# ASSIGNMENT 3

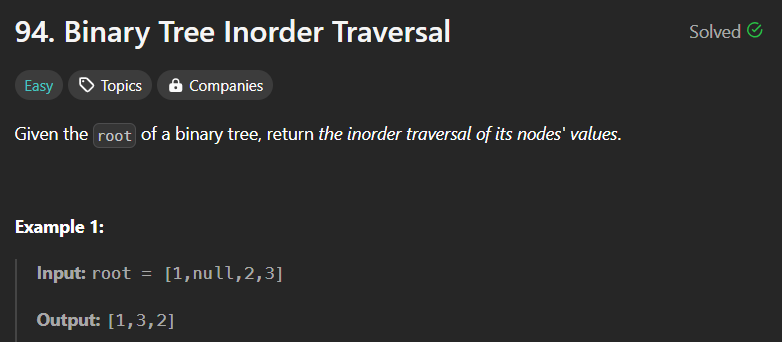
**Student Name: Mehak Chauhan UID: 22BCS12675**

**Branch: CSE Section: 22BCS\_IOT\_605 B**

**Semester: 6th DOP:14-02-2025**

**Subject: Advanced Programming Lab-II Subject Code: 22CSP-351**

**Question 1**

****

**Code:**

class Solution {

public:

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

        vector<int> res;

        if(root== NULL) return res;

        helper(root,res);

        return res;

    }

    void helper(TreeNode\* root, vector<int> &res)

    {

        if(root->left)

        {

         helper(root->left,res);

        }

        res.push\_back(root->val);

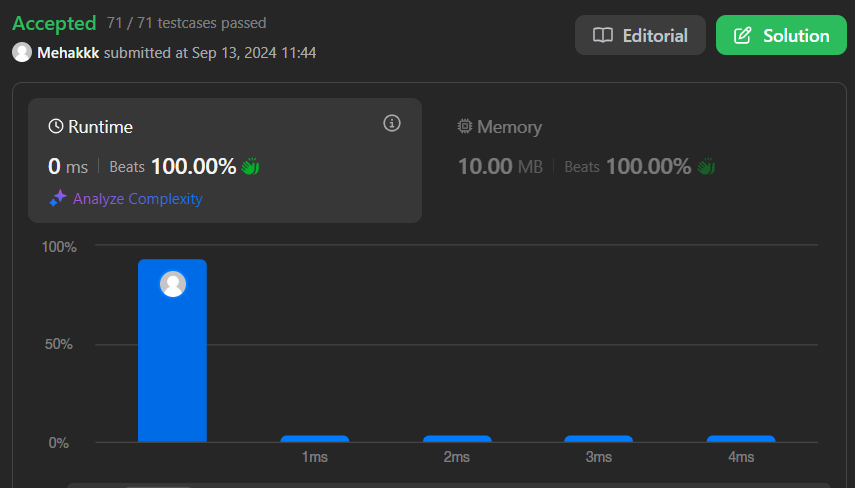
        if(root->right)

          helper(root->right,res);

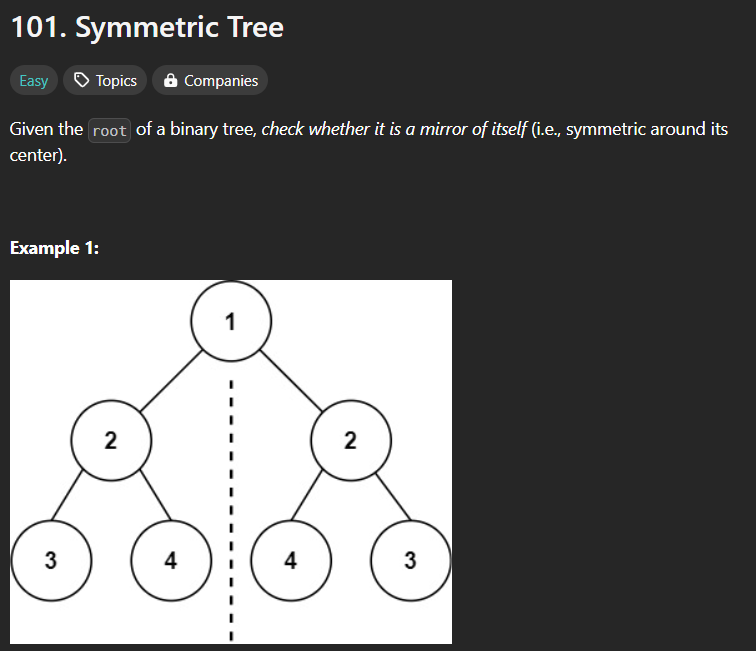
    }

};

**Output:**

****

**Question 2**

****

**Code:**

class Solution {

public:

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);

}

public:

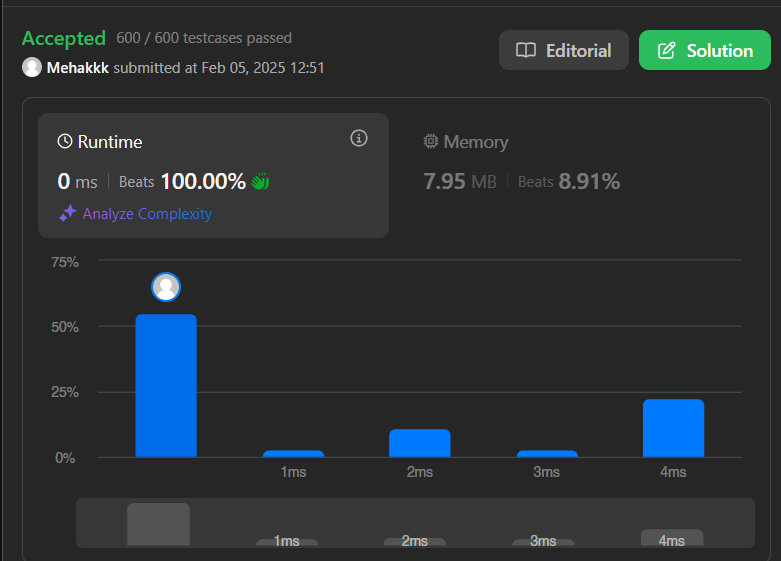
    bool isSymmetric(TreeNode\* root) {

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

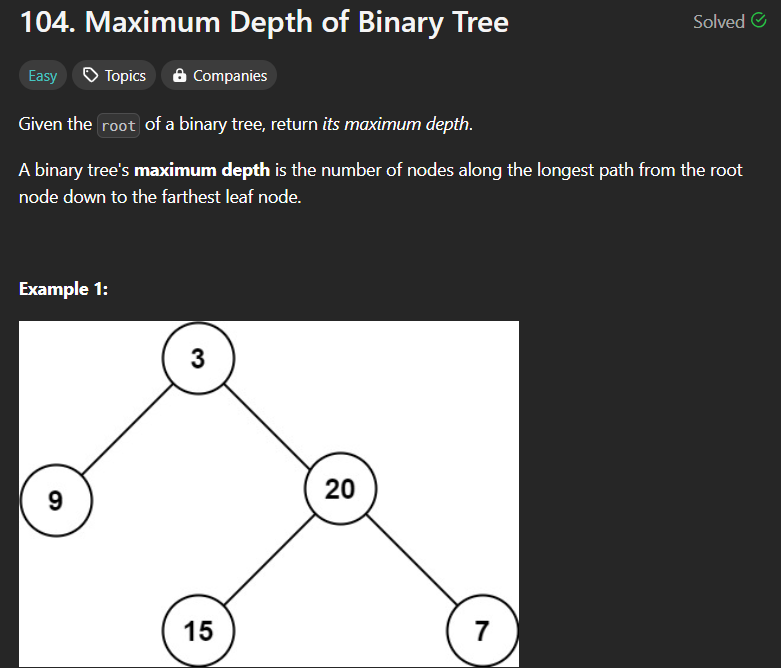
    }

};

**Output:**

****

**Question 3**

****

**Code:**

class Solution {

public:

    int maxDepth(TreeNode\* root) {

        if(root==NULL){

            return 0;

        }

       int left\_height = maxDepth(root->left);

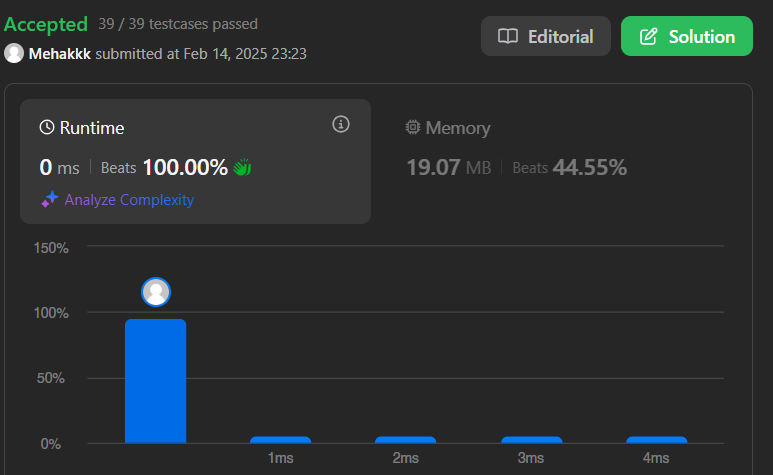
        int right\_height= maxDepth(root->right);

        return max(left\_height,right\_height)+1;

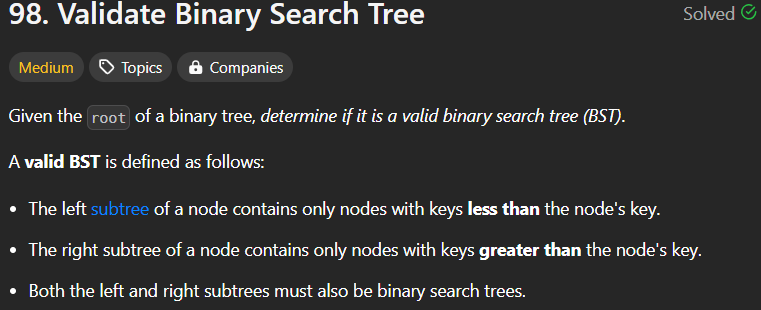
    }

};

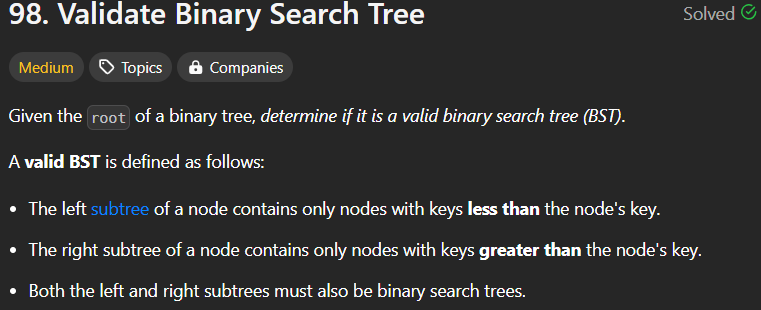
**Output:**

****

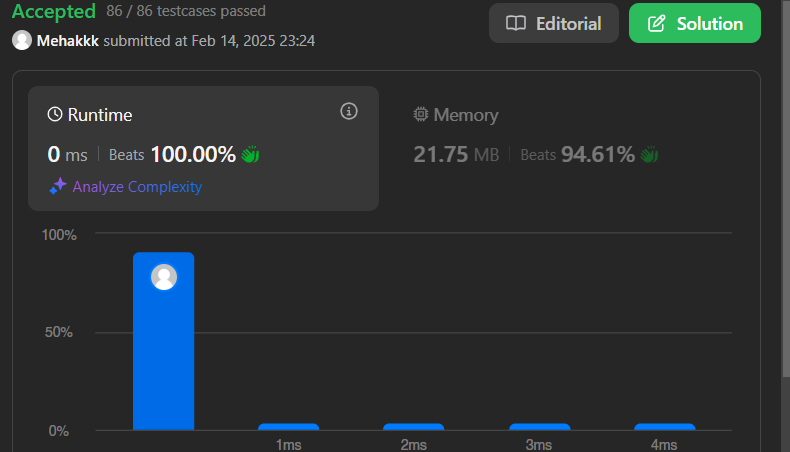
**Question 4**

****

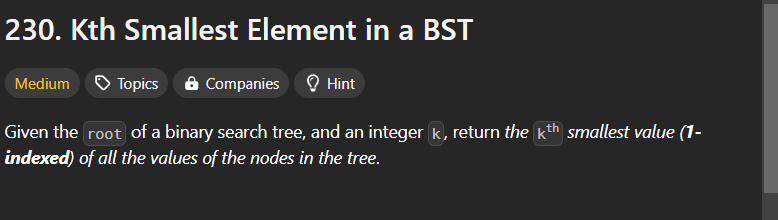
**Code:**

****

**Output:**

****

**Question 5**

****

**Code:**

#include <bits/stdc++.h>

using namespace std;

class TreeNode {

public:

    int val;

    TreeNode \*left, \*right;

    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

class Solution {

public:

    int count = 0; // Counter for visited nodes

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

        TreeNode\* result = helper(root, k);

        return result ? result->val : 0; // Return value or 0 if not found

    }

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

        if (root == nullptr) return nullptr;

        // Traverse left subtree

        TreeNode\* left = helper(root->left, k);

        if (left != nullptr) return left; // If found in left subtree

        count++; // Increment count for current node

        if (count == k) return root; // Found k-th smallest

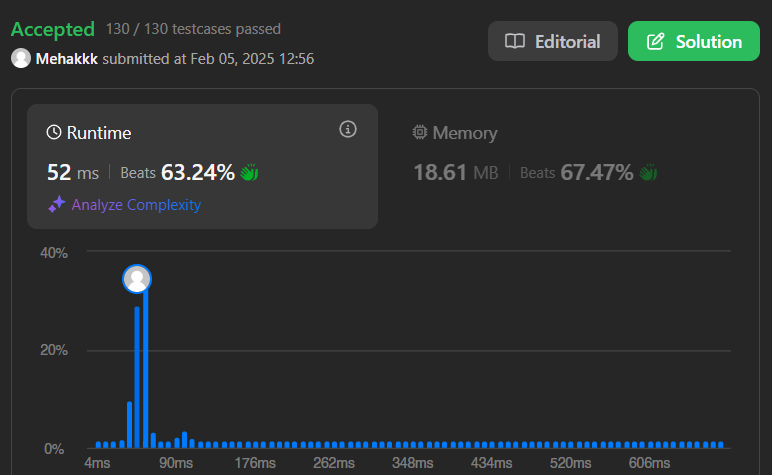
        // Traverse right subtree

        return helper(root->right, k);

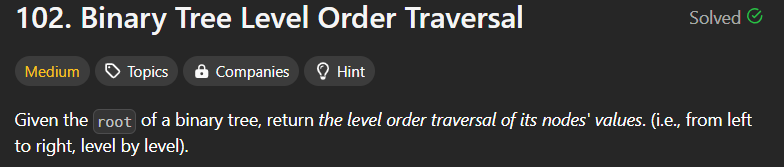
    }

};

**Output:**

****

**Question 6**

****

**Code**

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 size= q.size();

            vector <int> level;

            for(int i=0;i<size;i++){

                TreeNode\* node = q.front();

                q.pop();

                if(node->left!=NULL){

                    q.push(node->left);

                }

                if(node->right!=NULL){

                    q.push(node->right);

                }

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

            }

            ans.push\_back(level);

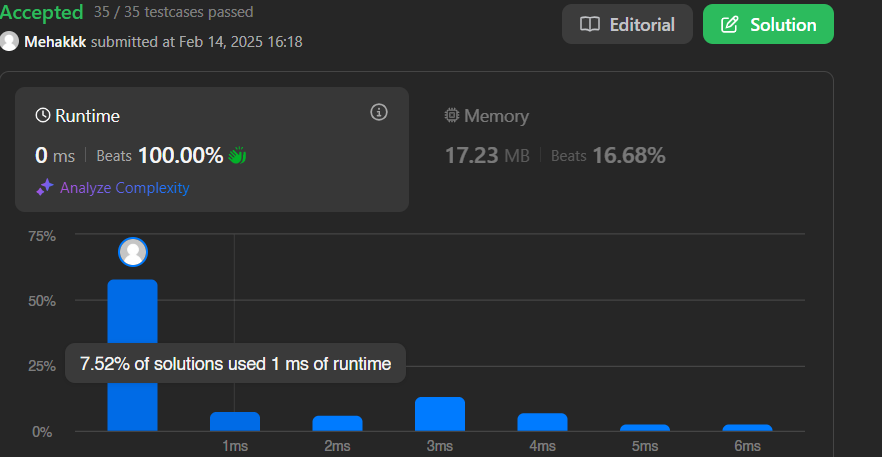
        }

        return ans;

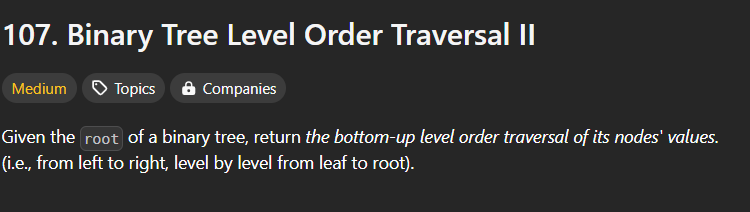
    }

};

**Output**

****

**Question 7**

****

**Code:**

class Solution {

public:

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

        if (!root) return {};

        vector<vector<int>> 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);

        }

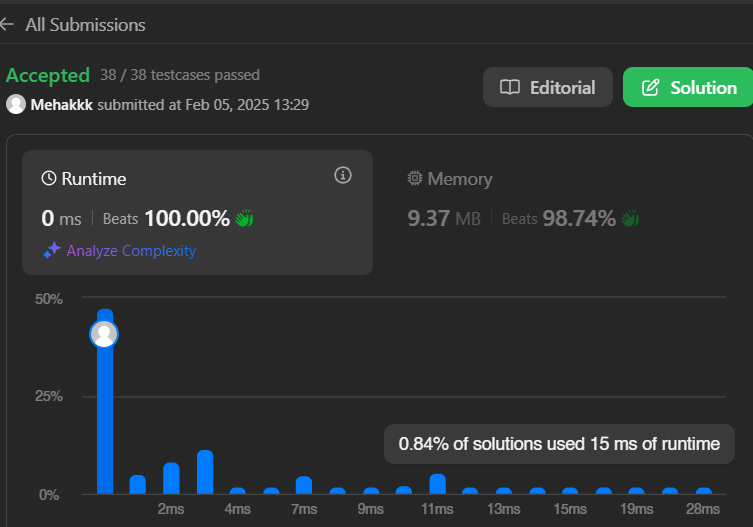
        reverse(result.begin(), result.end());

        return result;

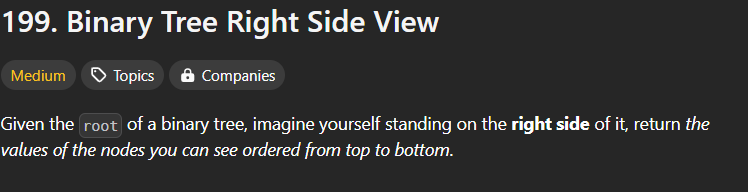
    }

};

**Output:**

****

**Question 8**

****

**Code:**

class Solution {

 public:

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

    if (root == nullptr)

      return {};

    vector<int> ans;

    queue<TreeNode\*> q{{root}};

    while (!q.empty()) {

      const int size = q.size();

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

        TreeNode\* node = q.front();

        q.pop();

        if (i == size - 1)

          ans.push\_back(node->val);

        if (node->left)

          q.push(node->left);

        if (node->right)

          q.push(node->right);

      }

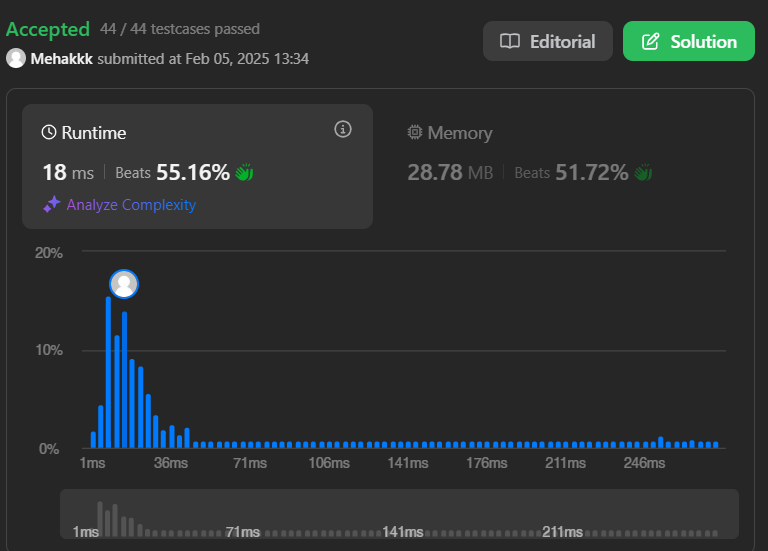
    }

    return ans;

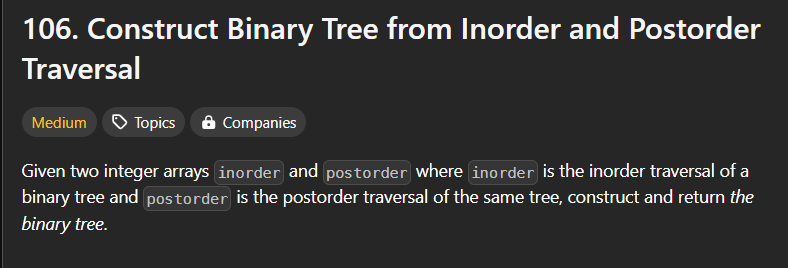
  }

};

**Output:**

****

**Question 9**

****

**Code:**

#include <bits/stdc++.h>

using namespace std;

class TreeNode {

public:

    int val;

    TreeNode \*left, \*right;

    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

class Solution {

public:

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

        unordered\_map<int, int> rec;

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

            rec[inorder[i]] = i;

        }

        return helper(inorder, postorder, 0, inorder.size() - 1, 0, postorder.size() - 1, rec);

    }

    TreeNode\* helper(vector<int>& inorder, vector<int>& postorder,

                     int inStart, int inEnd,

                     int postStart, int postEnd,

                     unordered\_map<int, int>& rec) {

        if (inStart > inEnd || postStart > postEnd) return nullptr;

        int val = postorder[postEnd];

        TreeNode\* root = new TreeNode(val);

        int idx = rec[val];

        int leftSubtreeSize = idx - inStart;

        root->left = helper(inorder, postorder,

                             inStart, idx - 1,

                             postStart, postStart + leftSubtreeSize - 1,

                             rec);

        root->right = helper(inorder, postorder,

                              idx + 1, inEnd,

                              postStart + leftSubtreeSize, postEnd - 1,

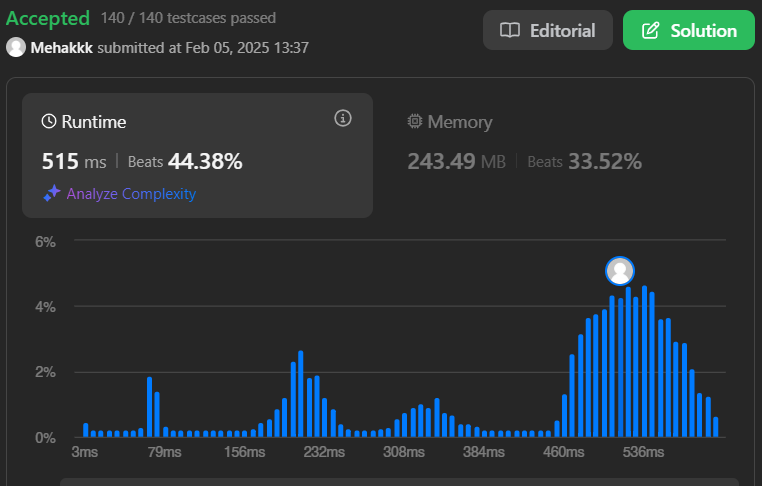
                              rec);

        return root;

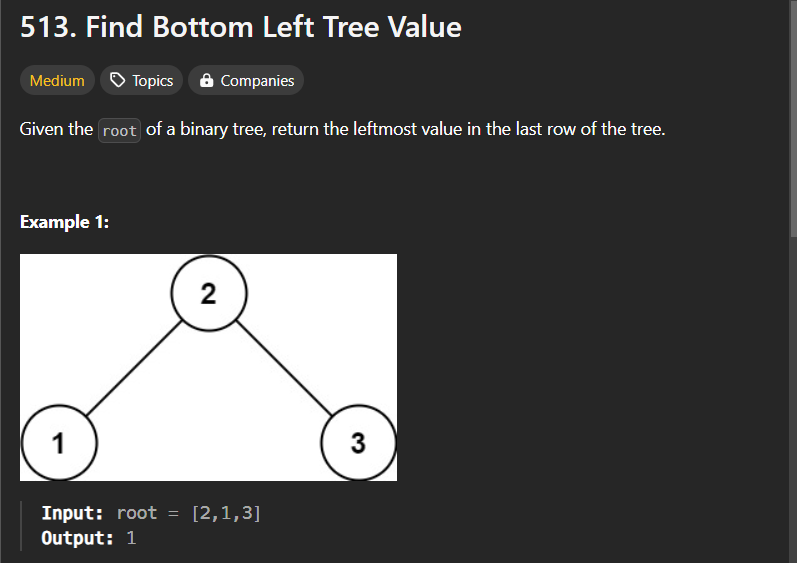
    }

};

**Output:**

****

**Question 10**

****

**Code:**

class Solution {

public:

    int findBottomLeftValue(TreeNode\* root) {

        queue<TreeNode\*> q;

        q.push(root);

        int leftmost\_value;

        while (!q.empty()) {

            TreeNode\* node = q.front();

            q.pop();

            leftmost\_value = node->val;

            if (node->right) {

                q.push(node->right);

            }

            if (node->left) {

                q.push(node->left);

            }

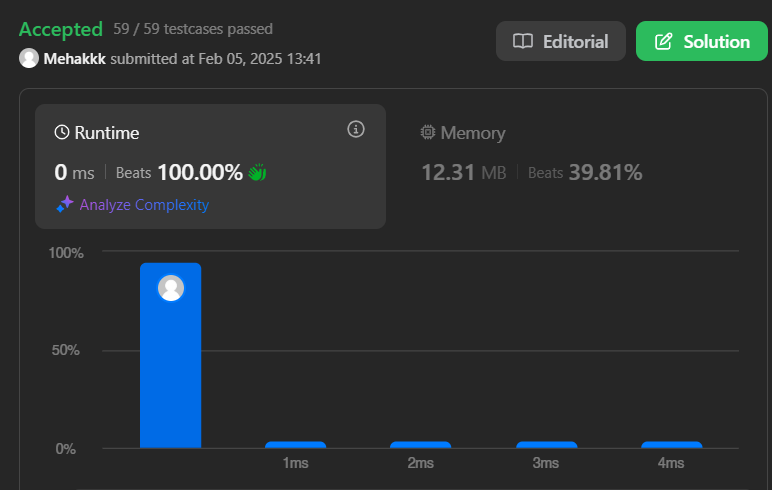
        }

        return leftmost\_value;

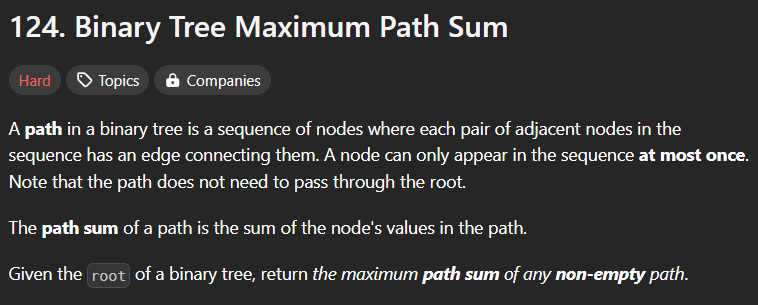
    }

};

**Output:**

****

**Question 11**

****

**Code:**

class Solution {

 public:

  int maxPathSum(TreeNode\* root) {

    int ans = INT\_MIN;

    maxPathSumDownFrom(root, ans);

    return ans;

  }

 private:

  int maxPathSumDownFrom(TreeNode\* root, int& ans) {

    if (root == nullptr)

      return 0;

    const int l = max(0, maxPathSumDownFrom(root->left, ans));

    const int r = max(0, maxPathSumDownFrom(root->right, ans));

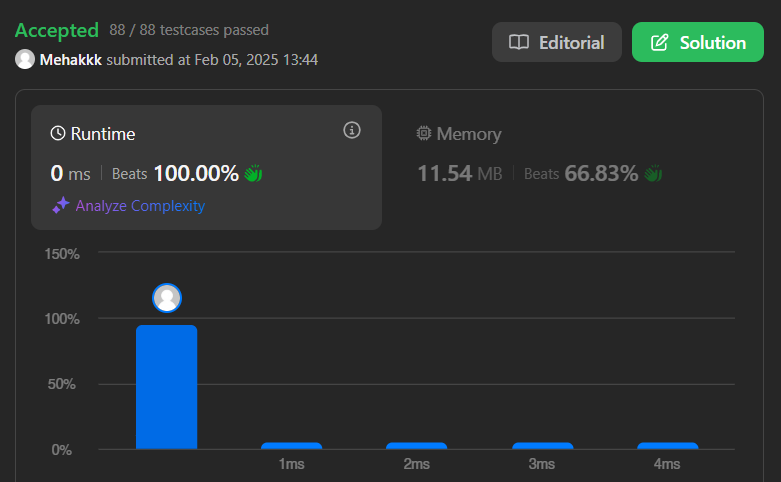
    ans = max(ans, root->val + l + r);

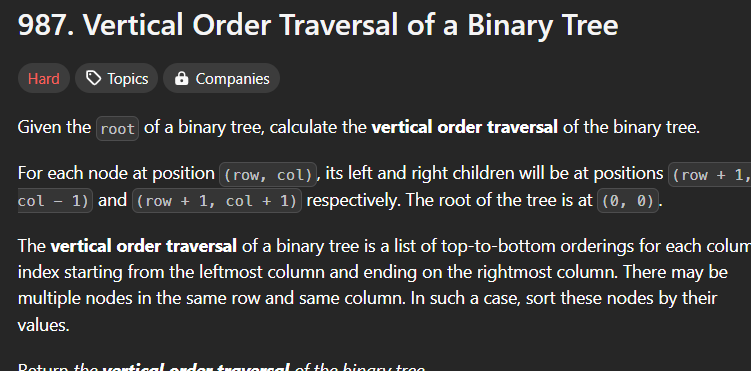
    return root->val + max(l, r);

  }

};

**Output:**

****

**Question 12**

**Code:**

class Solution {

public:

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

        vector<vector<int>> out;

        map<int, vector<int>> final\_mp;

        queue<pair<int, TreeNode\*>> q;

        q.push({0, root});

        while (!q.empty()) {

            int n = q.size();

            map<int, vector<int>> mp;

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

                auto it = q.front();

                int index = it.first;

                TreeNode\* node = it.second;

                mp[index].push\_back(node->val); // Add node value to the corresponding column in `mp`

                q.pop();

                if (node->left != NULL)

                    q.push({index - 1, node->left});

                if (node->right != NULL)

                    q.push({index + 1, node->right});

            }

            // Process nodes level by level

            for (auto it : mp) {

                sort(it.second.begin(), it.second.end()); // Sort nodes at the same column and level

                vector<int> temp = final\_mp[it.first];    // Get existing values for this column

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

                    temp.push\_back(it.second[i]);         // Append sorted nodes for this column

                }

                final\_mp[it.first] = temp;                // Update the column in the final map

            }

        }

        // Convert the map to a vector of vectors

        for (auto it : final\_mp) {

            out.push\_back(it.second);

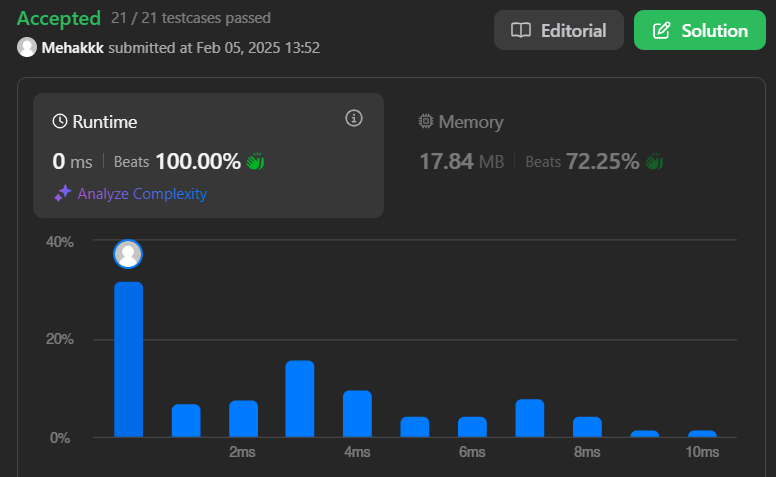
        }

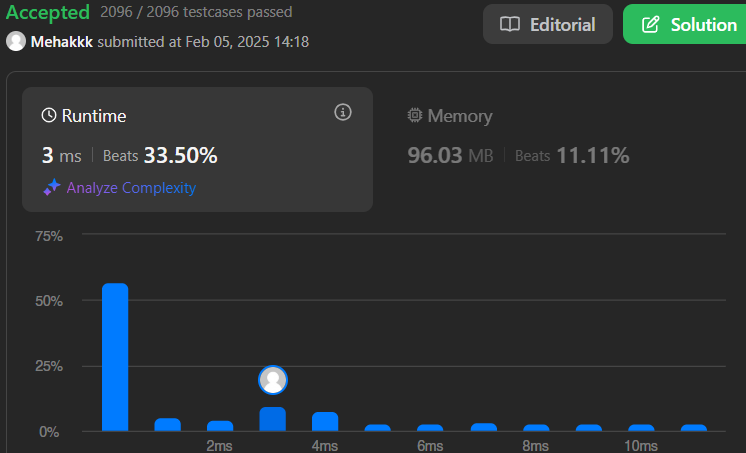
        return out;

    }

};

**Output:**

****

****