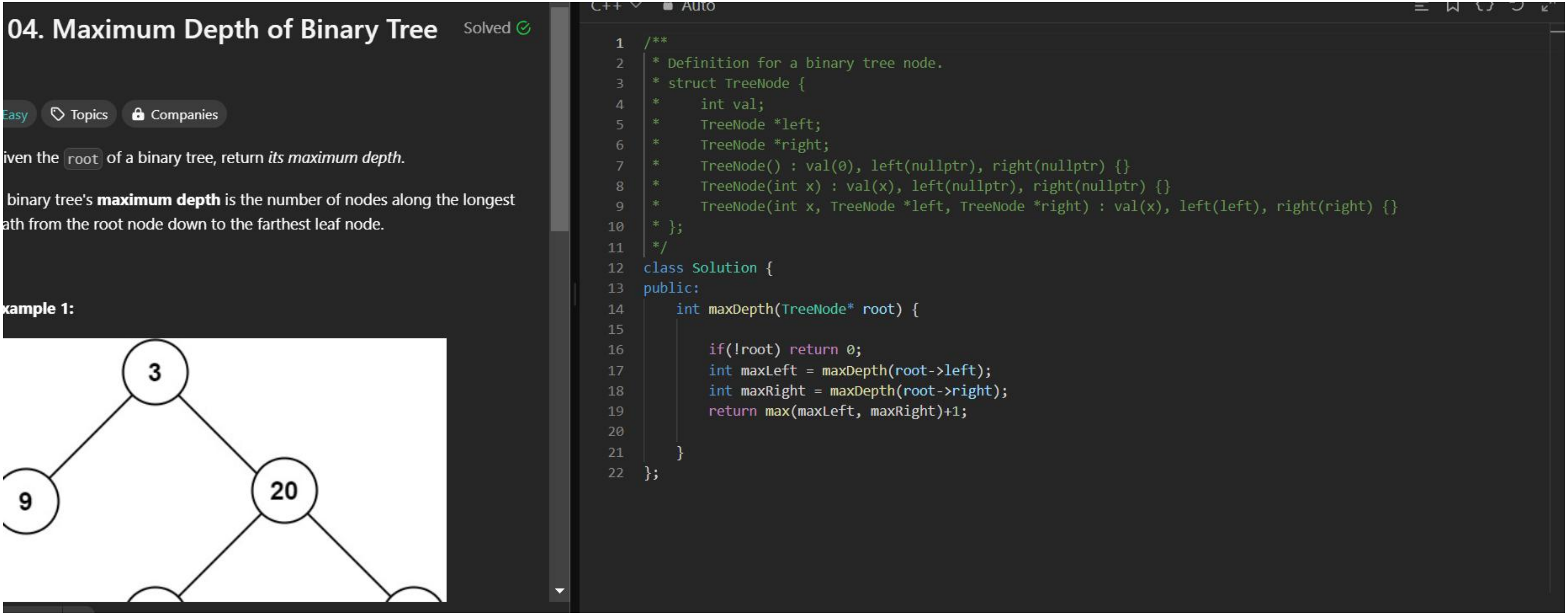


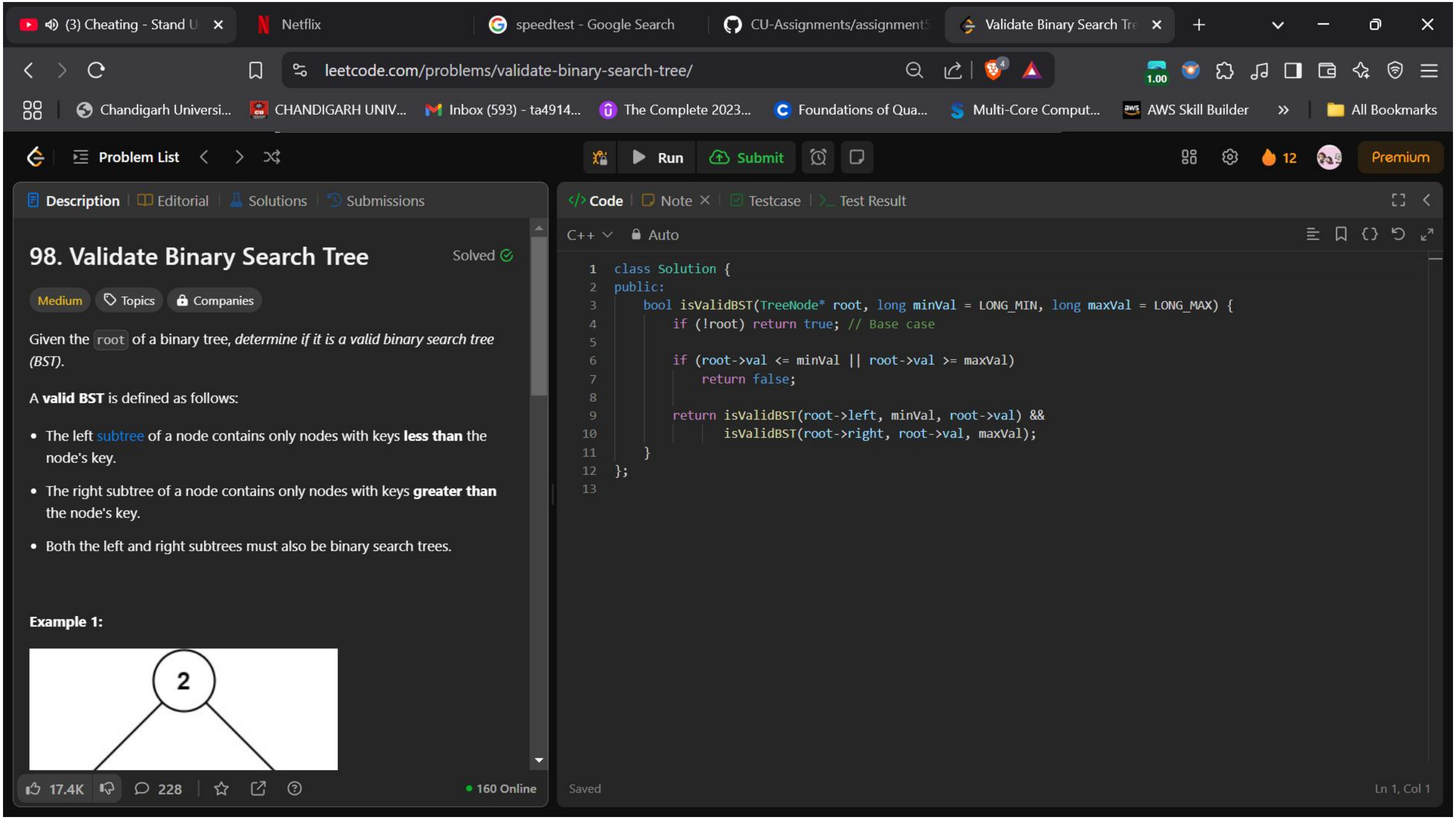
04. Maximum Depth of Binary Tree

iven the `root` of a binary tree, return *its maximum depth*.

binary tree's **maximum depth** is the number of nodes along the longest ath from the root node down to the farthest leaf node.

xample 1:

```cpp
/**
 * 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) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int maxDepth(TreeNode* root) {

        if(!root) return 0;
        int maxLeft = maxDepth(root->left);
        int maxRight = maxDepth(root->right);
        return max(maxLeft, maxRight)+1;

    }
};
```



98. Validate Binary Search Tree

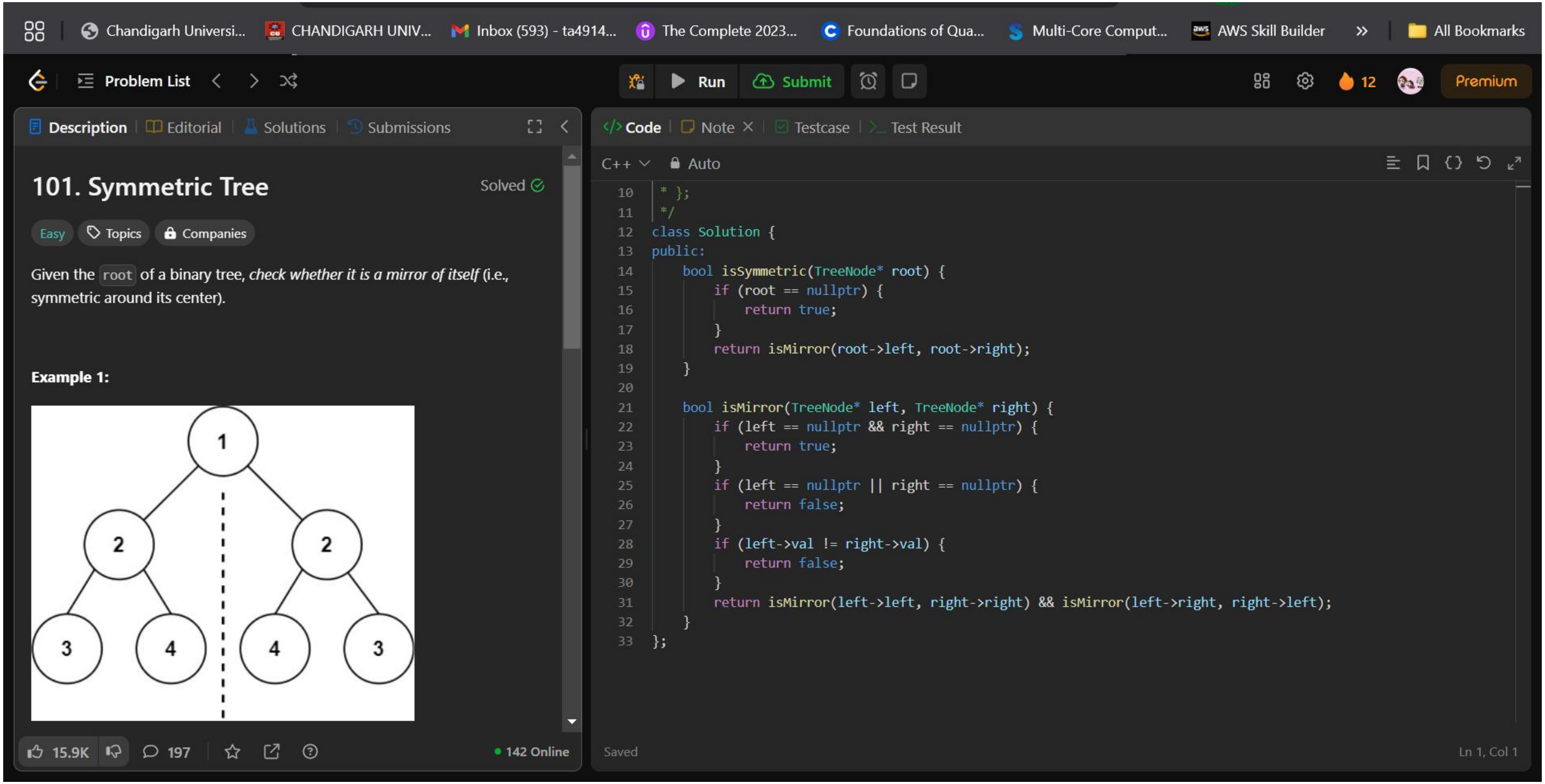
Given the `root` of a binary tree, *determine if it is a valid binary search tree (BST).*

A **valid BST** is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:

```cpp
class Solution {
public:
    bool isValidBST(TreeNode* root, long minVal = LONG_MIN, long maxVal = LONG_MAX) {
        if (!root) return true; // Base case

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

        return isValidBST(root->left, minVal, root->val) &&
            isValidBST(root->right, root->val, maxVal);
    }
};

```
]



101. Symmetric Tree

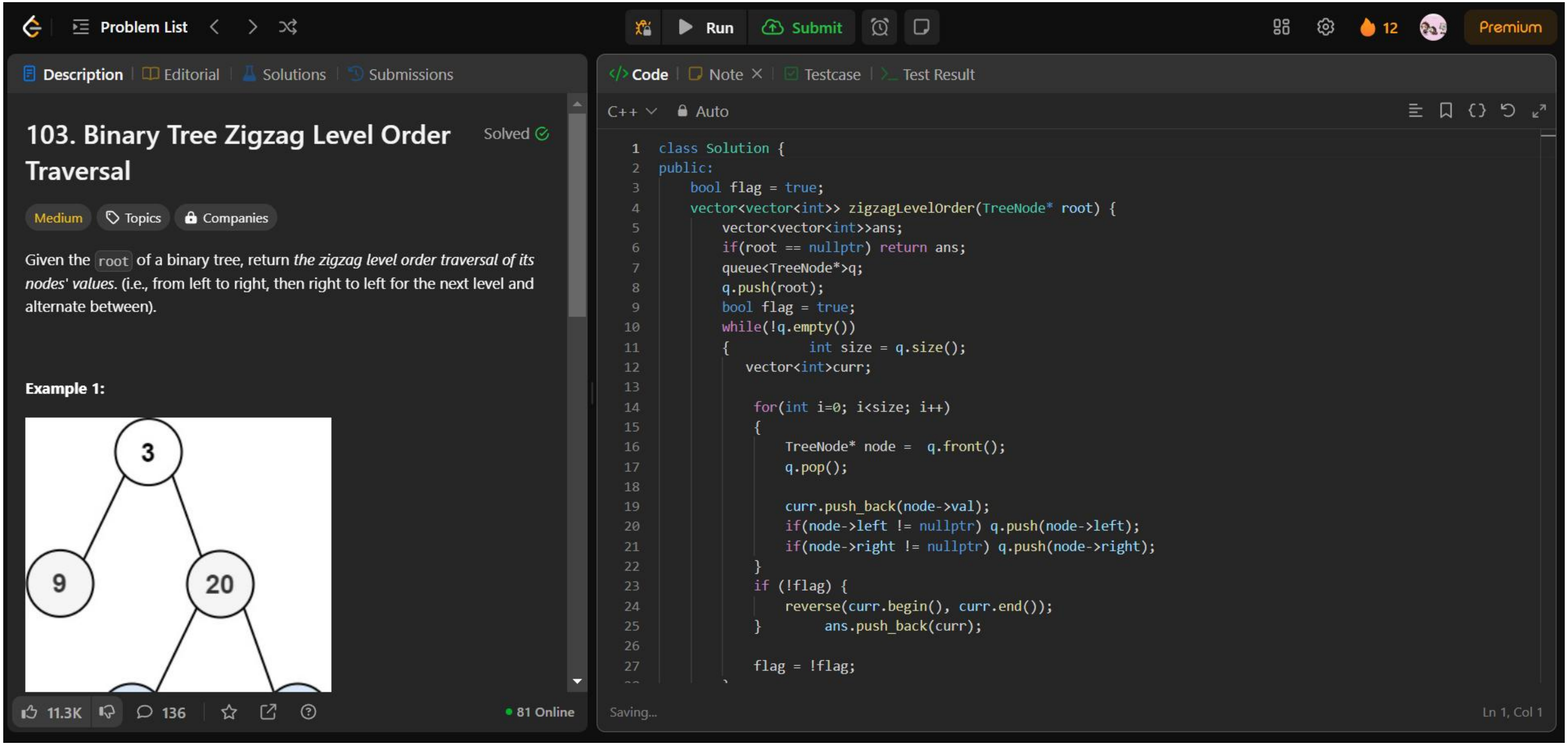
Solved

Easy · Topics · Companies

Given the `root` of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

Example 1:

```cpp
 * };
 */
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if (root == nullptr) {
            return true;
        }
        return isMirror(root->left, root->right);
    }

    bool isMirror(TreeNode* left, TreeNode* right) {
        if (left == nullptr && right == nullptr) {
            return true;
        }
        if (left == nullptr || right == nullptr) {
            return false;
        }
        if (left->val != right->val) {
            return false;
        }
        return isMirror(left->left, right->right) && isMirror(left->right, right->left);
    }
};
```



103. Binary Tree Zigzag Level Order Traversal

Solved

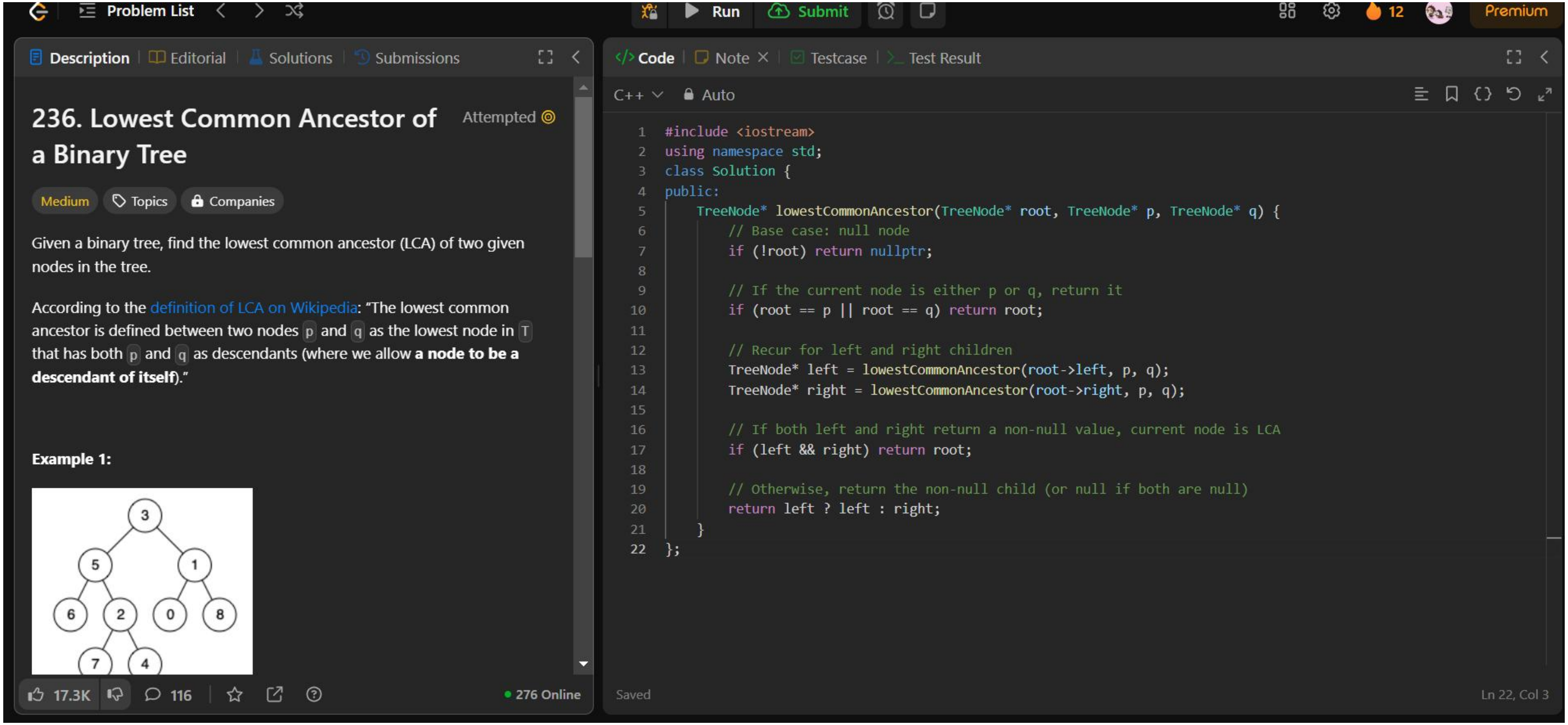
Given the `root` of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

Example 1:

```
class Solution {
public:
    bool flag = true;
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>>ans;
        if(root == nullptr) return ans;
        queue<TreeNode*>q;
        q.push(root);
        bool flag = true;
        while(!q.empty())
        {          int size = q.size();
           vector<int>curr;

            for(int i=0; i<size; i++)
            {
                TreeNode* node =  q.front();
                q.pop();

                curr.push_back(node->val);
                if(node->left != nullptr) q.push(node->left);
                if(node->right != nullptr) q.push(node->right);
            }
            if (!flag) {
                reverse(curr.begin(), curr.end());
            }        ans.push_back(curr);

            flag = !flag;
```

Saving...

Ln 1, Col 1



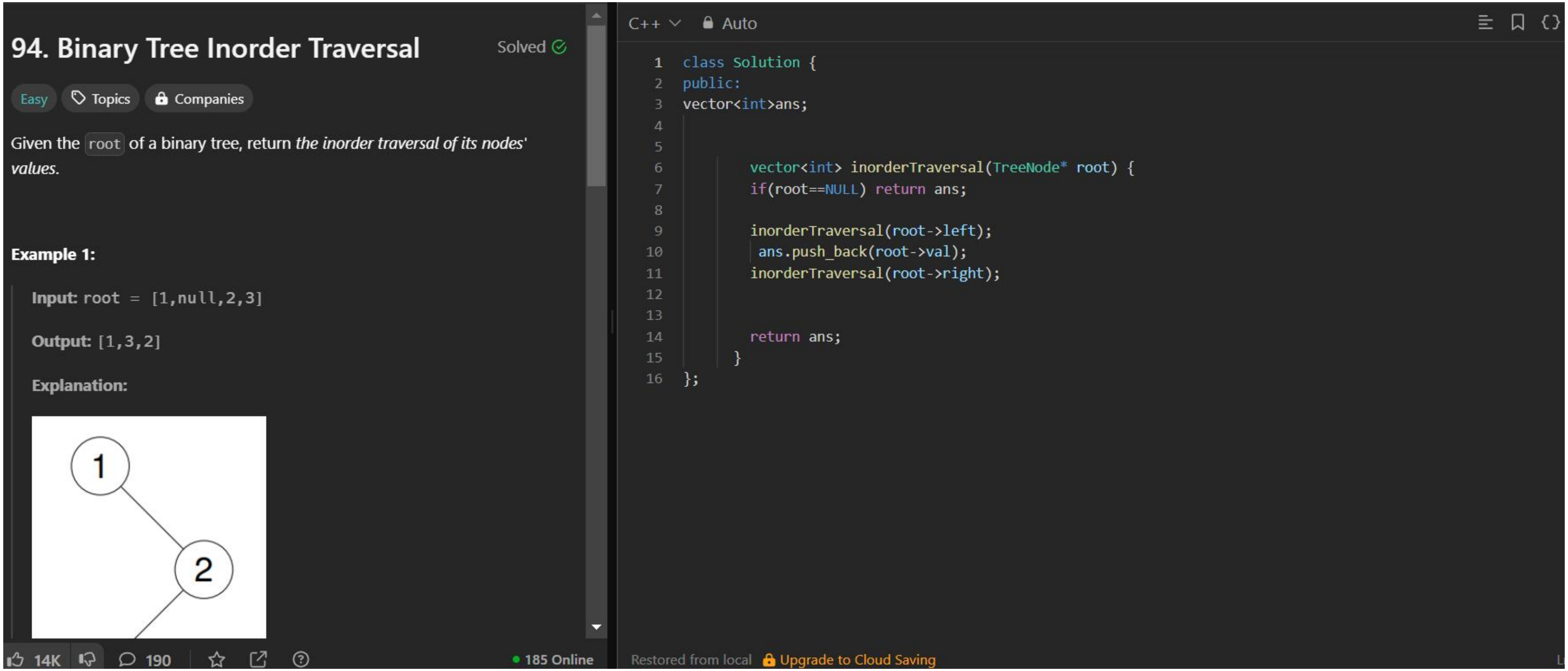
236. Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes `p` and `q` as the lowest node in `T` that has both `p` and `q` as descendants (where we allow **a node to be a descendant of itself**)."

Example 1:

```cpp
#include <iostream>
using namespace std;
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        // Base case: null node
        if (!root) return nullptr;

        // If the current node is either p or q, return it
        if (root == p || root == q) return root;

        // Recur for left and right children
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);

        // If both left and right return a non-null value, current node is LCA
        if (left && right) return root;

        // Otherwise, return the non-null child (or null if both are null)
        return left ? left : right;
    }
};
```



94. Binary Tree Inorder Traversal

Given the `root` of a binary tree, return *the inorder traversal of its nodes' values.*

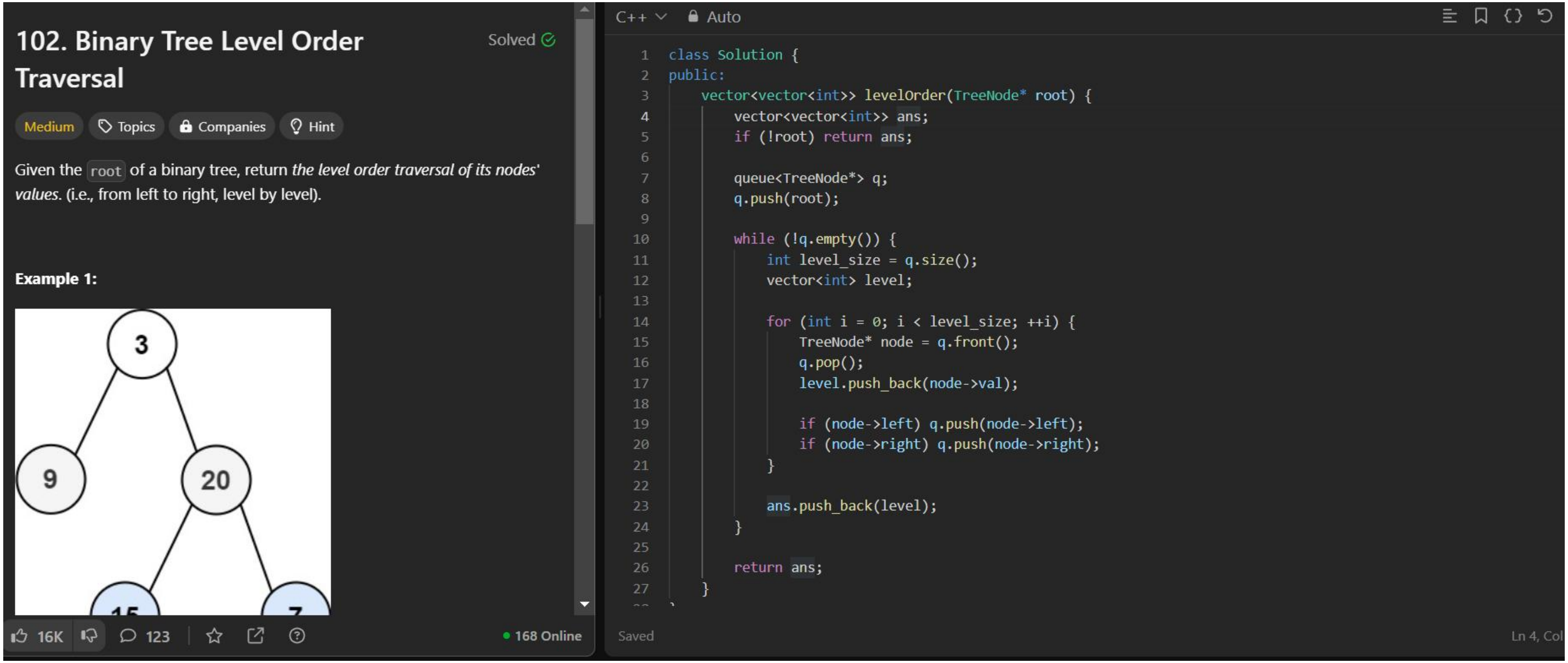
Example 1:

Input: `root = [1,null,2,3]`

Output: `[1,3,2]`

Explanation:

```cpp
class Solution {
public:
vector<int>ans;


        vector<int> inorderTraversal(TreeNode* root) {
        if(root==NULL) return ans;

        inorderTraversal(root->left);
         ans.push_back(root->val);
        inorderTraversal(root->right);


        return ans;
      }
};
```



102. Binary Tree Level Order Traversal

Solved

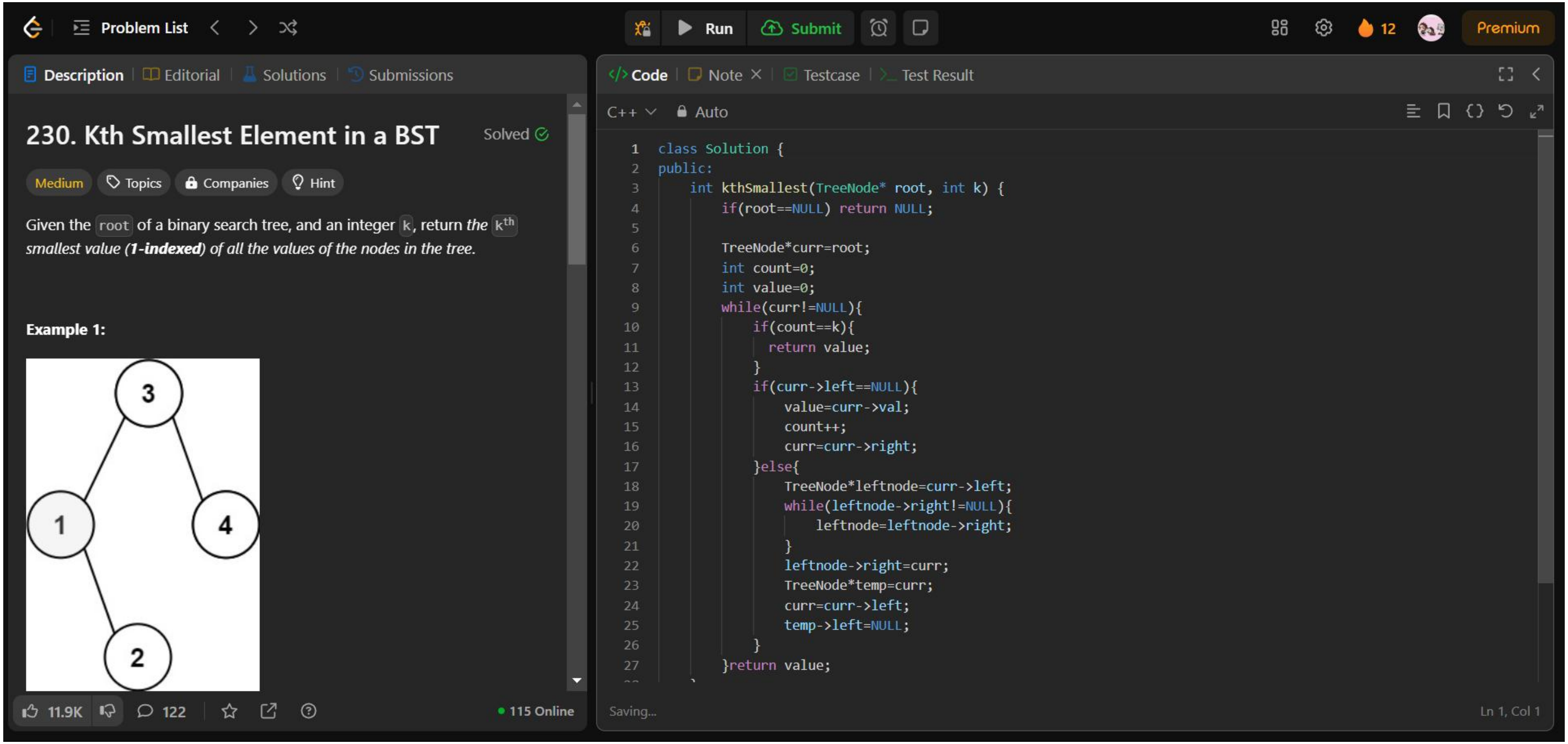
Given the `root` of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

Example 1:

```cpp
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        if (!root) return ans;

        queue<TreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            int level_size = q.size();
            vector<int> level;

            for (int i = 0; i < level_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);
            }

            ans.push_back(level);
        }

        return ans;
    }
```

Saved

Ln 4, Col

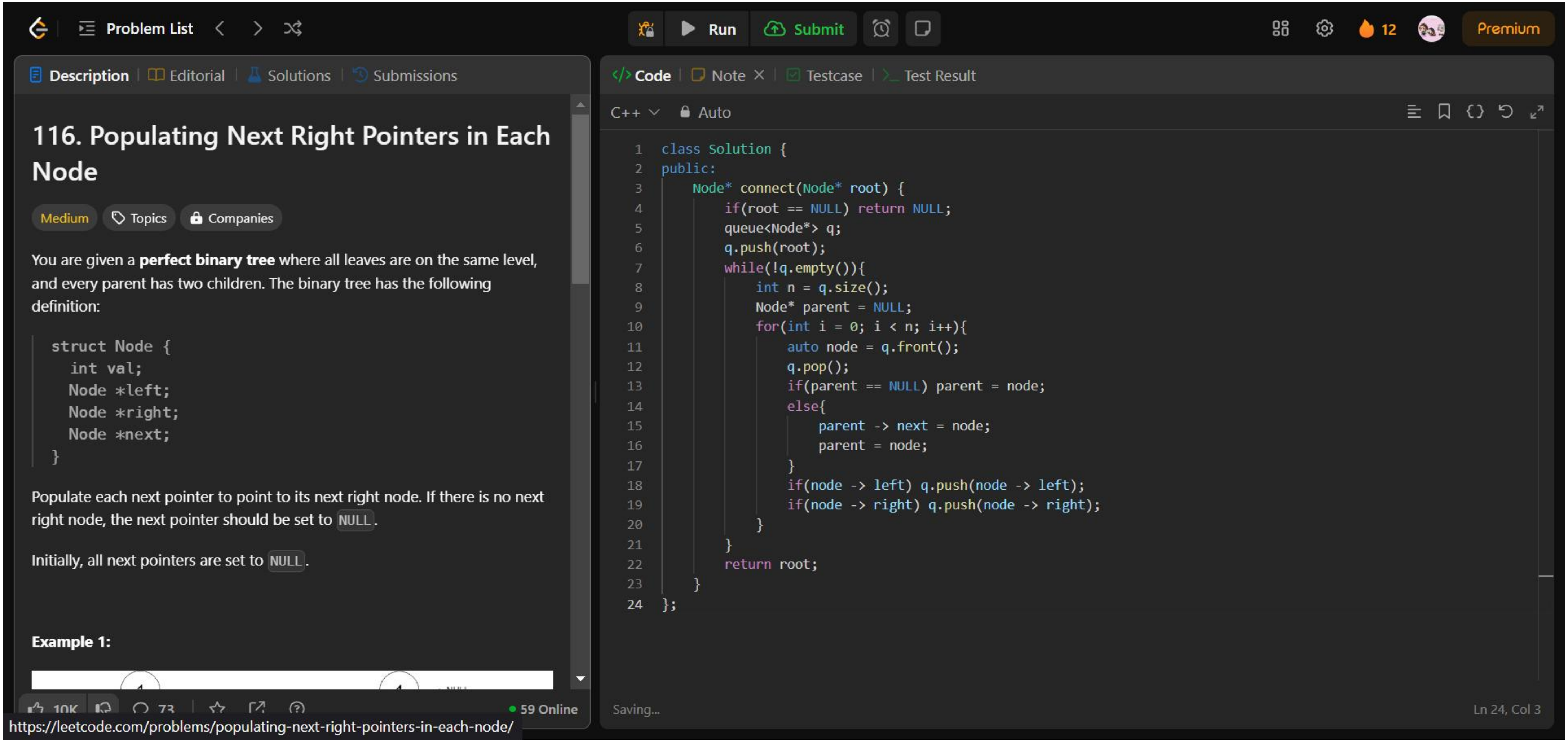


230. Kth Smallest Element in a BST

Given the `root` of a binary search tree, and an integer `k`, return *the* k^{th} *smallest value (**1-indexed**) of all the values of the nodes in the tree.*

Example 1:

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        if(root==NULL) return NULL;

        TreeNode*curr=root;
        int count=0;
        int value=0;
        while(curr!=NULL){
            if(count==k){
              return value;
            }
            if(curr->left==NULL){
                value=curr->val;
                count++;
                curr=curr->right;
            }else{
                TreeNode*leftnode=curr->left;
                while(leftnode->right!=NULL){
                    leftnode=leftnode->right;
                }
                leftnode->right=curr;
                TreeNode*temp=curr;
                curr=curr->left;
                temp->left=NULL;
            }
        }return value;
```



116. Populating Next Right Pointers in Each Node

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

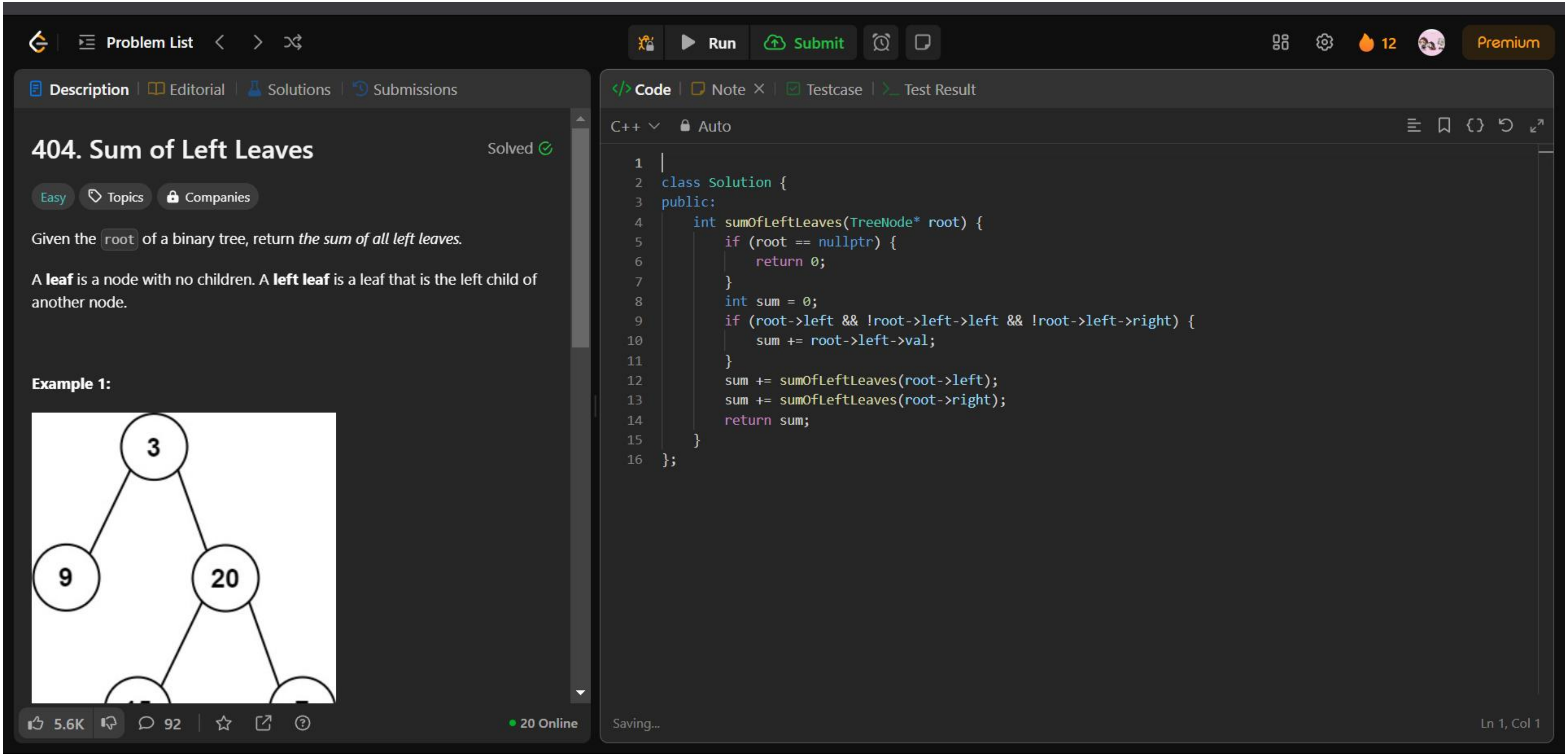
Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Example 1:

```cpp
class Solution {
public:
    Node* connect(Node* root) {
        if(root == NULL) return NULL;
        queue<Node*> q;
        q.push(root);
        while(!q.empty()){
            int n = q.size();
            Node* parent = NULL;
            for(int i = 0; i < n; i++){
                auto node = q.front();
                q.pop();
                if(parent == NULL) parent = node;
                else{
                    parent -> next = node;
                    parent = node;
                }
                if(node -> left) q.push(node -> left);
                if(node -> right) q.push(node -> right);
            }
        }
        return root;
    }
};
```

https://leetcode.com/problems/populating-next-right-pointers-in-each-node/



404. Sum of Left Leaves

Given the `root` of a binary tree, return *the sum of all left leaves.*

A **leaf** is a node with no children. A **left leaf** is a leaf that is the left child of another node.

Example 1:

```cpp

class Solution {
public:
    int sumOfLeftLeaves(TreeNode* root) {
        if (root == nullptr) {
            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;
    }
};
```