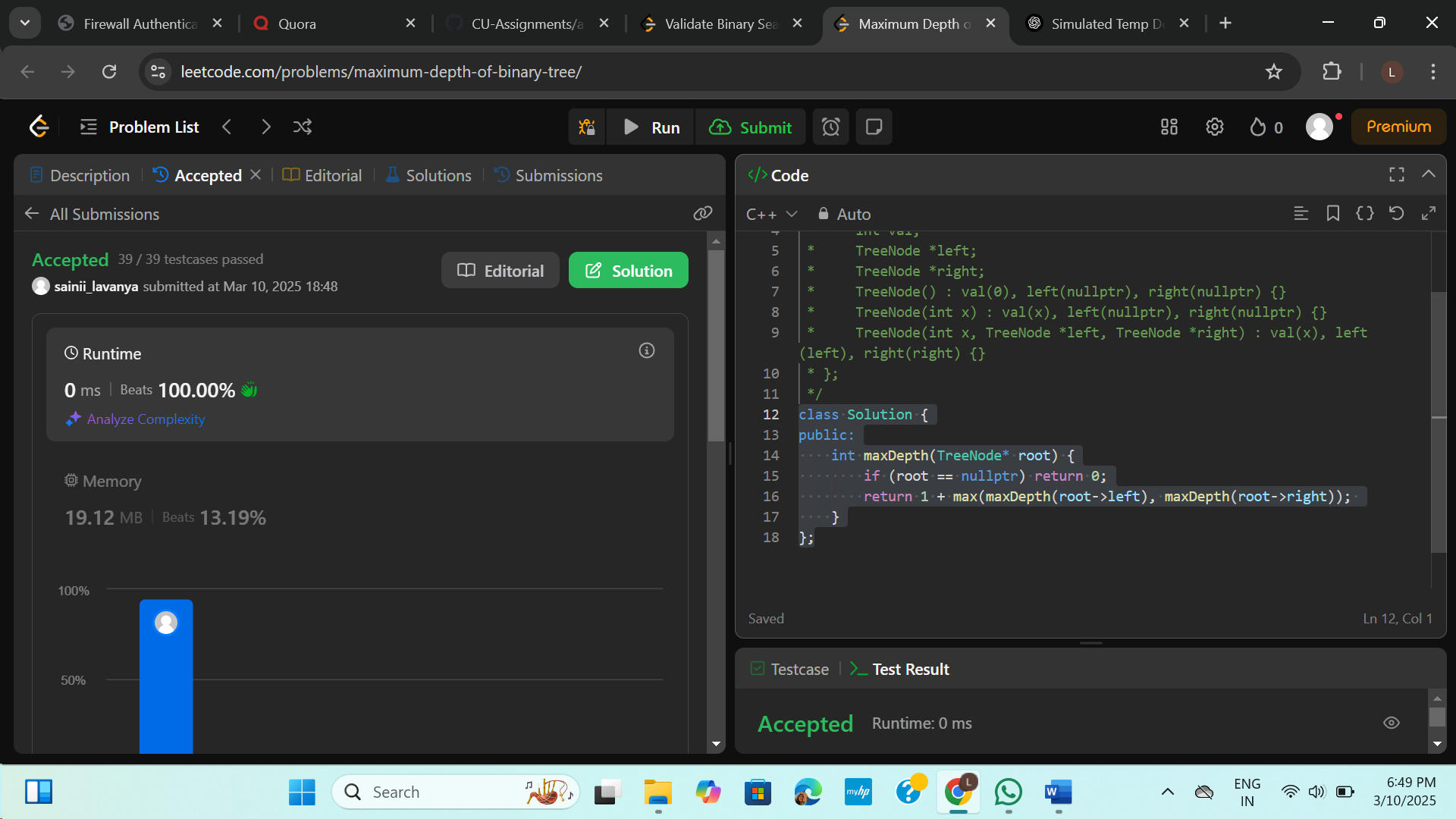
**ASSIGNMENT 5**

**AP LAB**

1. **MAXIMUM DEPTH OF BINARY TREE**

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



**2. VALIDATE BINARY SEARCH TREE**

class Solution {

public:

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

        if (root == nullptr) return true;

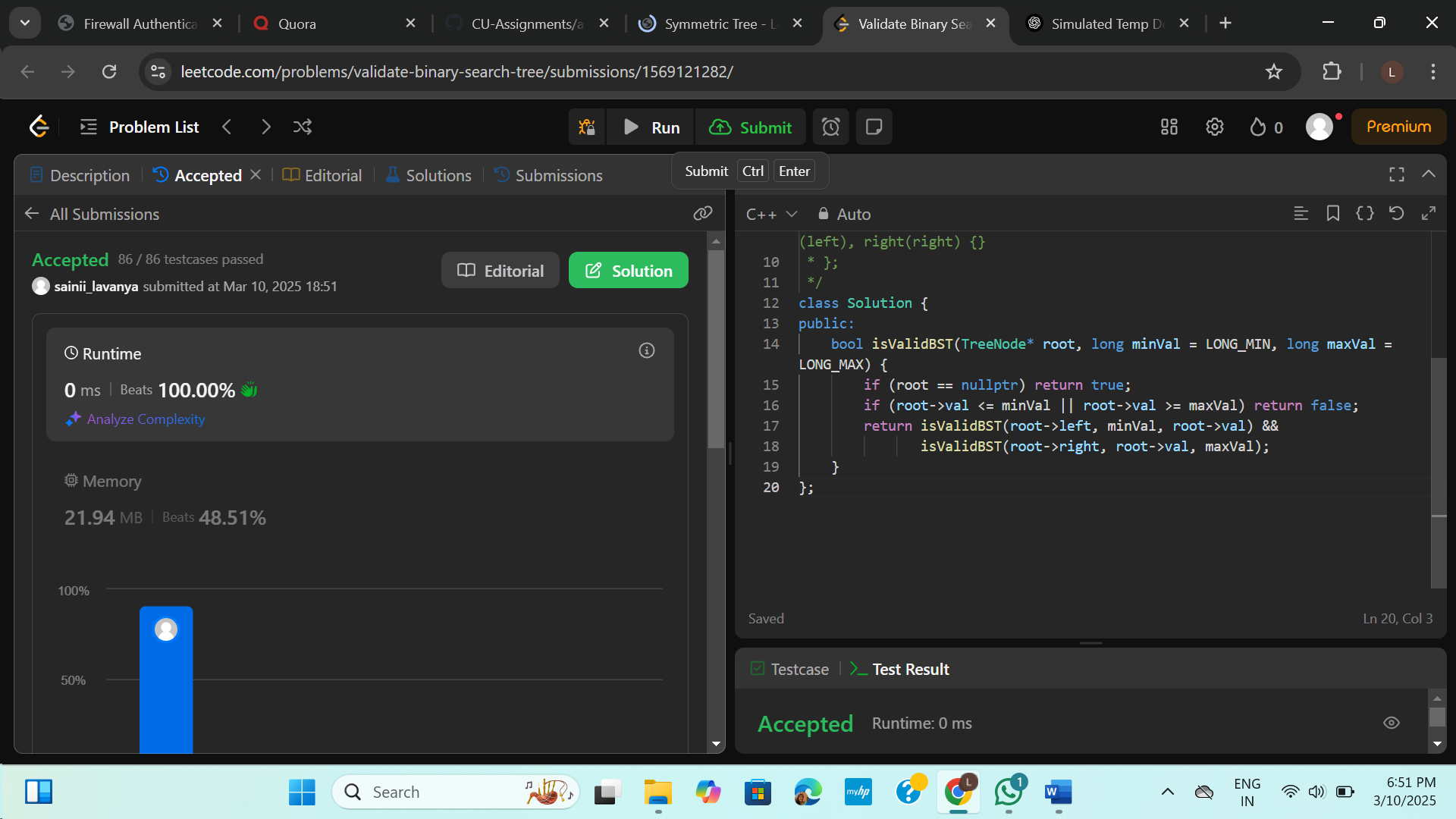
        if (root->val <= minVal || root->val >= maxVal) return false;

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

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

    }

};



**3. SYMMETRIC TREE**

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);

    }

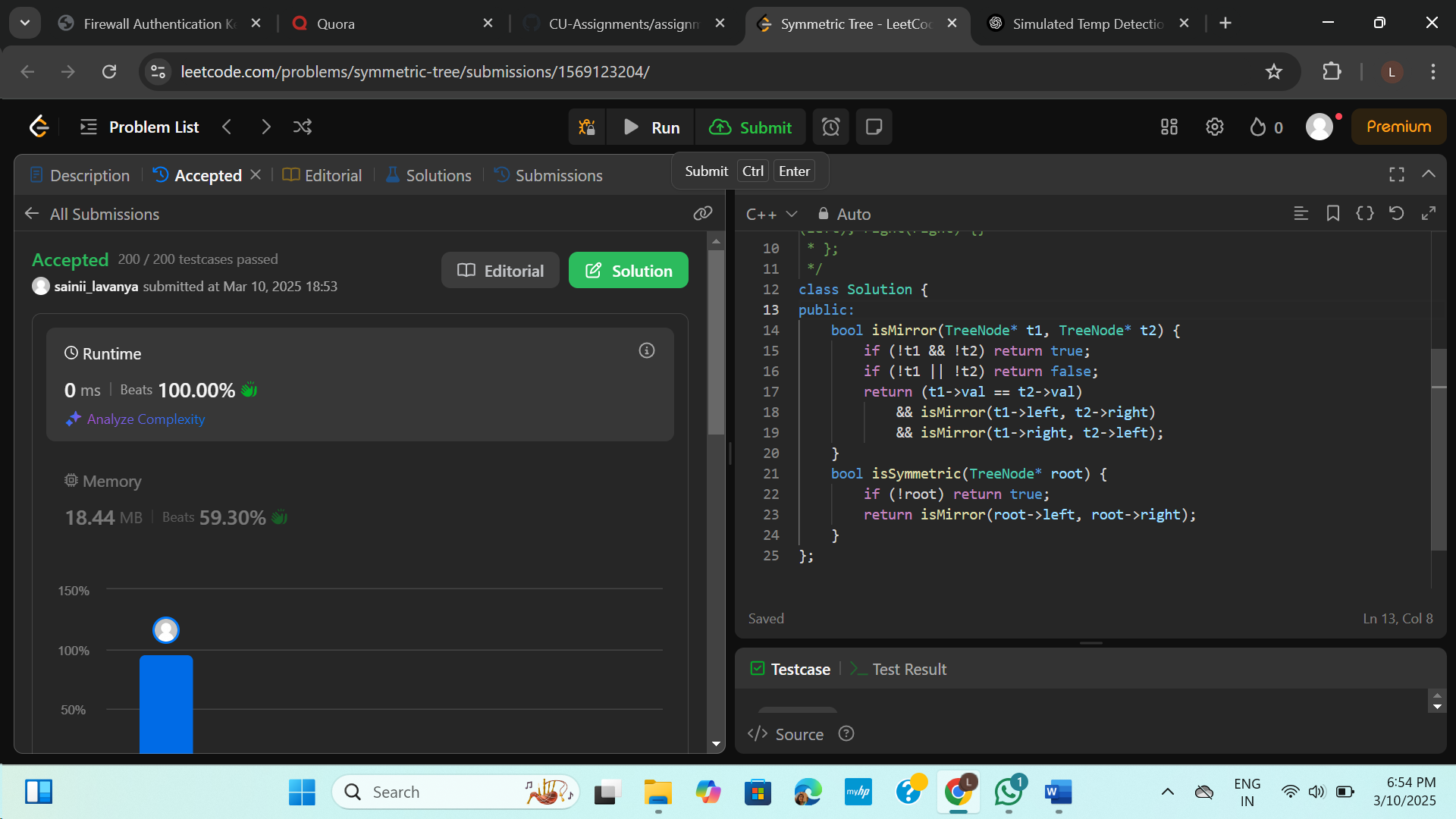
    bool isSymmetric(TreeNode\* root) {

        if (!root) return true;

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

    }

};



**4. BINARY TREE ZIGZAG LEVEL ORDER TRAVERSAL**

class Solution {

public:

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

        vector<vector<int>> result;

        if (!root) return result;

        queue<TreeNode\*> q;

        q.push(root);

        int count = 0;

        while (!q.empty()) {

            int size = q.size();

            vector<int> currentLevel;

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

                TreeNode\* node = q.front();

                q.pop();

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

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

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

            }

            if (count % 2 == 1) {

                reverse(currentLevel.begin(), currentLevel.end());

            }

            result.push\_back(currentLevel);

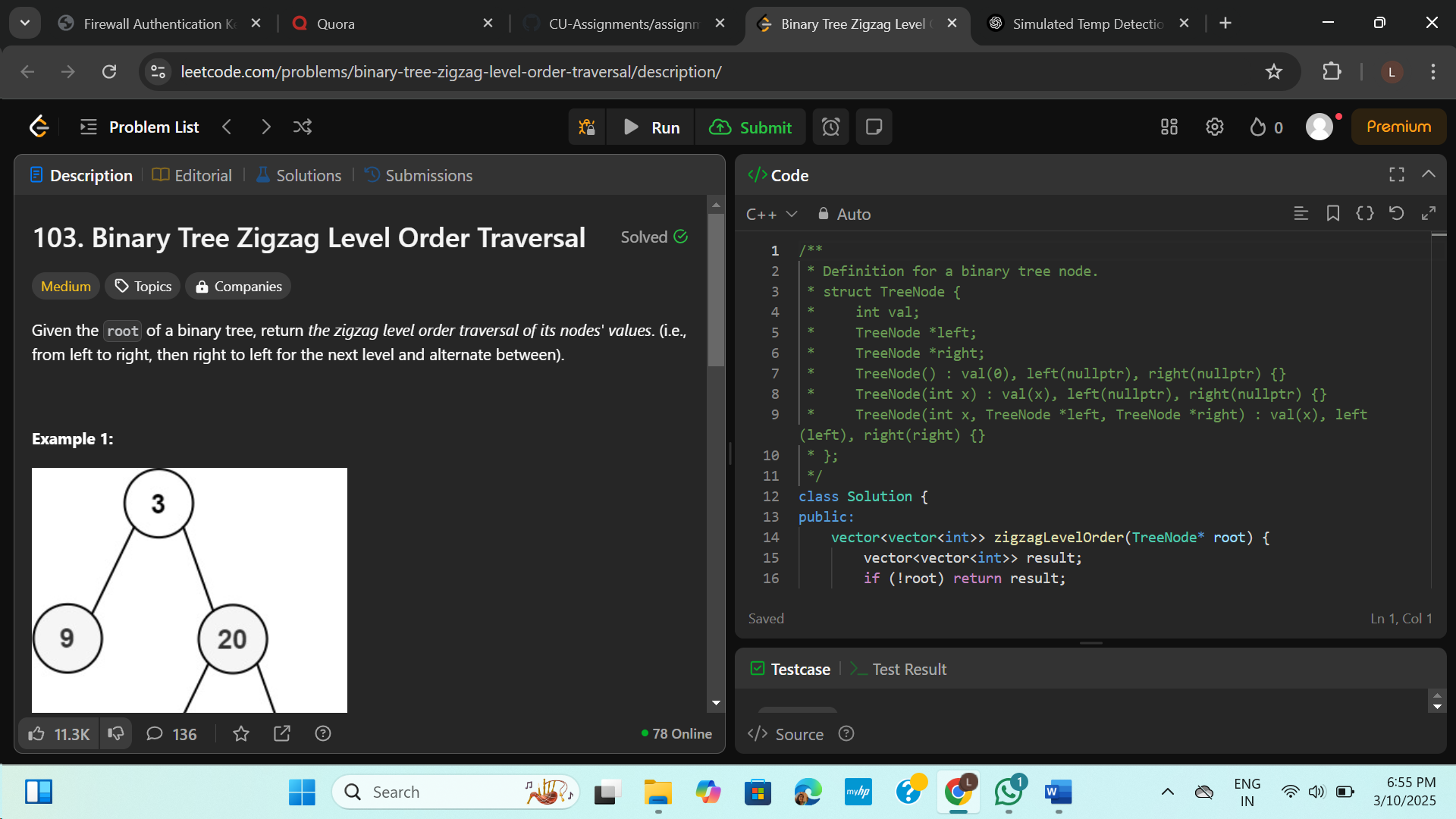
            count++;

        }

        return result;

    }

};



**5. LOWEST COMMON ANCESTOR OF A BINARY TREE**

class Solution {

public:

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

        if (!root || root == p || root == q) return root;

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

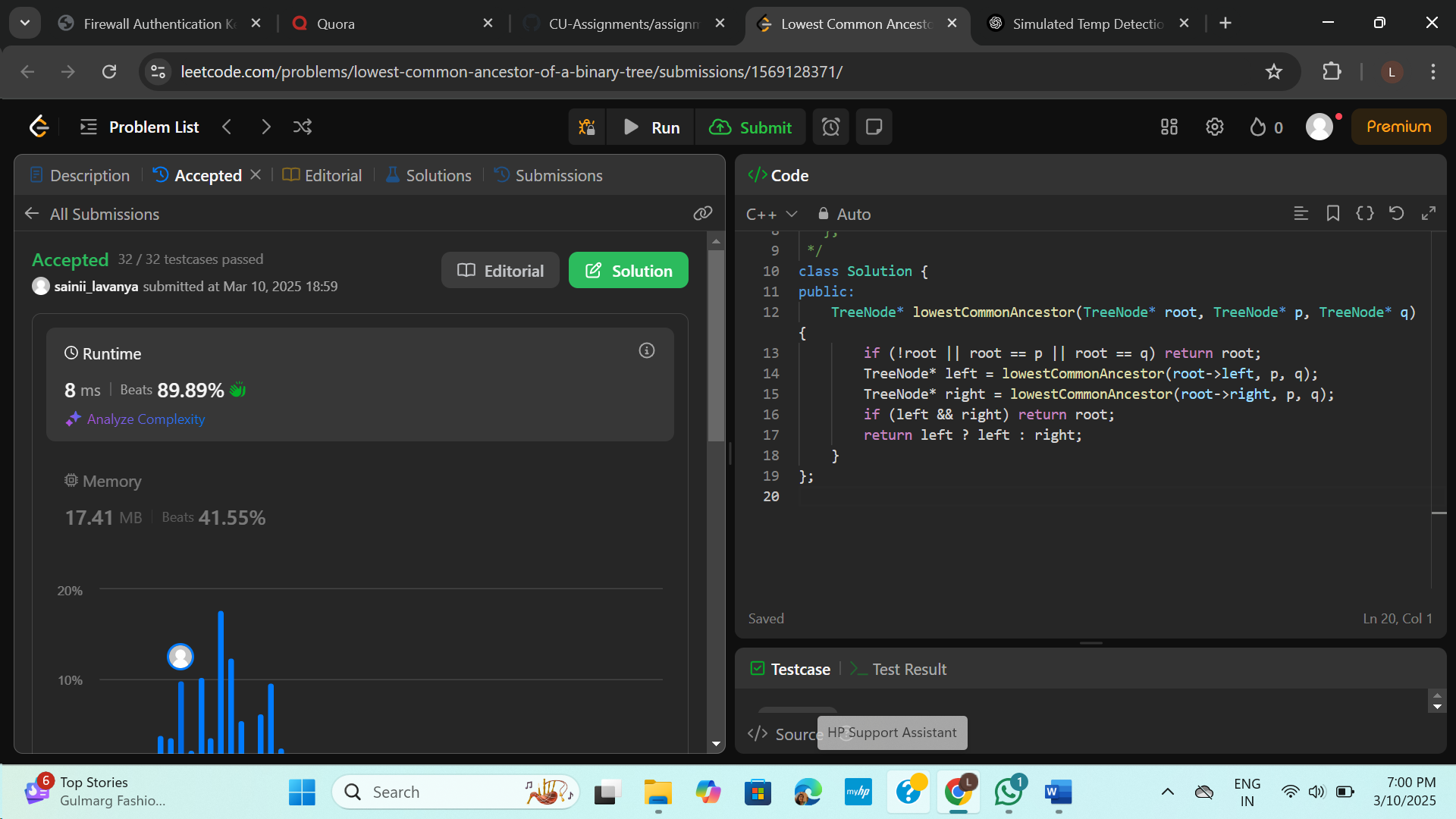
        TreeNode\* right = lowestCommonAncestor(root->right, p, q);

        if (left && right) return root;

        return left ? left : right;

    }

};



**6. BINARY TREE INORDER TRAVERSAL**

class Solution {

public:

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

       if (root == nullptr) return;

        inorder(root->left, result);

        result.push\_back(root->val);

        inorder(root->right, result);

    }

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

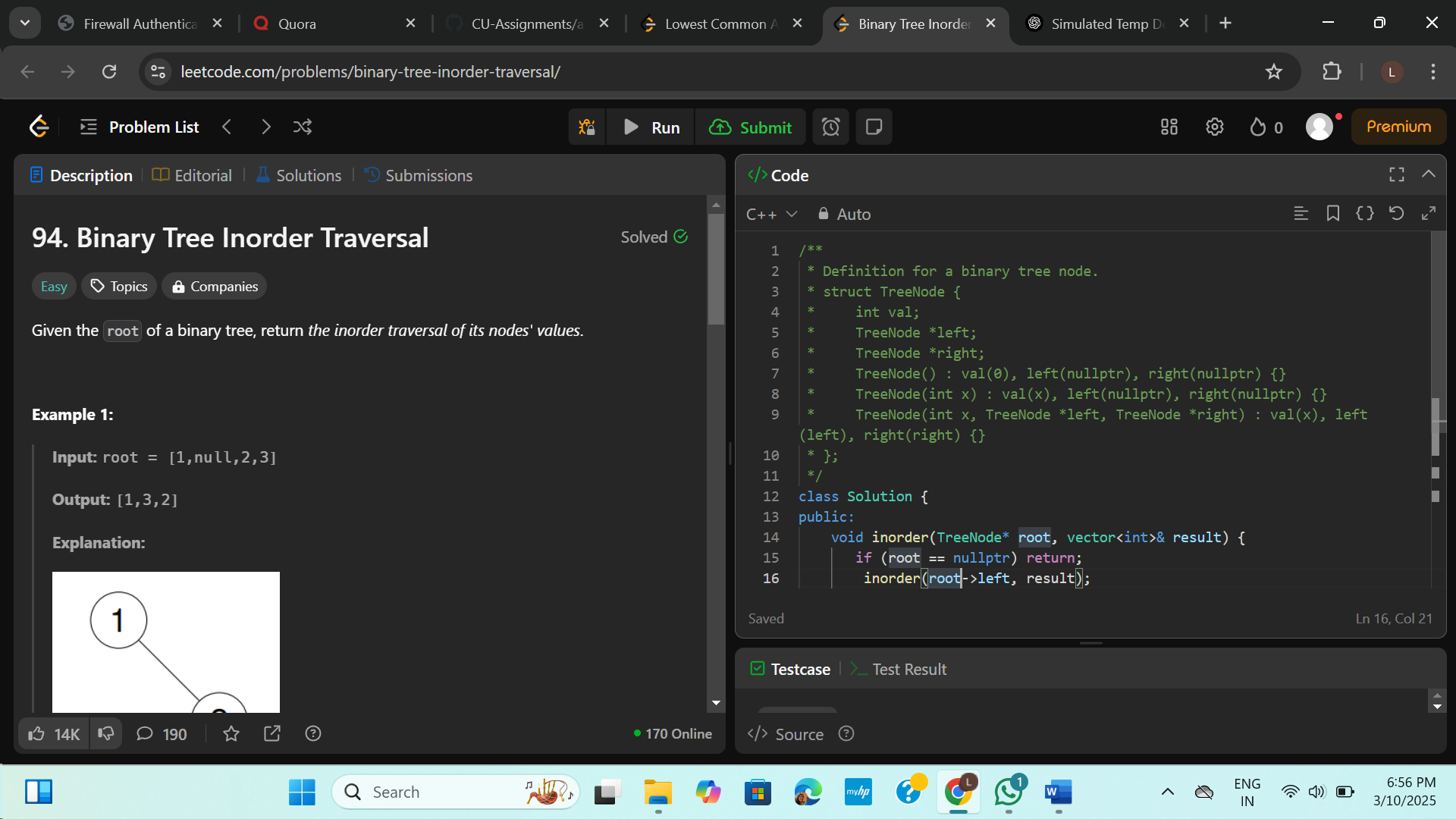
        vector<int> result;

        inorder(root, result);

        return result;

    }

};



**7. BINARY TREE LEVEL ORDER TRAVERSAL**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);

            }

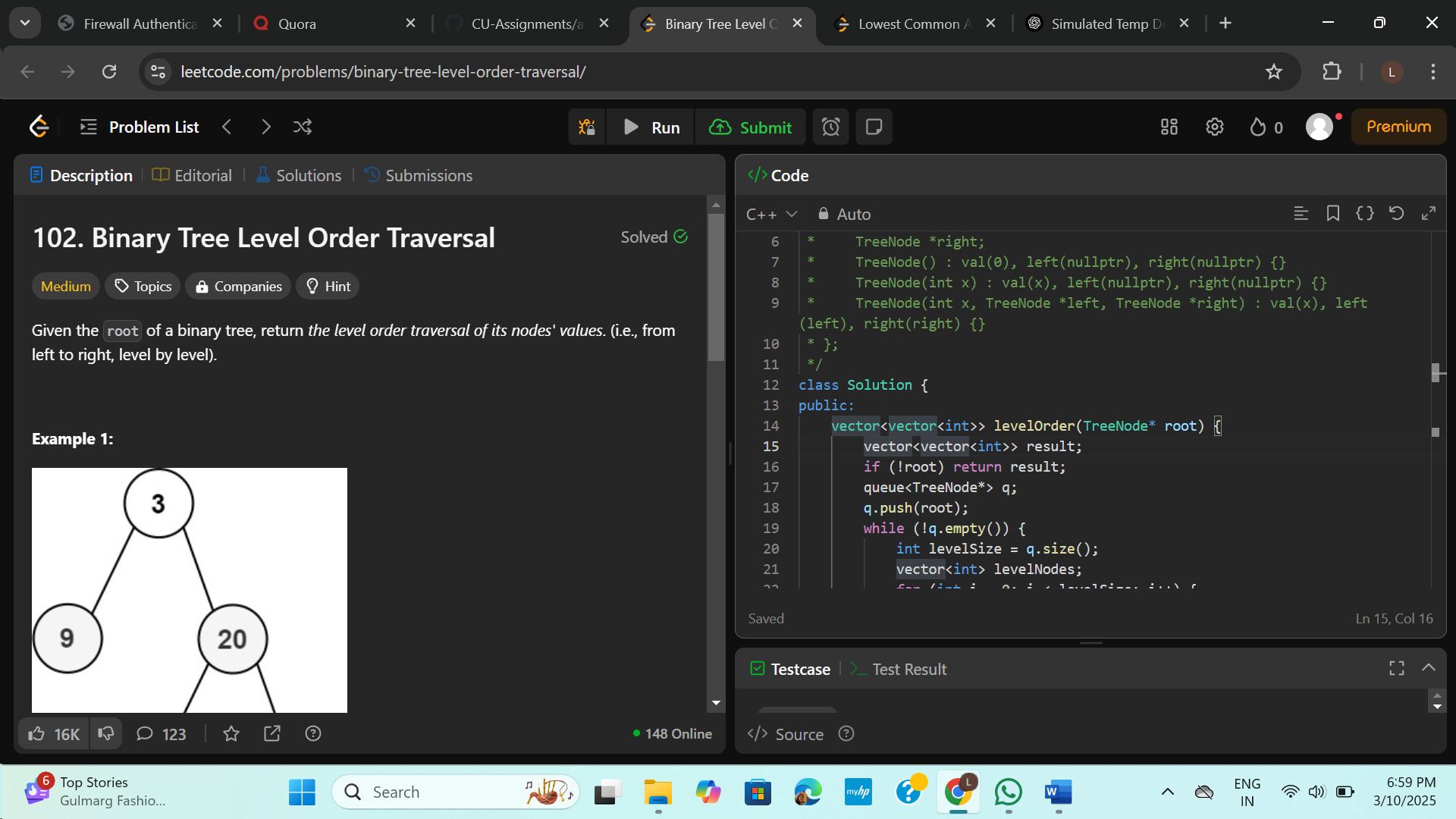
            result.push\_back(levelNodes);

        }

        return result;

    }

};



**8. KTH SMALLEST ELEMENT IN BST**   
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);

    }

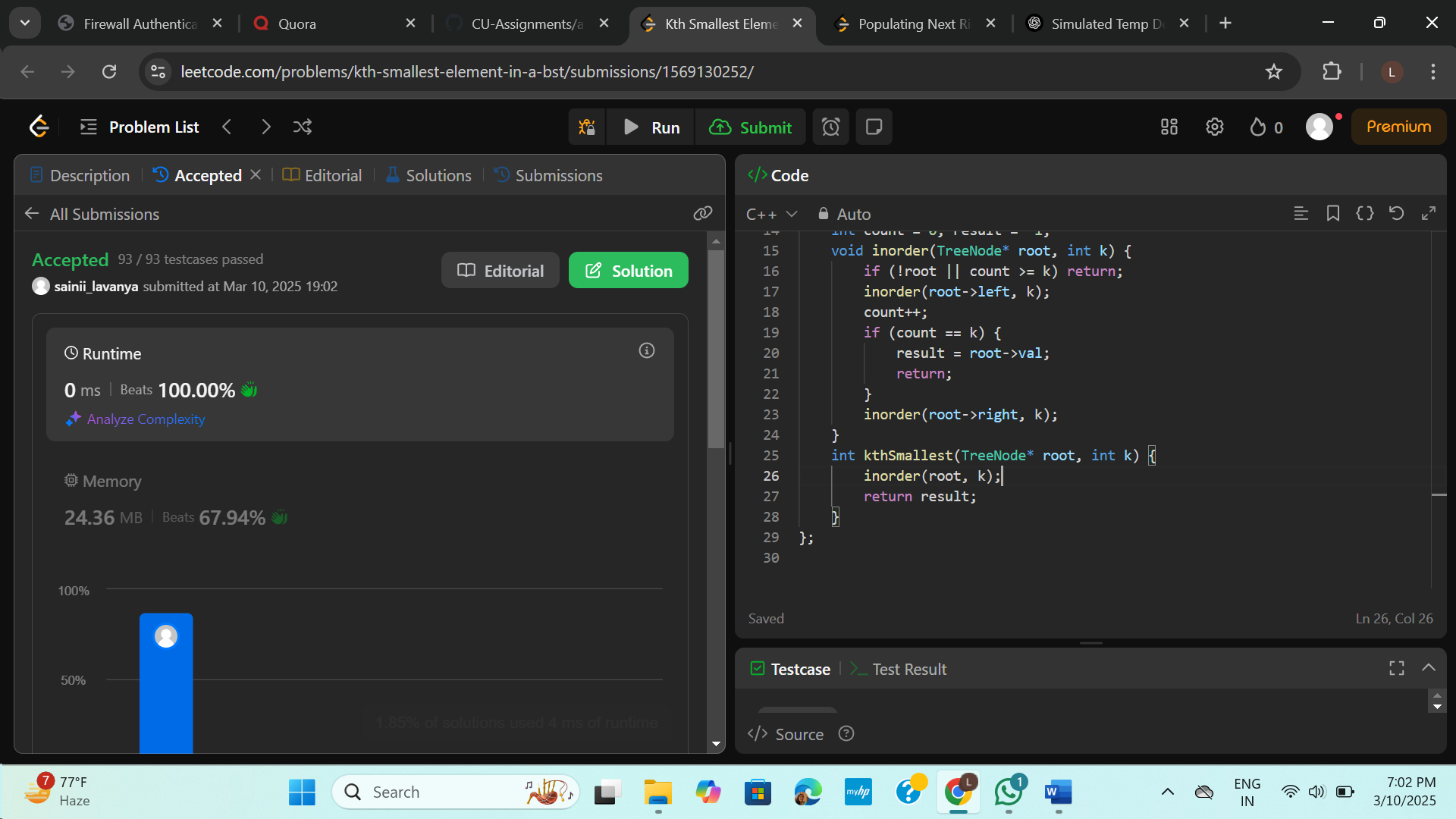
    int kthSmallest(TreeNode\* root, int k) {

        inorder(root, k);

        return result;

    }

};



**9. POPULATING NEXT RIGHT POINTERS IN EACH NODE**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);

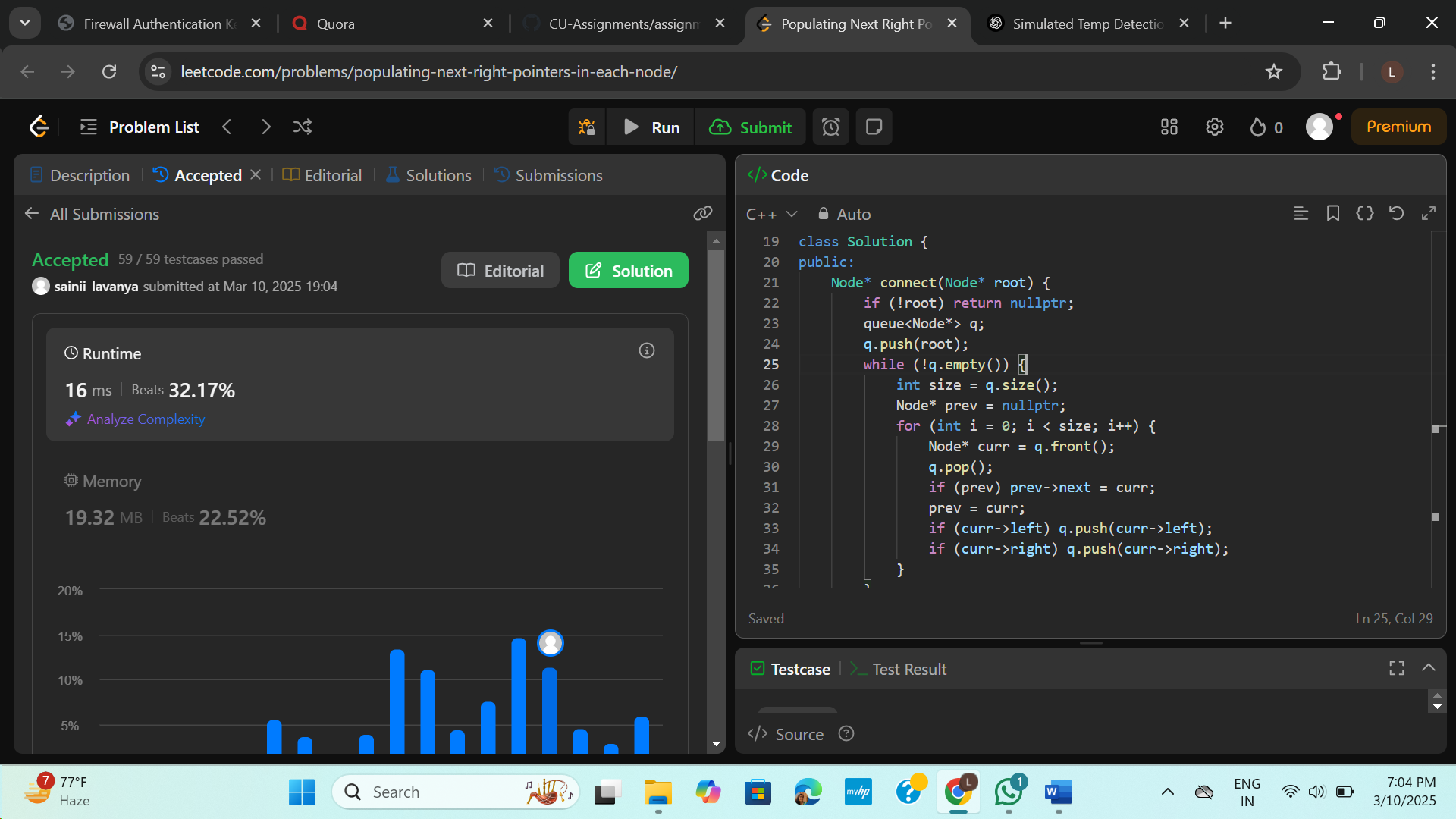
            }

        }

        return root;

    }

};



**10. SUM OF LEFT LEAVES**

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;

    }

};

