CU
CHANDIGARH UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Experiment 6

Student Name: Payal Singh **UID:22BET10347**
Branch: BE-IT **Section/Group: IOT-702(A)**
Semester: 6 **Date of Performance:06/03/25**
Subject Name: AP LAB-II **Subject Code: 22ITP-351**

PROBLEM 1:

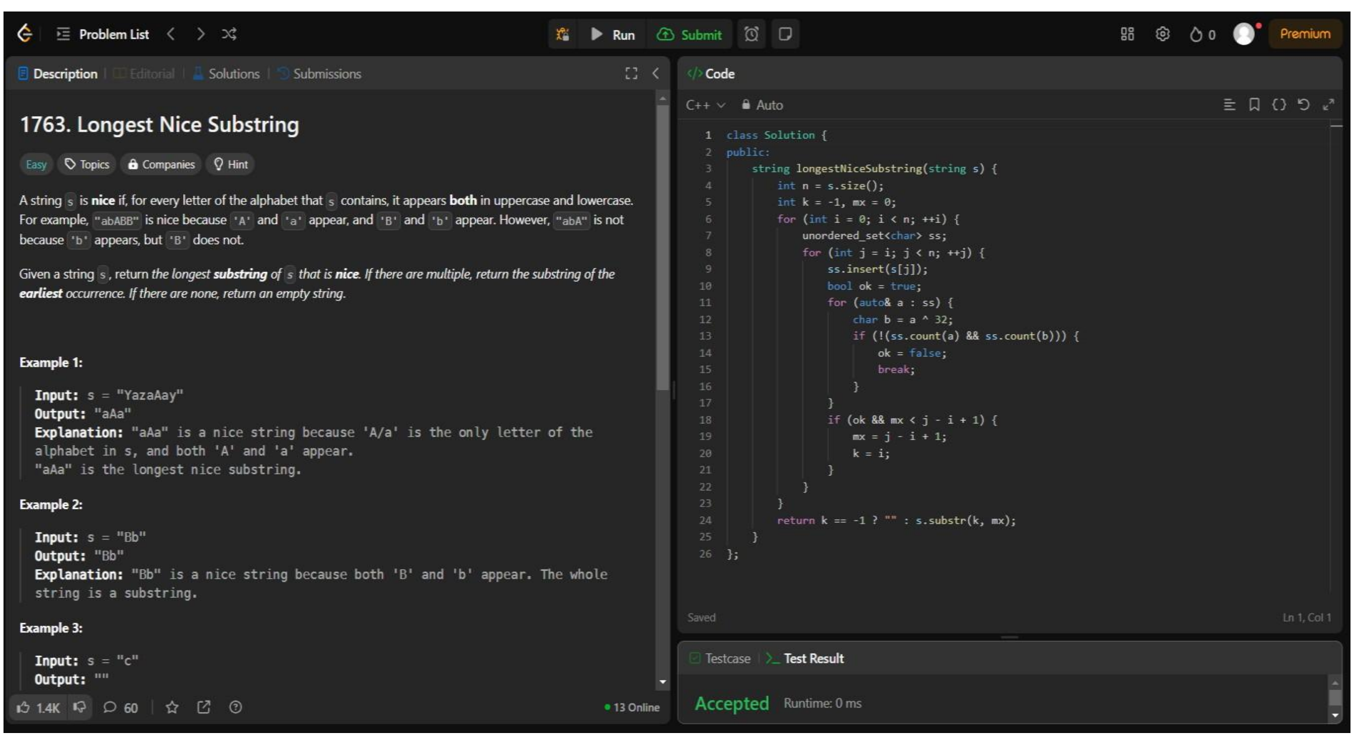
Aim:

Given the root of a binary tree, return *its maximum depth*.
A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Code:

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

Output:



PROBLEM 2:

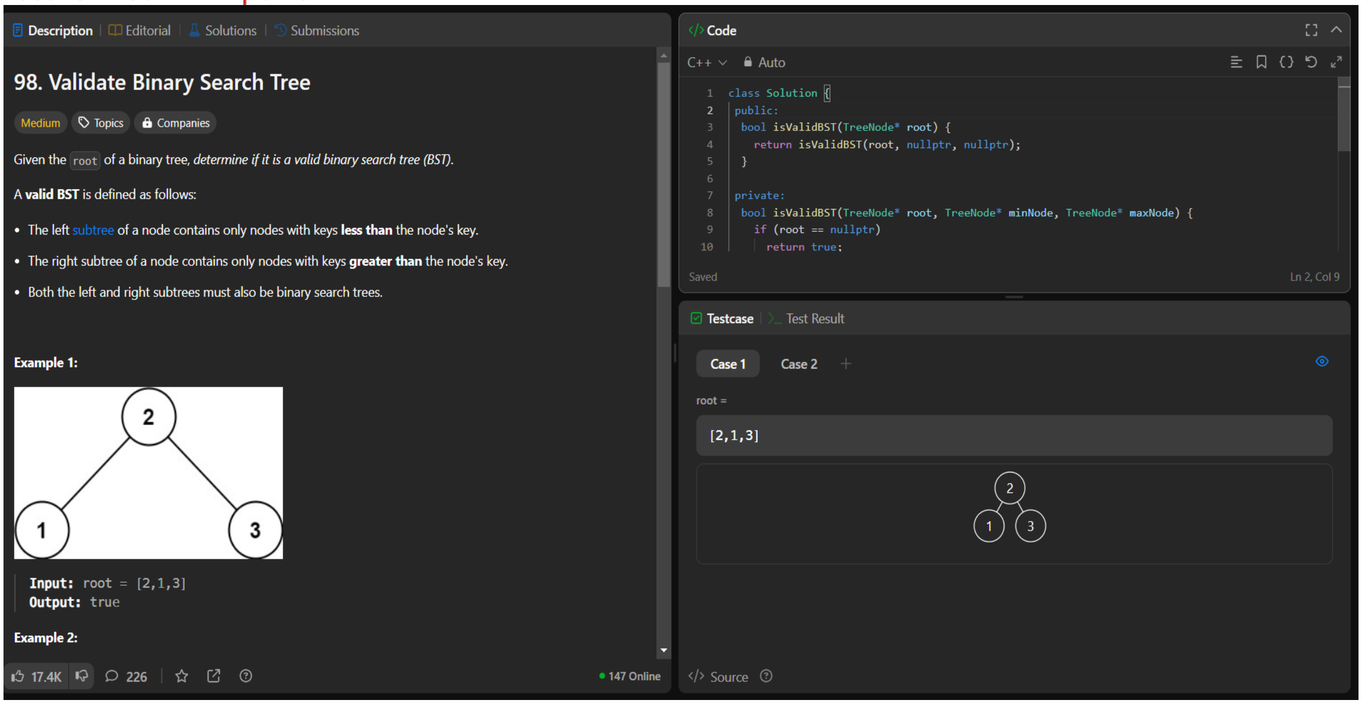
Aim: Given the root of a binary tree, *determine if it is a valid binary search tree (BST).*
A valid BST is defined as follows:

- 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.

Code:

```
class Solution {
 public:
  bool isValidBST(TreeNode* root) {
    return isValidBST(root, nullptr, nullptr);
  }
 private:
  bool isValidBST(TreeNode* root, TreeNode* minNode, TreeNode* maxNode) {
    if (root == nullptr)
      return true;
    if (minNode && root->val <= minNode->val)
      return false;
    if (maxNode && root->val >= maxNode->val)
      return false;

    return isValidBST(root->left, minNode, root) &&
           isValidBST(root->right, root, maxNode);
  }
};
```

Output:



PROBLEM 3:

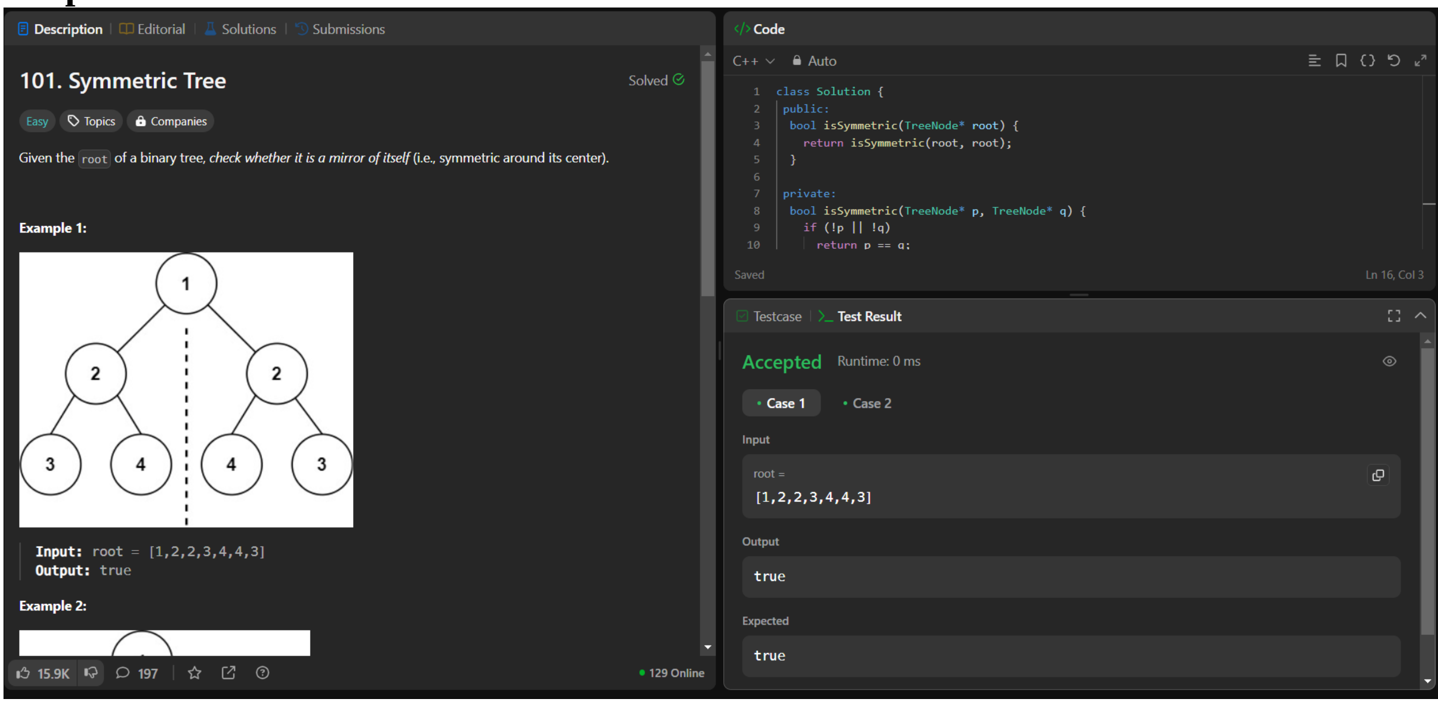
Aim: Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

Code:

```
class Solution {

public:

bool isSymmetric(TreeNode* root) {

    return isSymmetric(root, root);

  }



 private:

  bool isSymmetric(TreeNode* p, TreeNode* q) {

    if (!p || !q)

      return p == q;



    return p->val == q->val &&                //
```

```
        isSymmetric(p->left, q->right) &&  //

        isSymmetric(p->right, q->left);

  }

};
```

Output:



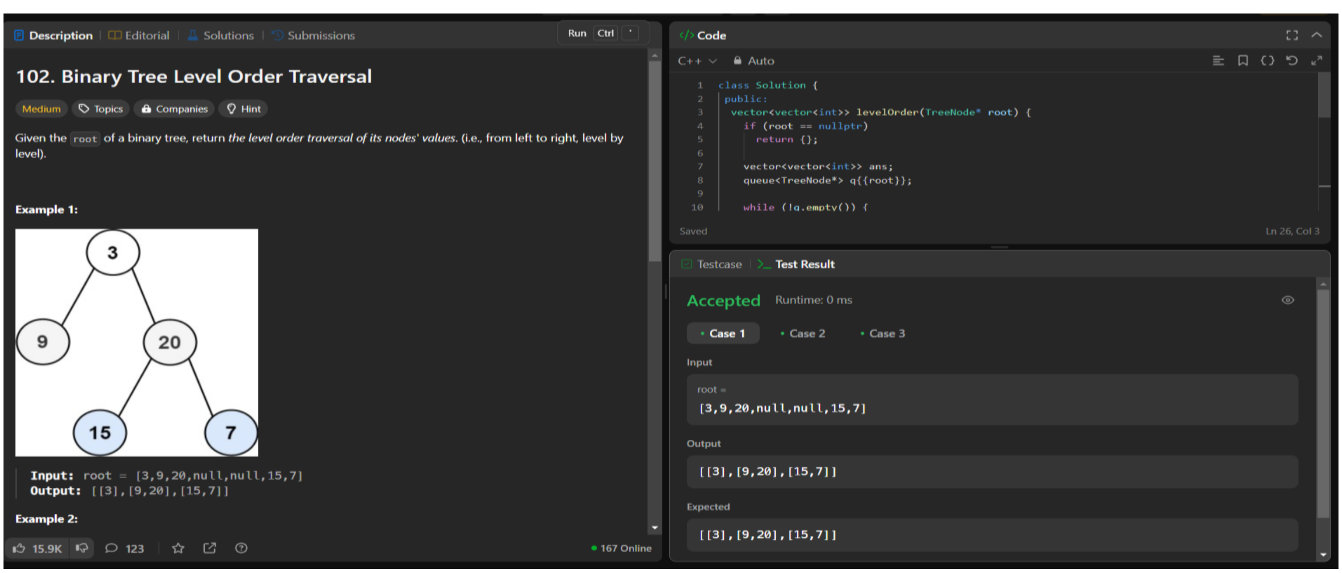
PROBLEM 4:

Aim: 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).

Code:

```
class Solution {
 public:
  vector<vector<int>> levelOrder(TreeNode* root) {
    if (root == nullptr)
      return {};

    vector<vector<int>> ans;
    queue<TreeNode*> q{{root}};

    while (!q.empty()) {
      vector<int> currLevel;
      for (int sz = q.size(); sz > 0; --sz) {
        TreeNode* node = q.front();
        q.pop();
        currLevel.push_back(node->val);
        if (node->left)
          q.push(node->left);
        if (node->right)
          q.push(node->right);
      }
      ans.push_back(currLevel);
    }

    return ans;
  }
};
```

OUTPUT:



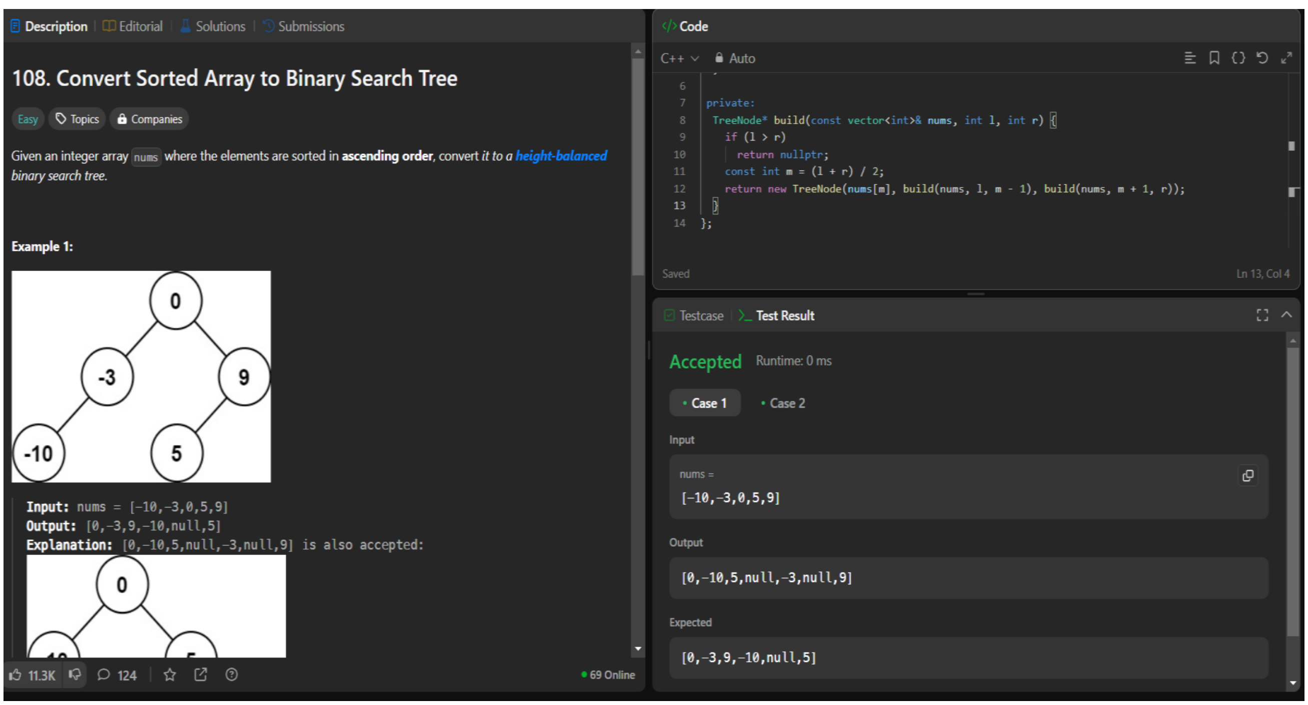
PROBLEM 5:

Aim: Given an integer array nums where the elements are sorted in ascending order, convert *it to a height-balanced binary search tree*.

Code:

```
class Solution {
 public:
  TreeNode* sortedArrayToBST(vector<int>& nums) {
    return build(nums, 0, nums.size() - 1);
  }

 private:
  TreeNode* build(const vector<int>& nums, int l, int r) {
    if (l > r)
      return nullptr;
    const int m = (l + r) / 2;
    return new TreeNode(nums[m], build(nums, l, m - 1), build(nums, m + 1, r));
  }
};
```

Output:



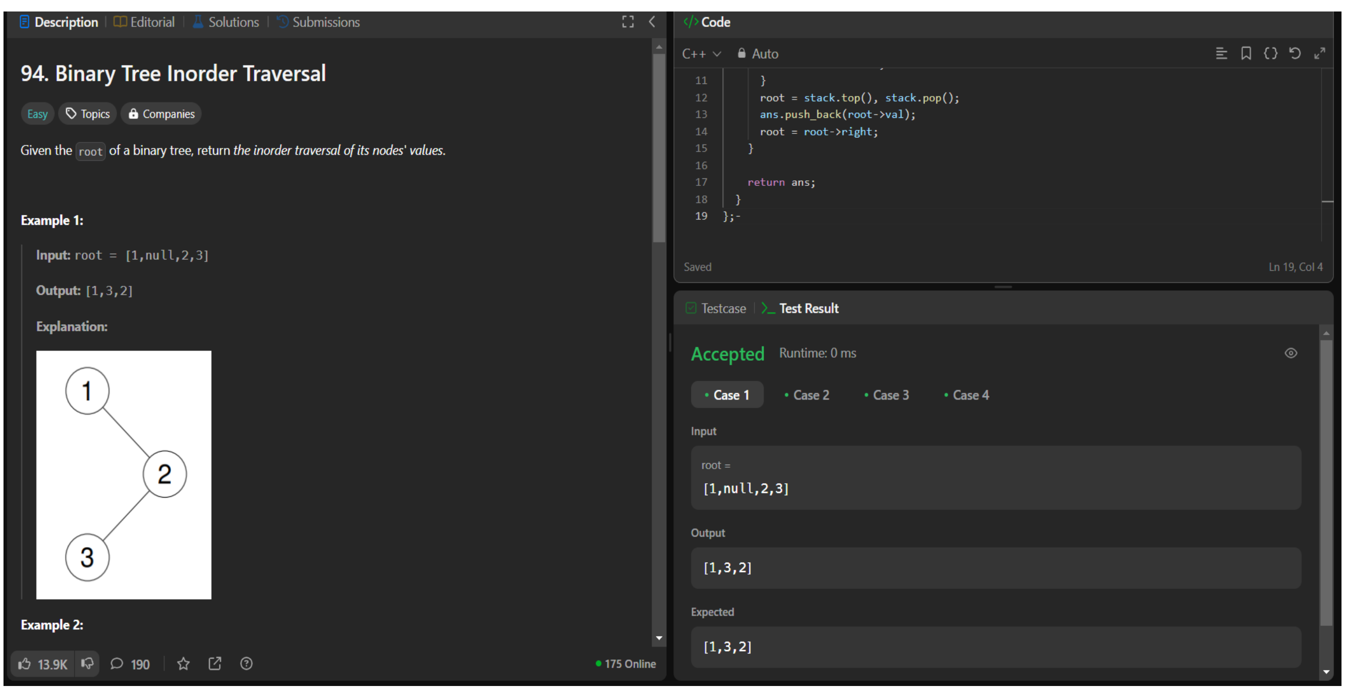
PROBLEM 6:

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

Code:

```
class Solution {
 public:
  vector<int> inorderTraversal(TreeNode* root) {
    vector<int> ans;
    stack<TreeNode*> stack;

    while (root != nullptr || !stack.empty()) {
      while (root != nullptr) {
        stack.push(root);
        root = root->left;
      }
      root = stack.top(), stack.pop();
      ans.push_back(root->val);
      root = root->right;
    }

    return ans;
  }
};
```

Output:



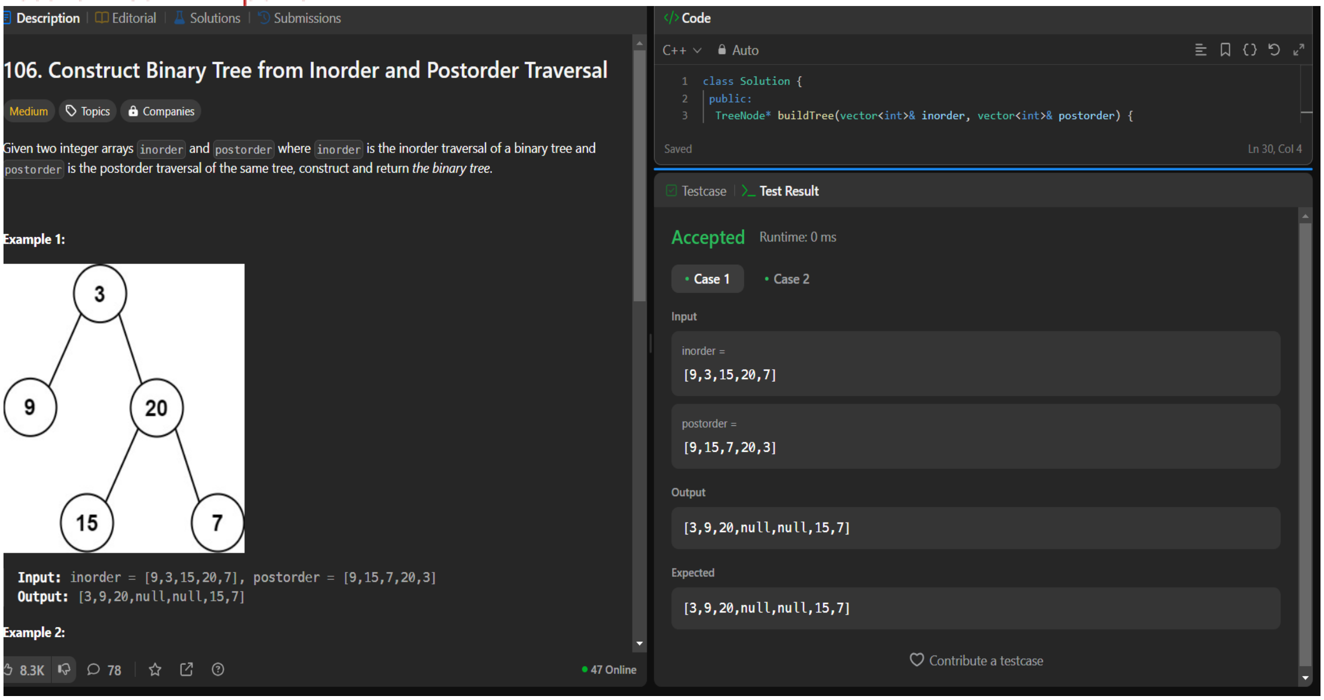
PROBLEM 7:

Aim: Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return *the binary tree*.

Code:

```
class Solution {
 public:
  TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
    unordered_map<int, int> inToIndex;

    for (int i = 0; i < inorder.size(); ++i)
      inToIndex[inorder[i]] = i;

    return build(inorder, 0, inorder.size() - 1, postorder, 0,
                 postorder.size() - 1, inToIndex);
  }

 private:
  TreeNode* build(const vector<int>& inorder, int inStart, int inEnd,
                  const vector<int>& postorder, int postStart, int postEnd,
                  const unordered_map<int, int>& inToIndex) {
    if (inStart > inEnd)
      return nullptr;

    const int rootVal = postorder[postEnd];
    const int rootInIndex = inToIndex.at(rootVal);
    const int leftSize = rootInIndex - inStart;

    TreeNode* root = new TreeNode(rootVal);
    root->left = build(inorder, inStart, rootInIndex - 1, postorder, postStart,
                       postStart + leftSize - 1, inToIndex);
    root->right = build(inorder, rootInIndex + 1, inEnd, postorder,
                        postStart + leftSize, postEnd - 1, inToIndex);
    return root;
  }
};
```

Output:



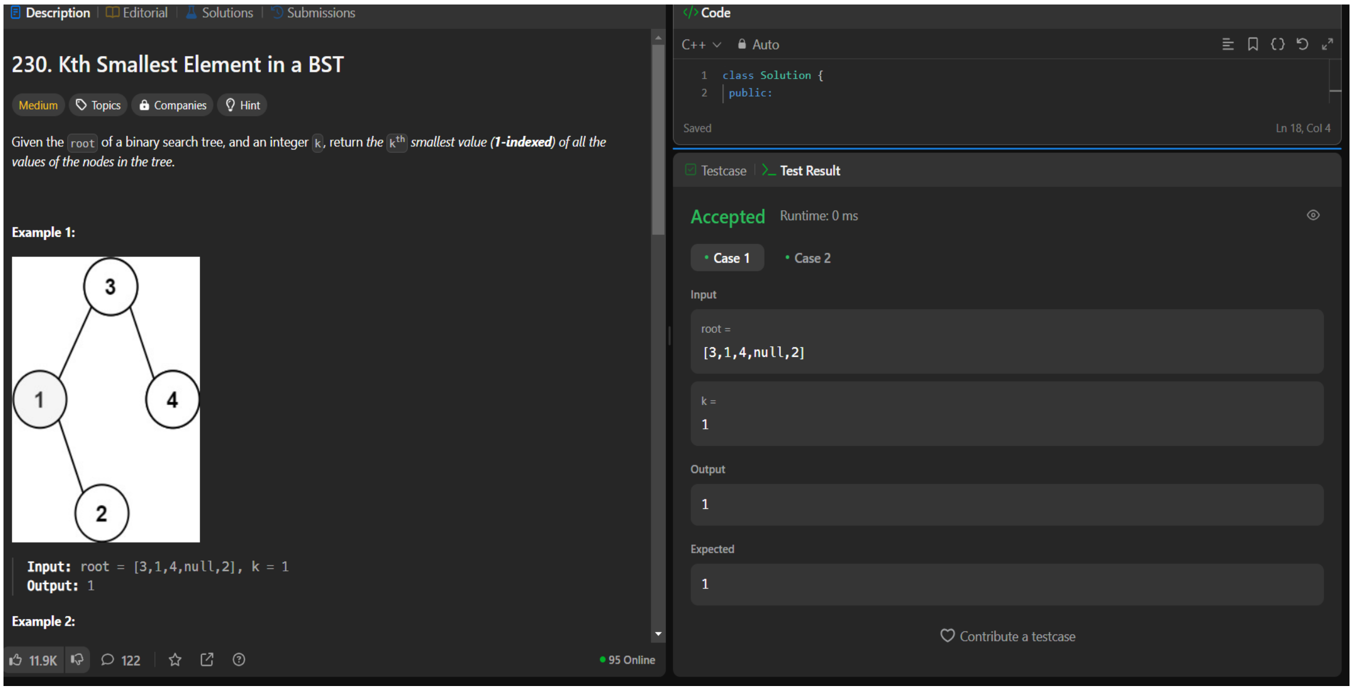
PROBLEM 9:

Aim: 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*.

Code:

```
class Solution {
 public:
  int kthSmallest(TreeNode* root, int k) {
    const int leftCount = countNodes(root->left);

    if (leftCount == k - 1)
      return root->val;
    if (leftCount >= k)
      return kthSmallest(root->left, k);
    return kthSmallest(root->right, k - 1 - leftCount);  // leftCount < k
  }

 private:
  int countNodes(TreeNode* root) {
    if (root == nullptr)
      return 0;
    return 1 + countNodes(root->left) + countNodes(root->right);
  }
};
```

Output:



PROBLEM 9:

Aim: 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.

Code:

```
class Solution {
 public:
  Node* connect(Node* root) {
    if (root == nullptr)
      return nullptr;
    connectTwoNodes(root->left, root->right);
    return root;
  }

 private:
  void connectTwoNodes(Node* p, Node* q) {
    if (p == nullptr)
```

```
      return;
    p->next = q;
    connectTwoNodes(p->left, p->right);
    connectTwoNodes(q->left, q->right);
    connectTwoNodes(p->right, q->left);
  }
};
```

Output: