ASSIGNMENT-2

NAME: Sachin dadhwal SECTION: 605/B

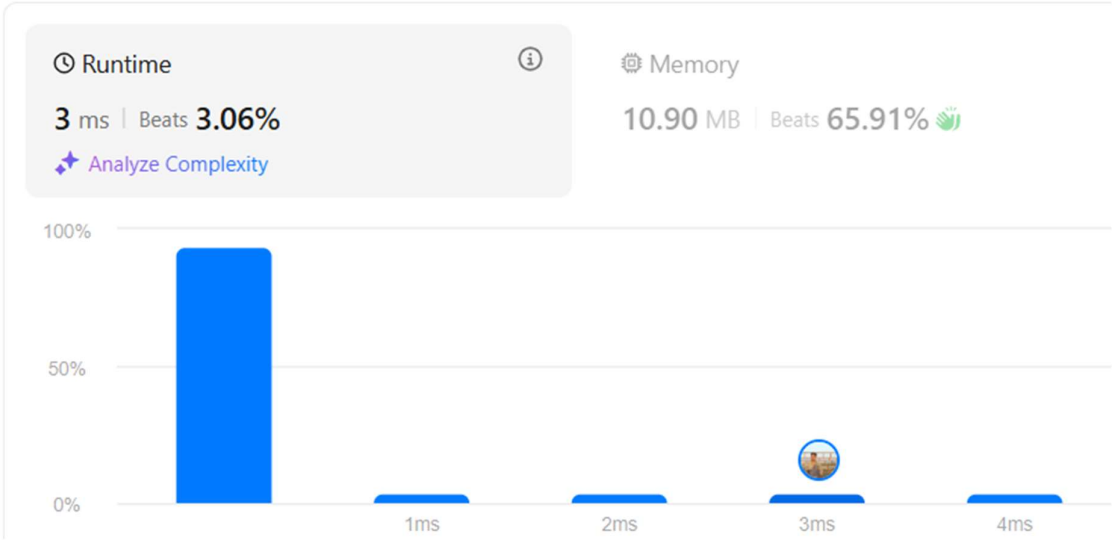
UID: 22BCS12149

1. 94.Binary Tree Inorder Traversal.
   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>in;
      TreeNode* curr=root;
      while(curr!=nullptr){
        if(curr->left==nullptr){
            in.push_back(curr->val);
            curr=curr->right;
        }
        else{
            TreeNode* prev=curr->left;
            while(prev->right && prev->right!=curr){
                prev=prev->right;
            }
            if(prev->right==nullptr){
                prev->right=curr;
                curr=curr->left;
            }
            else{
                prev->right=nullptr;
                in.push_back(curr->val);
                curr=curr->right;
            }
        }
      }
      return in;
    }
```

OUTPUT:



2. 101.Symmetric Tree.

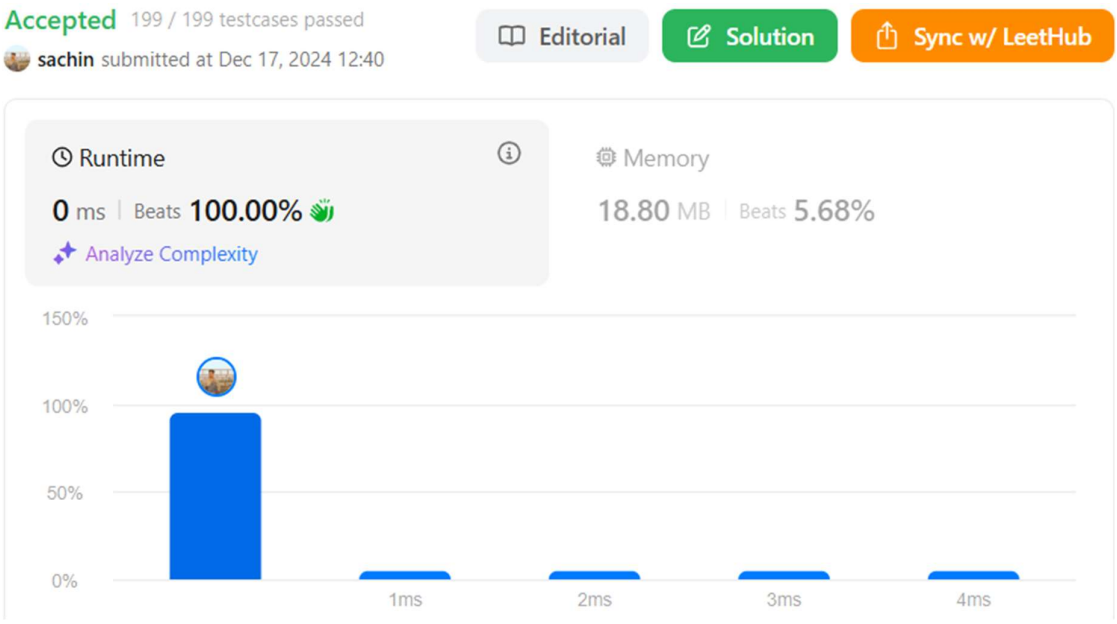
Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

CODE:

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        stack<TreeNode*>s1;
        stack<TreeNode*>s2;
        if(root==NULL){
            return true;
        }
        s1.push(root->left);
        s2.push(root->right);

        while(!s1.empty() && !s2.empty()){
            TreeNode* t1=s1.top();
            TreeNode* t2=s2.top();
```

```
            s1.pop();
            s2.pop();
            if(!t1 && !t2) continue;
            if( !t1 || !t2 || (t1->val!=t2->val)) return false;
                s1.push(t1->left);
                 s1.push(t1->right);
               s2.push(t2->right);
                s2.push(t2->left);
        return true;
};
```

OUTPUT:



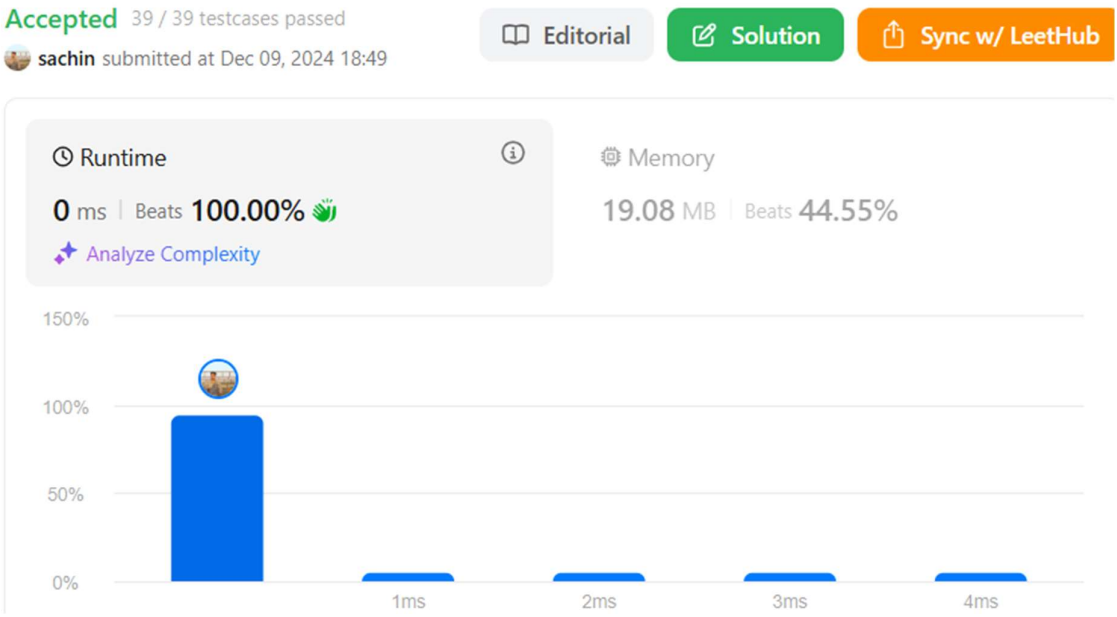
3. **104.[Maximum Depth of Binary Tree](#).**

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==NULL){
        return 0;
    }

    int l=maxDepth(root->left);
    int r=maxDepth(root->right);

      return 1+max(l,r);
    }
};
```

OUTPUT:

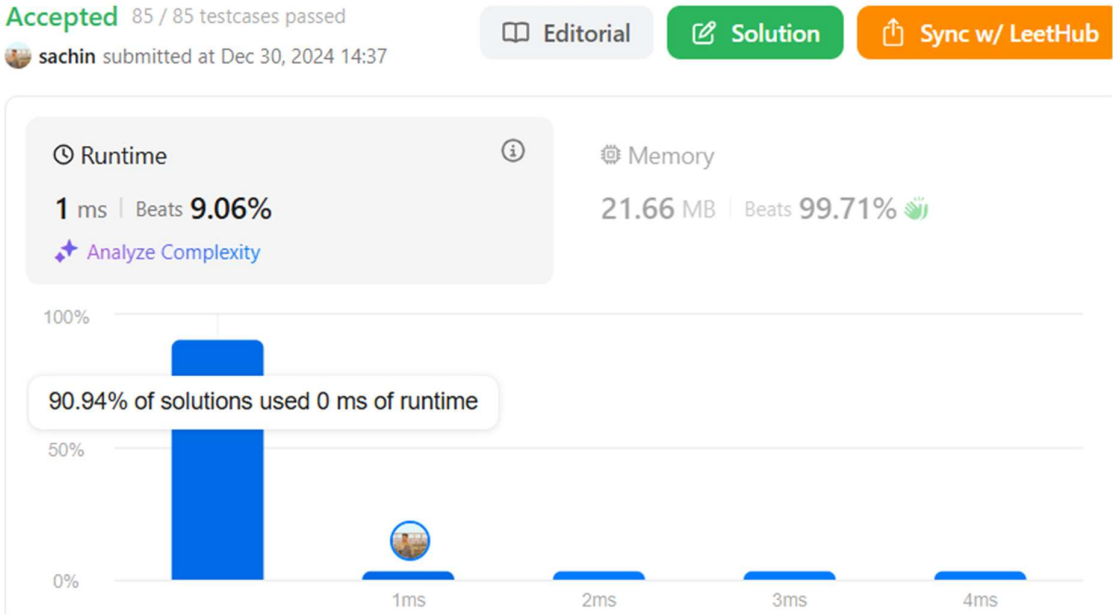


4. 98.Validate Binary Search Tree.
   Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

   CODE:

```
class Solution {
public:
void solve(TreeNode* root,long long& prev,bool& t){
   if(!root|| !t) return;

   solve(root->left,prev,t);
      if(prev>=root->val){
         t=false;
         return;
      }
      prev=root->val;
      solve(root->right,prev,t);
}
   bool isValidBST(TreeNode* root) {
      long long prev=LONG_MIN;
      bool t=true;
      solve(root,prev,t);
      return t;};
```

   OUTPUT:

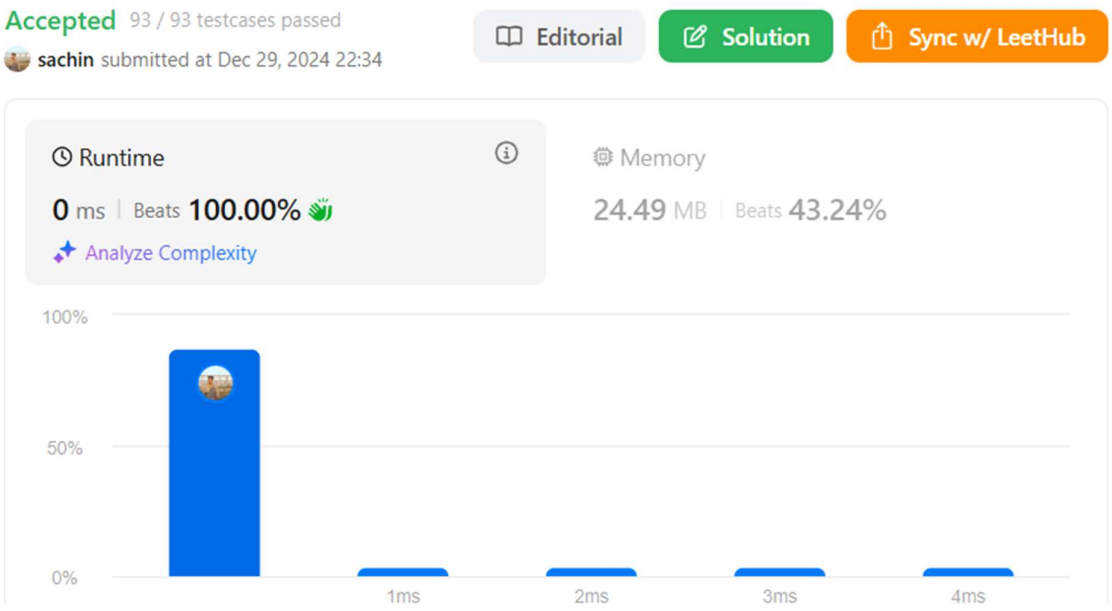


5. 230.Kth Smallest Element in a BST.
   Given the root of a binary search tree, and an integer k,
   return *the* kth *smallest value (**1-indexed**) of all the values of the nodes in the tree*.

   CODE:

```
class Solution {
public:
void solve(TreeNode* root,int& i,int k,int& x){
   if(root==nullptr || x!=-1) return ;

   solve(root->left,i,k,x);
   i++;
   if(i==k){
      x=root->val;
      return;
   }
   solve(root->right,i,k,x);
}
   int kthSmallest(TreeNode* root, int k) {
      int x=-1;
      int i=0;
      solve(root,i,k,x);
      return x; }};
```

   OUTPUT:

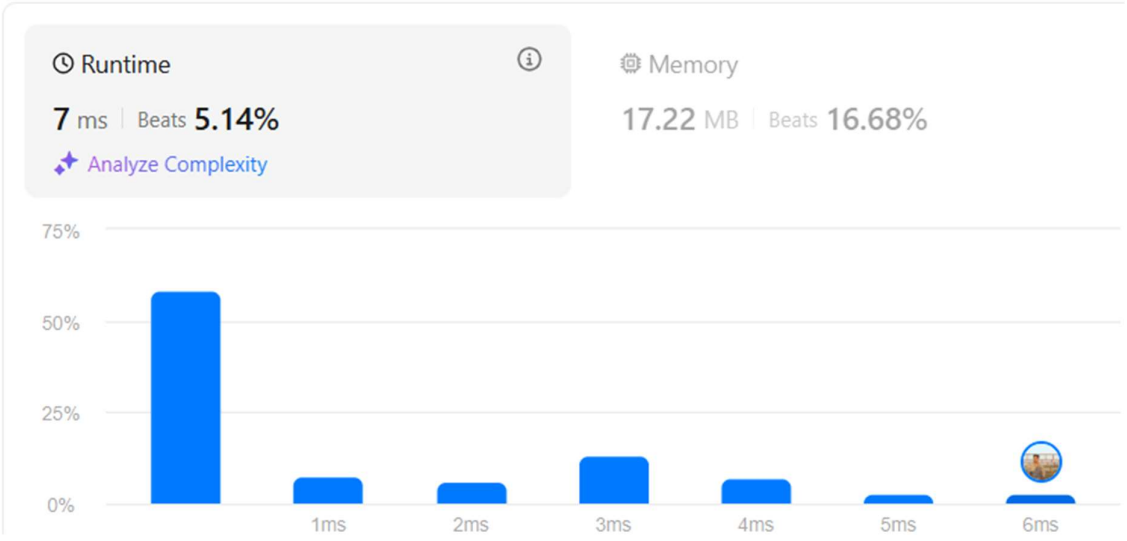


6. **102. Binary Tree Level Order Traversal.**
Given the root of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

CODE:

```
class Solution {
public:

    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>>ans;
        if (root == nullptr) return ans;
        queue<TreeNode*>que;
        que.push(root);
        while(!que.empty()){
        vector<int>t;
        int s=que.size();
            for(int i=0;i<s;i++){
            TreeNode* temp=que.front();
            que.pop();
            if(temp->left!=NULL) que.push(temp->left);
            if(temp->right!=NULL) que.push(temp->right);
            t.push_back(temp->val);
            }
            ans.push_back(t); }
        return ans; }};
```

OUTPUT:

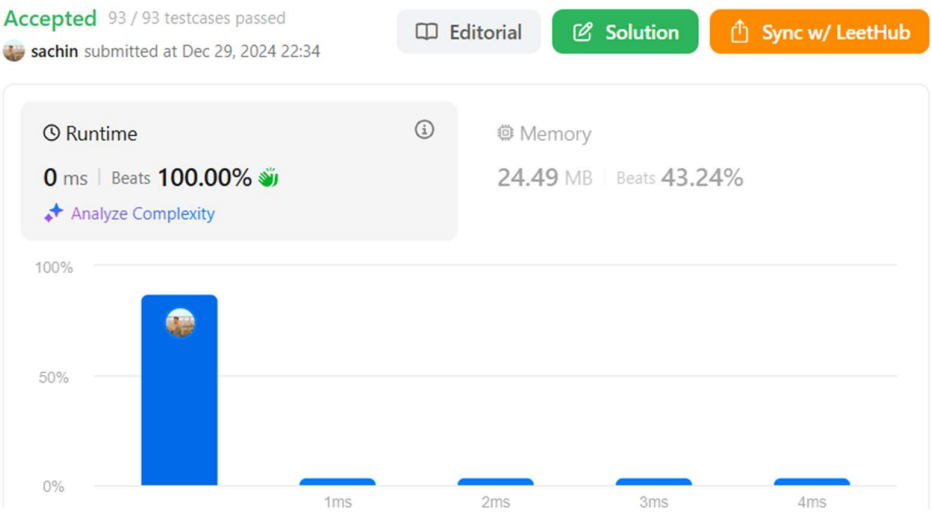


7. **107.Binary Tree Level Order Traversal II.**
Given the root of a binary tree, return *the bottom-up level order traversal of its nodes' values*. (i.e., from left to right, level by level from leaf to root).

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:

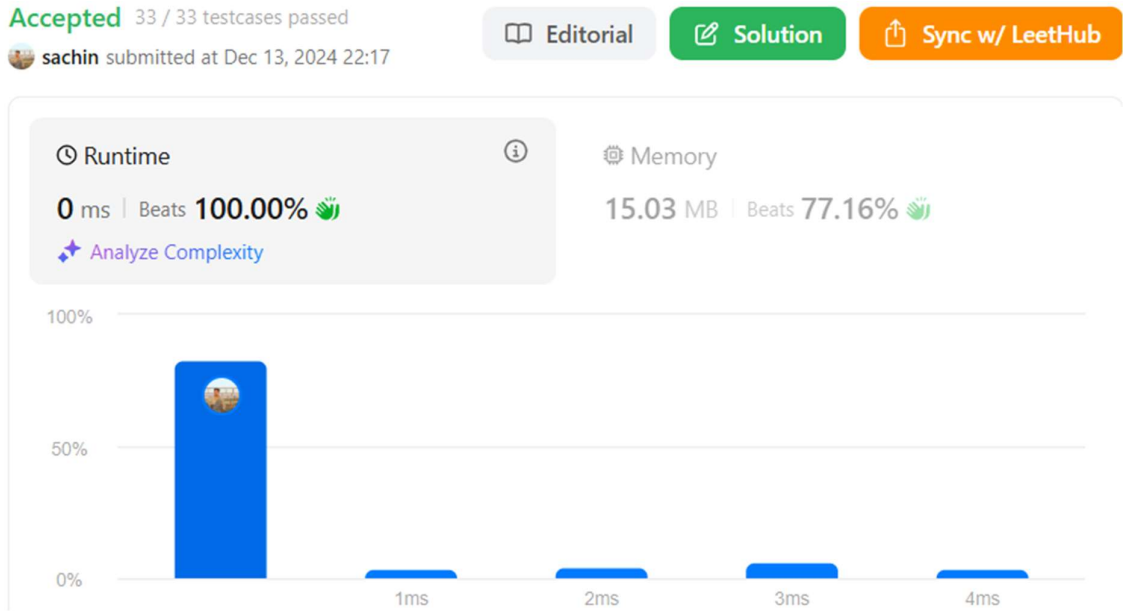


8. 103.Binary Tree Zigzag Level Order Traversal.
Given the root of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

CODE:

```
class Solution {
public:

    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>>ans;
        if(root==NULL) return ans;
        queue<TreeNode*>que;
    que.push(root);
    int flag=1;
    while(!que.empty()){
        int size=que.size();
        vector<int>temp(size);
        for(int i=0;i<size;i++){
       TreeNode* t=que.front();
       que.pop();
       int ind=(flag)?i:(size-1-i);
       temp[ind]=t->val;

       if(t->left){
        que.push(t->left);
       }
       if(t->right){
        que.push(t->right);
       }
        }
        flag=!flag;
        ans.push_back(temp);

    }
        return ans;
    }
};
```

OUTPUT:

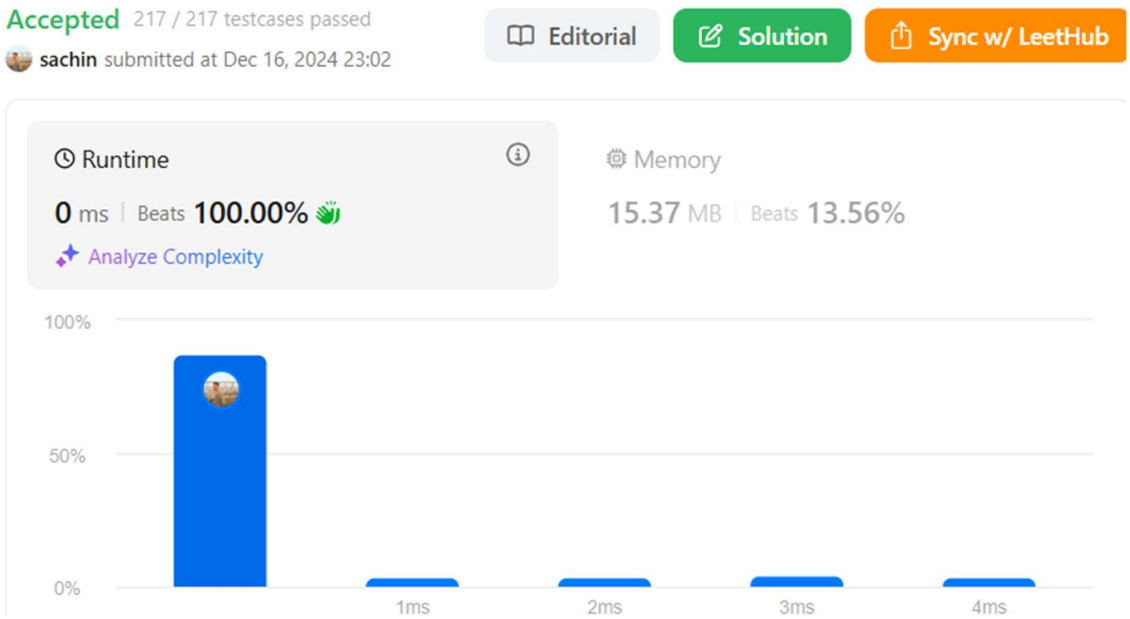


9. **199.[Binary Tree Right Side View](#).**
   Given the root of a binary tree, imagine yourself standing on the right side of it, return *the values of the nodes you can see ordered from top to bottom*.

   CODE:

```
void solve(TreeNode* root,map<int,int>&mapp,int lvl){
    if(root==NULL){
        return;
    }

    mapp[lvl]=root->val;
    solve(root->left,mapp,lvl+1);
    solve(root->right,mapp,lvl+1);
}
    vector<int> rightSideView(TreeNode* root) {
        vector<int>ans;
        map<int,int>mapp;
        solve(root,mapp,0);
        for(auto i:mapp){
            ans.push_back(i.second);
        }
        return ans;
    }
```

OUTPUT:



10.106.[Construct Binary Tree from Inorder and Postorder Traversal](#).

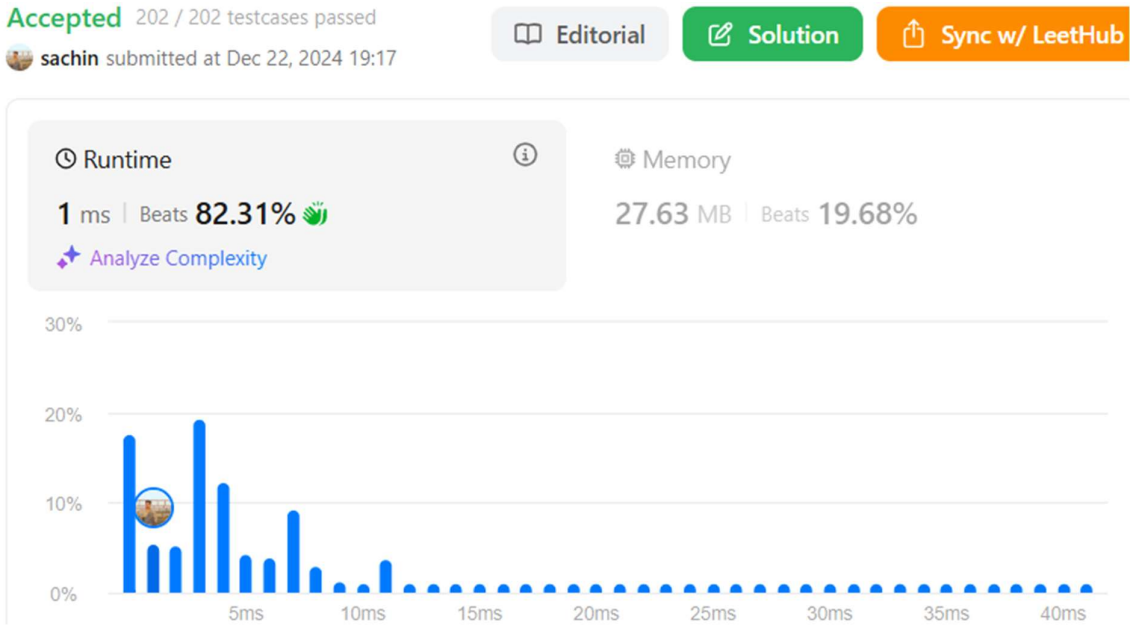
Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return *the binary tree*.

CODE:

```
TreeNode* solve(vector<int>& postorder,int prstart,int prend,
vector<int>& inorder,int instart,int inend,map<int,int>&map){
        if(prstart>prend || instart>inend) return NULL;
        TreeNode* root=new TreeNode(postorder[prend]);

        int x=map[root->val];
        int ind=x-instart;
        root->left=solve(postorder,prstart,prstart+ind-1,inorder,instart,x-
1,map);
        root->right=solve(postorder,prstart+ind,prend-
1,inorder,x+1,inend,map);
        return root;
    }
  TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
      map<int,int>in;
    for(int i=0;i<postorder.size();i++){
        in[inorder[i]]=i;
    }
```

```
        TreeNode*  root=solve(postorder,0,postorder.size()-
1,inorder,0,inorder.size()-1,in);
        return root;
    }
```

OUTPUT:



11.513.[Find Bottom Left Tree Value](#).

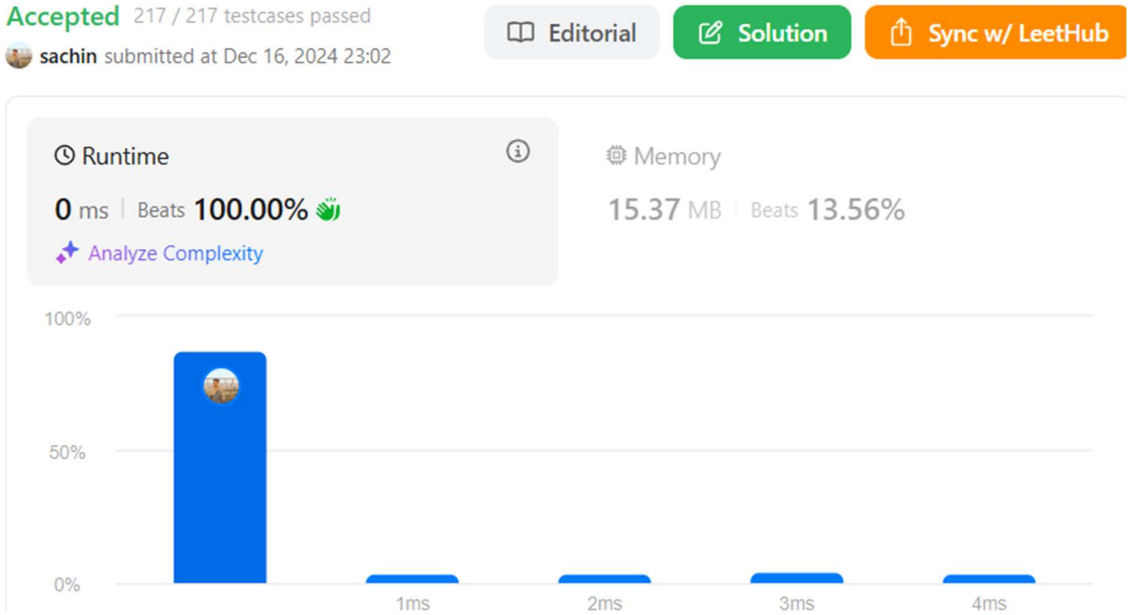
Given the root of a binary tree, return the leftmost value in the last row of the tree.

CODE:

```
void tt(TreeNode* root, int level, vector<vector<int>>&nums){
        if(root==NULL){
            return;
        }
        if(nums.size()<=level){
            nums.push_back({});
        }

        nums[level].push_back(root->val);

        tt(root->right,level+1,nums);
        tt(root->left,level+1,nums);

    }
    int findBottomLeftValue(TreeNode* root) {
```

```
    vector<vector<int>>nums;
     tt(root,0,nums);
     return nums.back().back();
    }
```

OUTPUT:



12.124. [Binary Tree Maximum Path Sum]().

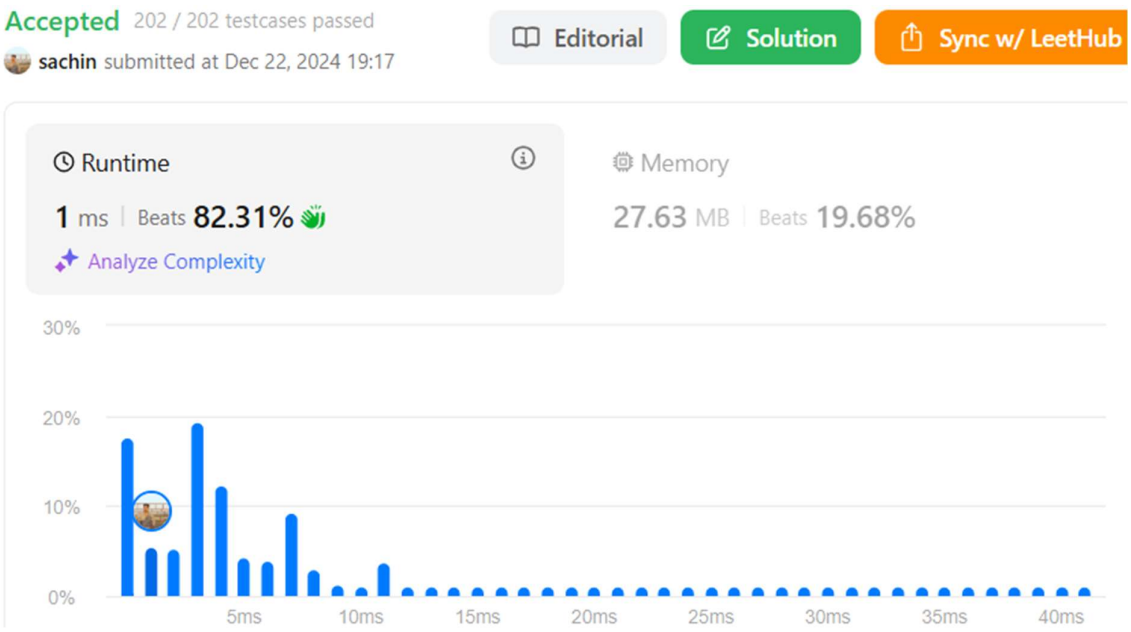
A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

CODE:

```
int solve(TreeNode* root,int &maxi){
    if(root==NULL) return 0;

    int l=max(0,solve(root->left,maxi));
    int r=max(0,solve(root->right,maxi));
    maxi=max(maxi,l+r+root->val);

    return root->val+max(l,r);
}
    int maxPathSum(TreeNode* root) {
        int maxi=INT_MIN;
        // if(root->left==NULL && root->right==NULL) return root->val;
```

```
        solve(root,maxi);
        return maxi;
    }
```

OUTPUT:



13. 987.[Vertical Order Traversal of a Binary Tree](#).

Given the root of a binary tree, calculate the **vertical order traversal** of the binary tree.

CODE:

```
vector<vector<int>> verticalTraversal(TreeNode* root) {
        vector<vector<int>> out;                // Final output vector
        map<int, vector<int>> final_mp;         // Map to store vertical
column indices and their respective node values
        queue<pair<int, TreeNode*>> q;
        q.push({0, root});

        while (!q.empty()) {
            int n = q.size();
            map<int, vector<int>> mp;           // Temporary map for nodes in
the current level

            // Process all nodes in the current level
            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});
         }
         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}}
      for (auto it : final_mp) {
         out.push_back(it.second);}
      return out;}
```

OUTPUT: