Assignment-3(AP lab)

Name:Akul UID:22BCS10330 Section:605-B

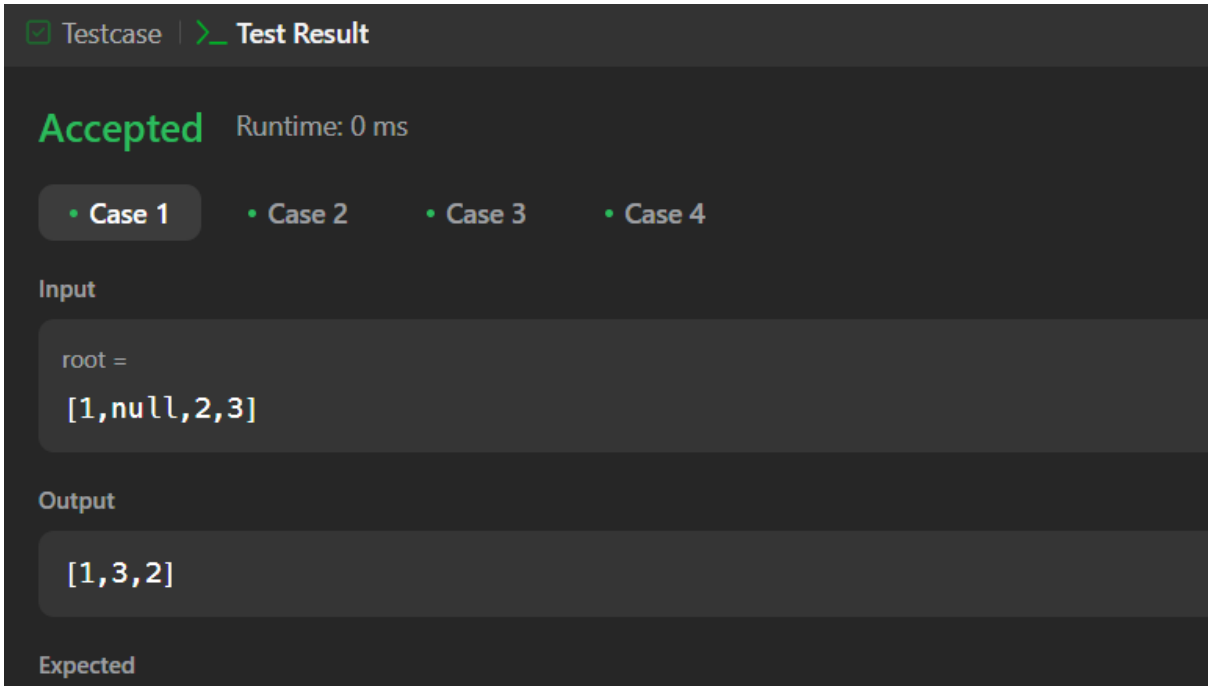
94. Binary tree inorder traversal

Code:

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        inorder(root, res);
        return res;
    }

private:
    void inorder(TreeNode* node, vector<int>& res) {
        if (!node) {
            return;
        }
        inorder(node->left, res);
        res.push_back(node->val);
        inorder(node->right, res);
    }
};
```

Output:



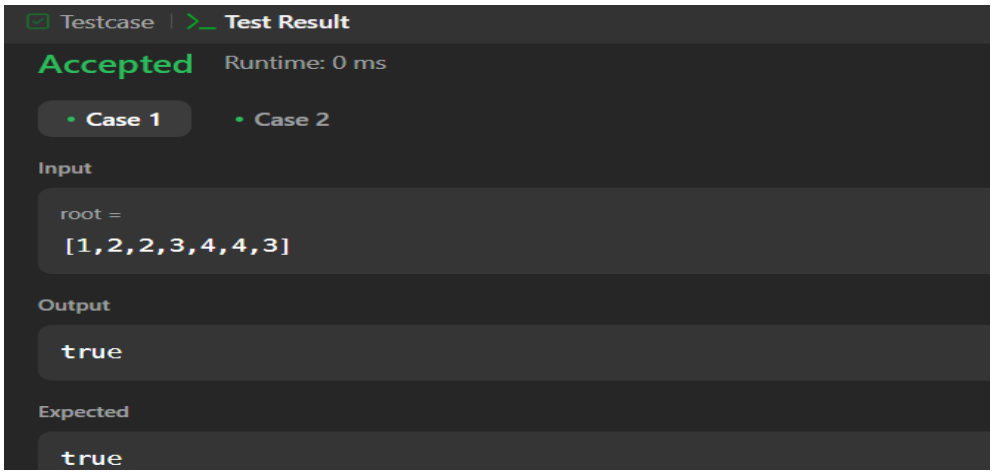
101.Symmetric tree

Code:

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

bool isSymmetric(TreeNode* root) {
    if (!root) return true;
    return isMirror(root->left, root->right);
      }
};
```

Output:

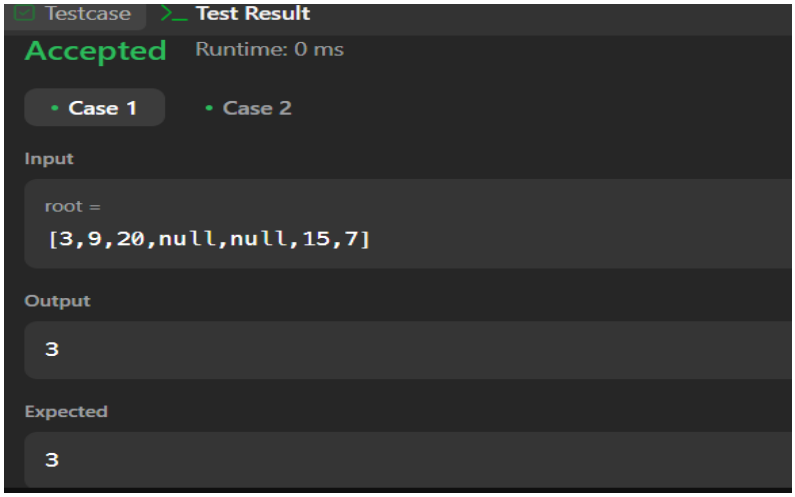


104.Maximum depth of binary tree

Code:

```
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;
    }
};
```

Output:



98.Validate binary search tree

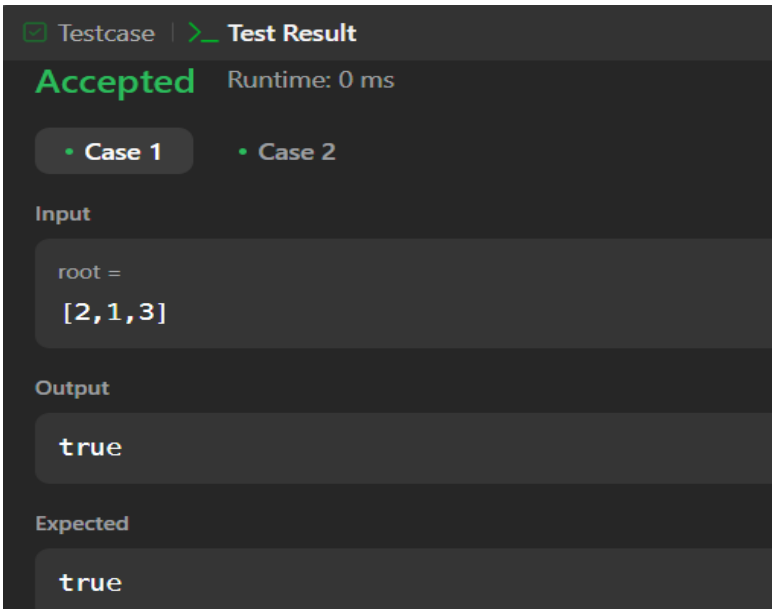
Code:

```
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        return valid(root, LONG_MIN, LONG_MAX);
    }
private:
    bool valid(TreeNode* node, long minimum, long maximum) {
        if (!node) return true;

        if (!(node->val > minimum && node->val < maximum)) return false;

        return valid(node->left, minimum, node->val) && valid(node->right, node->val, maximum);
    }
};
```

Output:

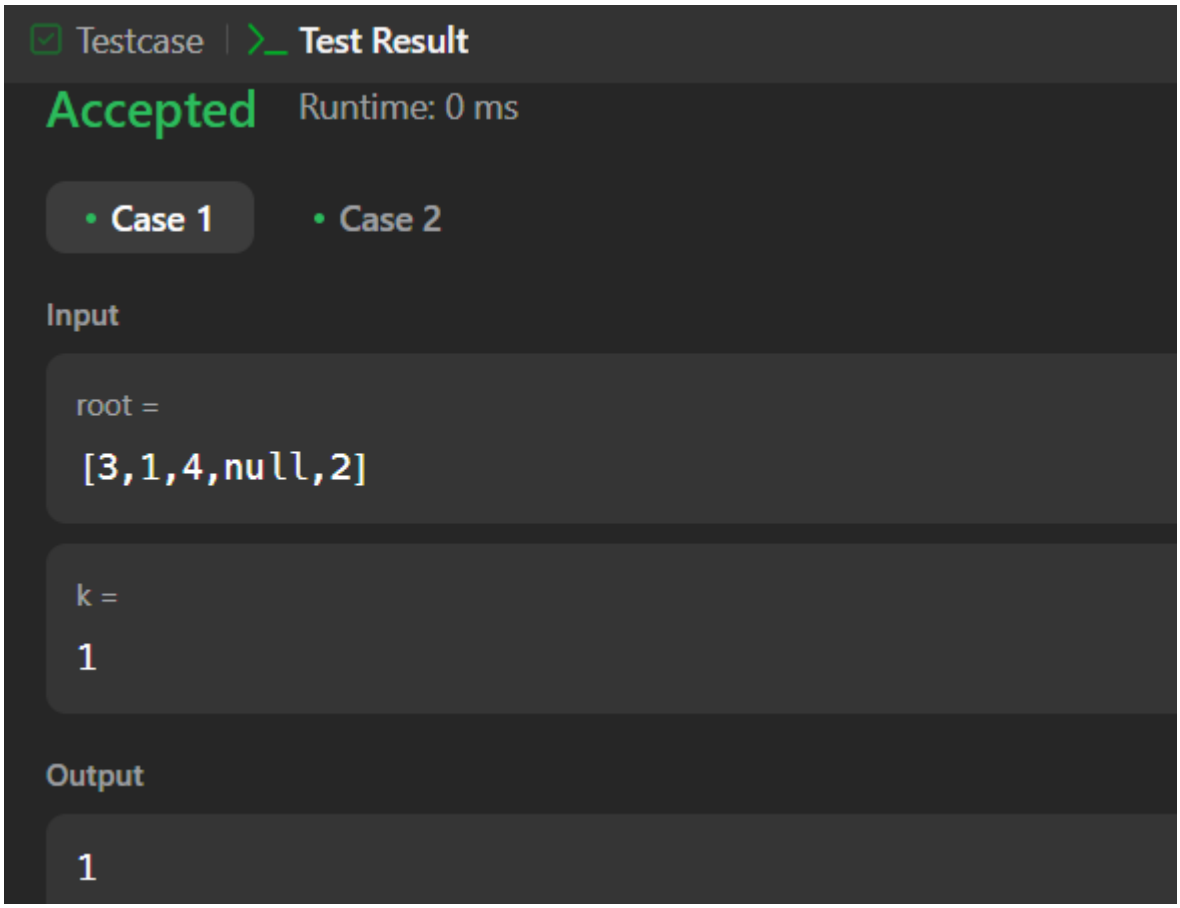


230.k th smallest element in a BST

Code:

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

Output:

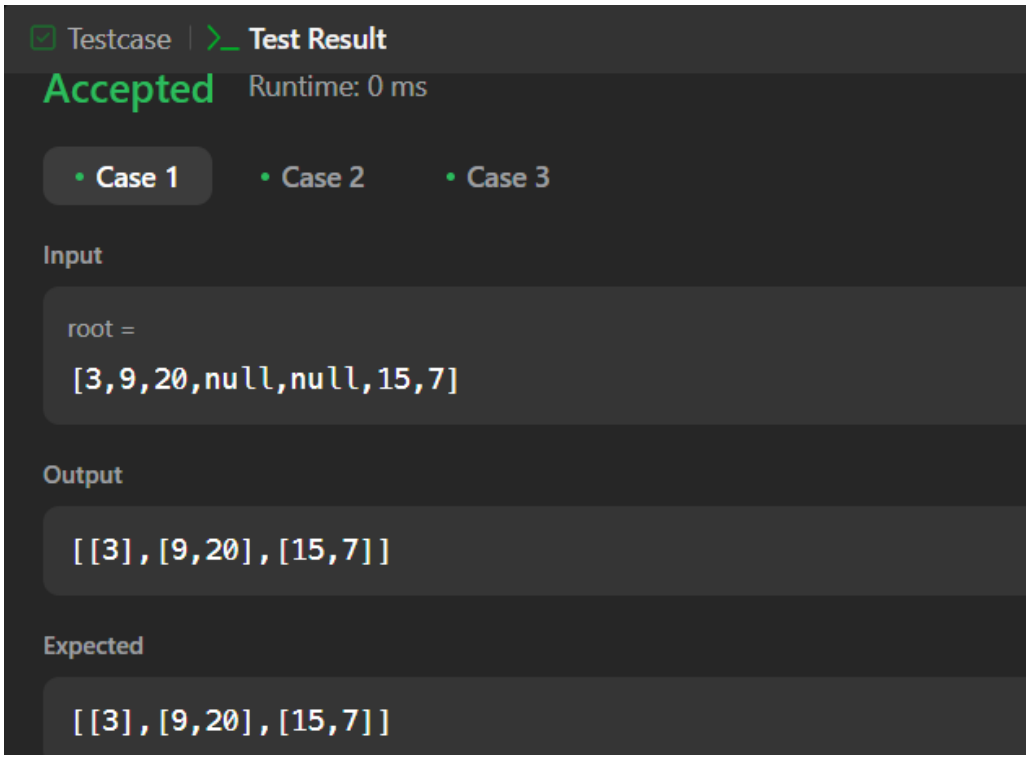


102.Binary tree level order traversal

Code:

```cpp
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 s=q.size();
            vector<int>v;
            for(int i=0;i<s;i++){
                TreeNode *node=q.front();
                q.pop();
                if(node->left!=NULL)q.push(node->left);
                if(node->right!=NULL)q.push(node->right);
                v.push_back(node->val);
            }
            ans.push_back(v);
        }
        return ans;
    }
};
```

Output:

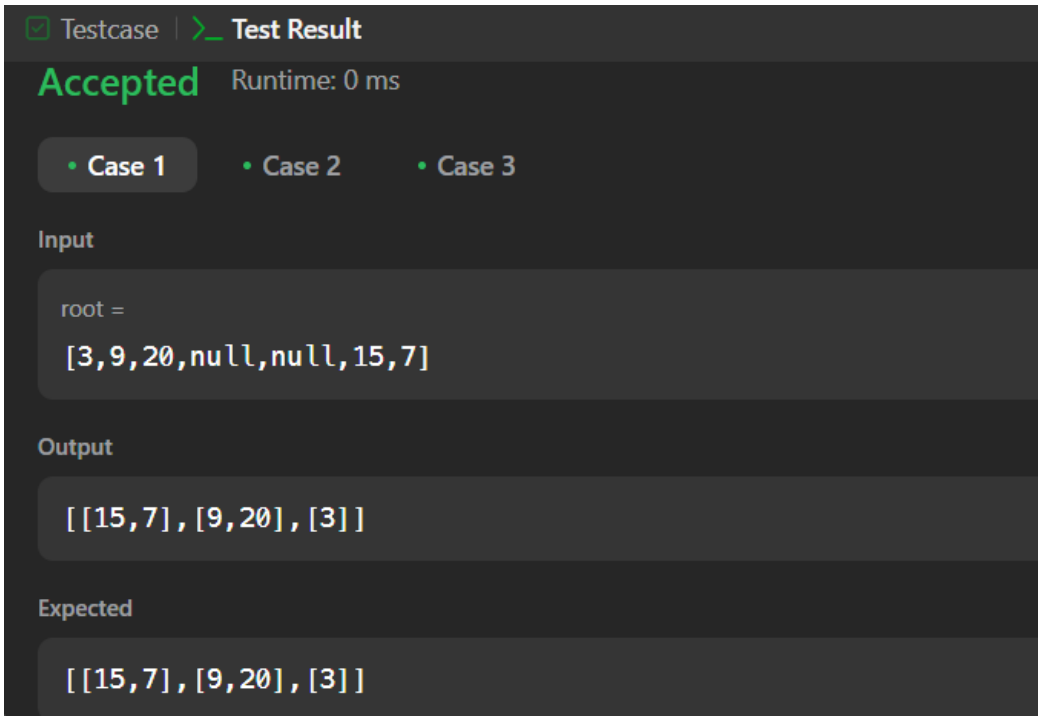


107.Binary tree level order traversal

Code:

```
class Solution {
public:
    vector<vector<int>> levelOrderBottom(TreeNode* root) {
            vector<vector<int>> res;
        if(!root) return res;
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()){
            int size = q.size();
            vector<int> current;
            for(int i = 0; i < size; i++){
                TreeNode* node = q.front();
                q.pop();
                current.push_back(node -> val);
                if(node -> left) q.push(node -> left);
                if(node -> right) q.push(node -> right);
            }
            res.push_back(current);
        }
        reverse(res.begin(), res.end());
        return res;
    }
};
```

Output:

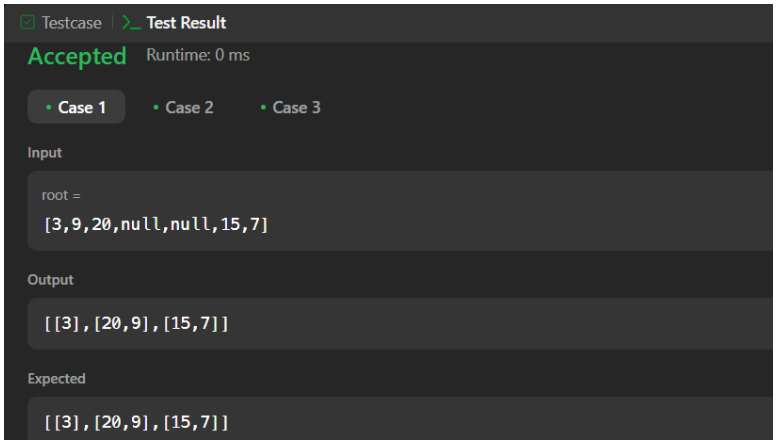


103.Binary tree zigzag level order traversal

Code:

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> res;
        if(root==NULL){
            return res;
        }
        queue<TreeNode*>q;
        q.push(root);
        bool L2R=true;
        while(!q.empty()){
            int size=q.size();
            vector<int> arr(size);
            for(int i=0;i<size;i++){
                TreeNode* node=q.front();
                q.pop();
                int ind=(L2R)?i:(size-i-1);
                arr[ind]=node->val;
                if(node->left){
                    q.push(node->left);
                }
                if(node->right){
                    q.push(node->right);
                }
            }
            L2R=!L2R;
            res.push_back(arr);
        }
        return res;
    }
};
```

Output:



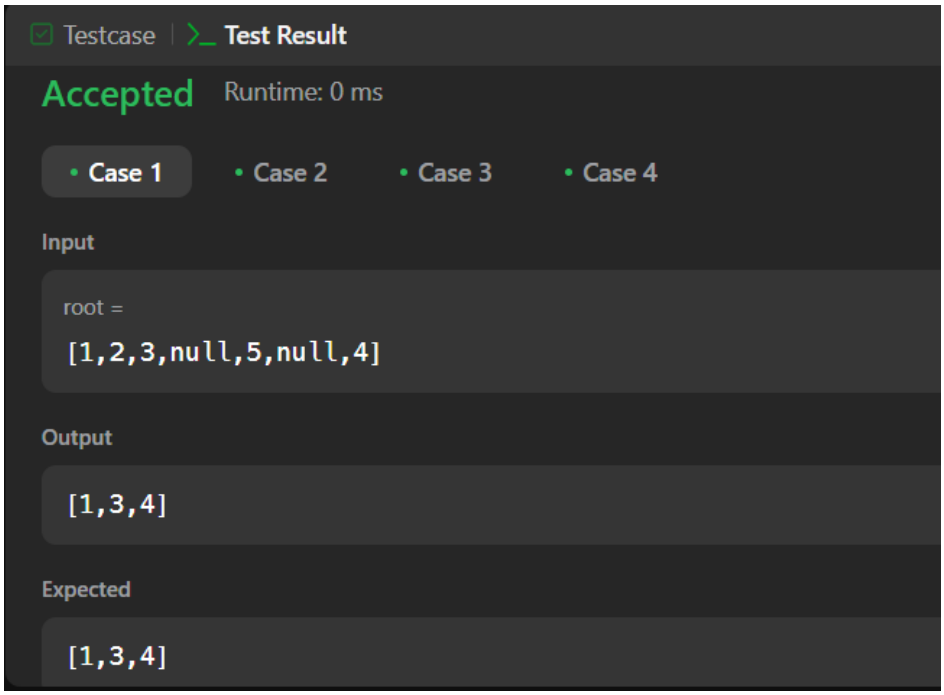
199.Binary tree right side view

Code:

```
class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
        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();
            level.push_back(node->val);
            if(node->left) q.push(node->left);
            if(node->right) q.push(node->right);
        }
        ans.push_back(level[size-1]);
    }
    return ans;
    }
};
```

Output:



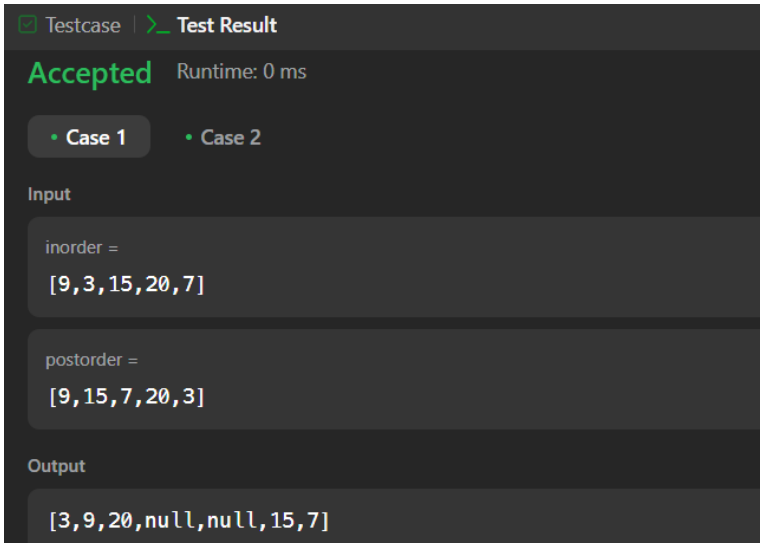
106.Contruct binary tree from inorder and postorder traversal

Code:

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        unordered_map<int, int> index;
        for (int i = 0; i < inorder.size(); i++) {
            index[inorder[i]] = i;
        }
        return buildTreeHelper(inorder, postorder, 0, inorder.size() - 1, 0,
postorder.size() - 1, index);
    }

    TreeNode* buildTreeHelper(vector<int>& inorder, vector<int>& postorder, int
inorderStart, int inorderEnd, int postorderStart, int postorderEnd,
unordered_map<int, int>& index) {
        if (inorderStart > inorderEnd || postorderStart > postorderEnd) {
            return nullptr;
        }
        int rootVal = postorder[postorderEnd];
        TreeNode* root = new TreeNode(rootVal);
        int inorderRootIndex = index[rootVal];
        int leftSubtreeSize = inorderRootIndex - inorderStart;
        root->left = buildTreeHelper(inorder, postorder, inorderStart,
inorderRootIndex - 1, postorderStart, postorderStart + leftSubtreeSize - 1,
index);
        root->right = buildTreeHelper(inorder, postorder, inorderRootIndex + 1,
inorderEnd, postorderStart + leftSubtreeSize, postorderEnd - 1, index);
        return root;
    }
};
```

Output:



513.Find bottom left tree value

Code:

```
class Solution {
public:
    int findBottomLeftValue(TreeNode* root) {
        queue <TreeNode*> q;
        q.push(root);
        TreeNode* cur = NULL;

        while(!q.empty()){
            cur = q.front();
            q.pop();

            if(cur->right) q.push(cur->right);
            if(cur->left) q.push(cur->left);
        }
        return cur->val;

    }
};
```

Output:

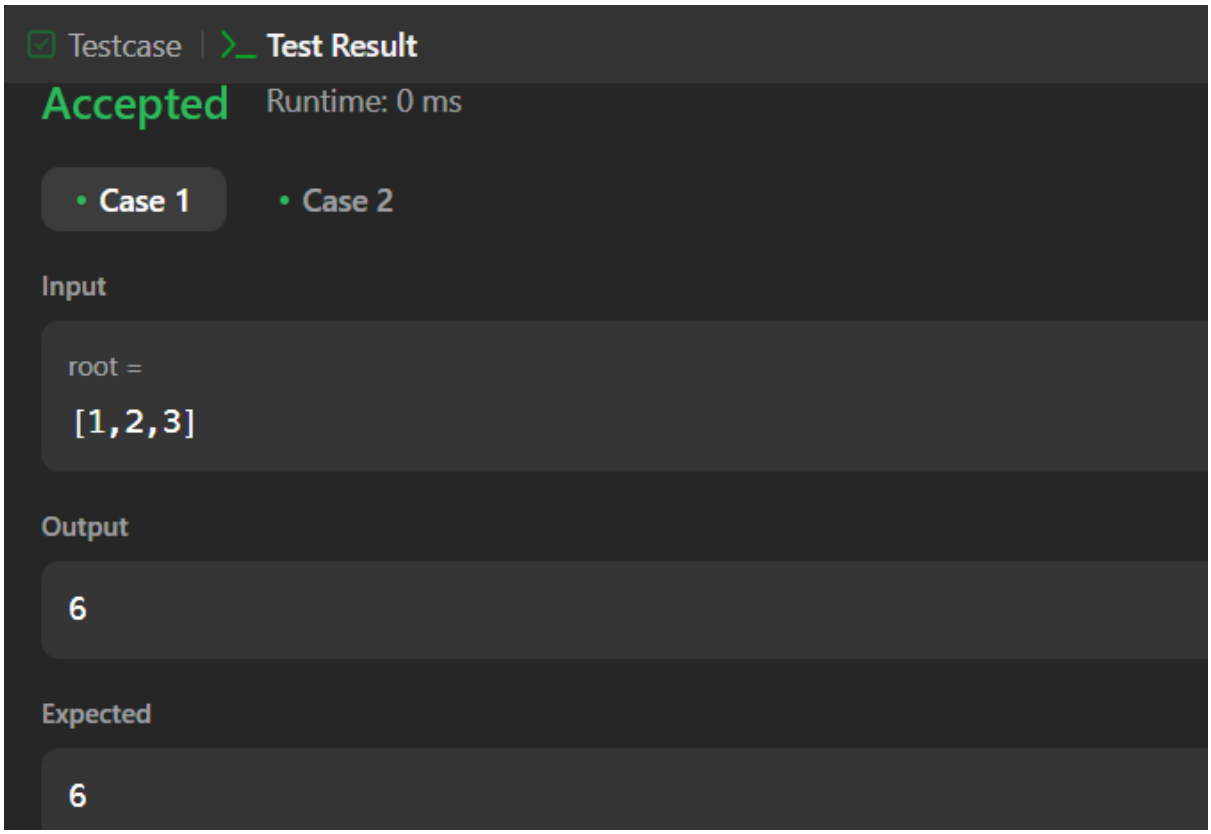
124. Binary tree maximum path sum

Code:

```cpp
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:



987.Vertical order traversal of a binary tree

Code:

```
class Solution {
    public:
        vector<vector<int>> verticalTraversal(TreeNode* root)
        {
            map<int, multiset<pair<int, int>>> mp; // [x][y, val]
            traverse(root, 0, 0, mp);
            vector<vector<int>> res;
            for(auto& [x, st] : mp)
            {
                res.push_back({});
                for(auto& [y, val] : st) res.back().push_back(val);
            }
            return res;
        }

    protected:
        void traverse(TreeNode* node, int x, int y,  map<int, multiset<pair<int,
int>>>& mp)
        {
            if(!node) return;
            mp[x].insert({y, node->val});
            traverse(node->left, x-1, y+1, mp);
            traverse(node->right, x+1, y+1, mp);
        }
};
```

Output: