



DOMAIN WINTER WINNING CAMP ASSIGNMENT

Student Name: Suryansh Gehlot
Branch: BE-CSE::CS201
Semester: 5th

UID: 22BCS10900
Section/Group: 22BCS_FL_IOT-603/B

➤ DAY-6 [25-12-2024]

1. Binary Tree Inorder Traversal

(*Very Easy*)

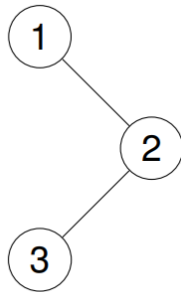
Given the root of a binary tree, return the inorder traversal of its nodes' values.

Example 1:

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

Output: [1,3,2]

Explanation:

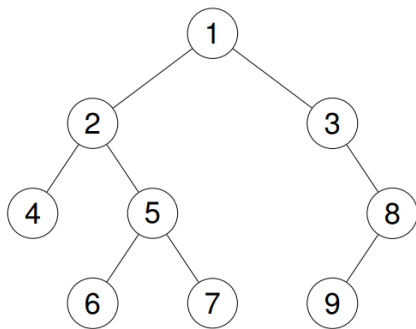


Example 2:

Input: root = [1,2,3,4,5,null,8,null,null,6,7,9]

Output: [4,2,6,5,7,1,3,9,8]

Explanation:



Constraints:

The number of nodes in the tree is in the range [0, 100].

$-100 \leq \text{Node.val} \leq 100$

Implementation/Code:

```
#include <iostream>
#include <vector>
using namespace std;

// 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) {}
};

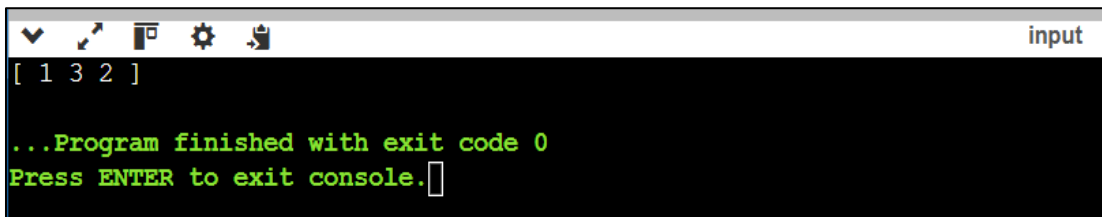
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        inorder(root, result);
        return result;
    }
};
```

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

// Helper function to create a binary tree
TreeNode* createTree() {
    TreeNode* root = new TreeNode(1);
    root->right = new TreeNode(2);
    root->right->left = new TreeNode(3);
    return root;
}

int main() {
    TreeNode* root = createTree();
    Solution sol;
    vector<int> result = sol.inorderTraversal(root);
    cout<<"[ ";
    for (int val : result) {
        cout << val << " ";
    }
    cout<<"]";
    return 0;
}
```

Output:



```
input
[ 1 3 2 ]
...Program finished with exit code 0
Press ENTER to exit console.
```

2. Count Complete Tree Nodes

(Very Easy)

Given the root of a complete binary tree, return the number of the nodes in the tree.

According to Wikipedia, every level, except possibly the last, is completely filled in a complete binary tree, and all nodes in the last level are as far left as possible. It can have between 1 and 2^h nodes inclusive at the last level h .

Design an algorithm that runs in less than $O(n)$ time complexity.

Example 1:

Input: root = [1,2,3,4,5,6]

Output: 6

Example 2:

Input: root = []

Output: 0

Example 3:

Input: root = [1]

Output: 1

Constraints:

The number of nodes in the tree is in the range $[0, 5 * 10^4]$.

$0 \leq \text{Node.val} \leq 5 * 10^4$

The tree is guaranteed to be complete.

Implementation/Code:

```
#include <iostream>
using namespace std;
```

```
// Definition for a binary tree node.
struct TreeNode {
    int val;
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
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 countNodes(TreeNode* root) {
        if (!root) return 0;
        int leftDepth = getDepth(root->left);
        int rightDepth = getDepth(root->right);

        if (leftDepth == rightDepth) {
            return (1 << leftDepth) + countNodes(root->right);
        } else {
            return (1 << rightDepth) + countNodes(root->left);
        }
    }
};
```

```
private:
    int getDepth(TreeNode* node) {
        int depth = 0;
        while (node) {
            depth++;
            node = node->left;
        }
        return depth;
    }
};
```

```
// Helper function to create a binary tree
TreeNode* createTree() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
}
```

```
    root->left->right = new TreeNode(5);
    root->right->left = new TreeNode(6);
    return root;
}

int main() {
    TreeNode* root = createTree();
    Solution sol;
    cout << sol.countNodes(root) << endl;
    return 0;
}
```

Output:



```
input
6

...Program finished with exit code 0
Press ENTER to exit console.
```

3. Binary Tree - Find Maximum Depth

(Very Easy)

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.

Example 1:

Input: [3,9,20,null,null,15,7]

Output: 3

Example 2:

Input: [1,null,2]

Output: 2

Constraints:

The number of nodes in the tree is in the range [0, 104].

-100 <= Node.val <= 100

Implementation/Code:

```
#include <iostream>
using namespace std;

// 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) {}
};

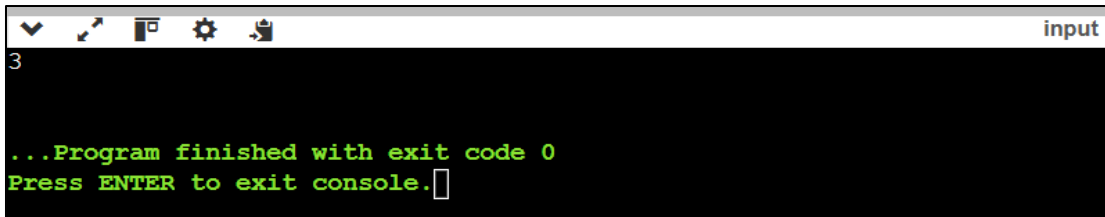
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (!root) return 0;
        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);
        return max(leftDepth, rightDepth) + 1;
    }
};

// Helper function to create a binary tree
TreeNode* createTree() {
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(9);
    root->right = new TreeNode(20);
    root->right->left = new TreeNode(15);
    root->right->right = new TreeNode(7);
    return root;
}

int main() {
    TreeNode* root = createTree();
    Solution sol;
```

```
cout << sol.maxDepth(root) << endl;  
return 0;  
}
```

Output:



```
3  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

4. Binary Tree Preorder Traversal

(Very Easy)

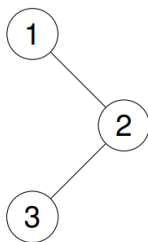
Given the root of a binary tree, return the preorder traversal of its nodes' values.

Example 1:

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

Output: [1,2,3]

Explanation:

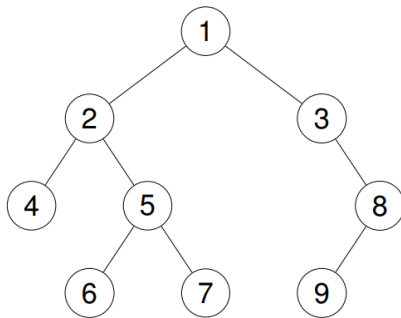


Example 2:

Input: root = [1,2,3,4,5,null,8,null,null,6,7,9]

Output: [1,2,4,5,6,7,3,8,9]

Explanation:



Constraints:

The number of nodes in the tree is in the range [1, 100].

$1 \leq \text{Node.val} \leq 1000$

Implementation/Code:

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
// 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:
```

```
    vector<int> preorderTraversal(TreeNode* root) {
```

```
        vector<int> result;
```

```
        preorder(root, result);
```

```
        return result;
```

```
    }
```

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

// Helper function to create a binary tree
TreeNode* createTree()
{
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->right = new TreeNode(8);
    root->left->right->left = new TreeNode(6);
    root->left->right->right = new TreeNode(7);
    root->right->right->left = new TreeNode(9);
    return root;
}

int main()
{
    TreeNode* root = createTree();
    Solution sol;
    vector<int> result = sol.preorderTraversal(root);
    for (int val : result)
    {
        cout << val << " ";
    }
    return 0;
}
```

Output:



```
input
1 2 4 5 6 7 3 8 9
...Program finished with exit code 0
Press ENTER to exit console.
```

5. Binary Tree - Sum of All Nodes

(Very Easy)

Given the root of a binary tree, you need to find the sum of all the node values in the binary tree.

Example 1:

Input: root = [1, 2, 3, 4, 5, null, 6]

Output: 21

Explanation: The sum of all nodes is $1 + 2 + 3 + 4 + 5 + 6 = 21$.

Example 2:

Input: root = [5, 2, 6, 1, 3, 4, 7]

Output: 28

Explanation: The sum of all nodes is $5 + 2 + 6 + 1 + 3 + 4 + 7 = 28$.

Implementation/Code:

```
#include <iostream>
using namespace std;

// 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 sumOfNodes(TreeNode* root) {
        if (!root) return 0;
        return root->val + sumOfNodes(root->left) + sumOfNodes(root->right);
    }
};

// Helper function to create a binary tree
TreeNode* createTree()
{
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->right = new TreeNode(6);
    return root;
}

int main()
{
    TreeNode* root = createTree();
    Solution sol;
    cout << sol.sumOfNodes(root) << endl;
    return 0;
}
```

Output:



```
21
...Program finished with exit code 0
Press ENTER to exit console.
```

6. Same Tree

(Easy)

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Example 1:

Input: p = [1,2,3], q = [1,2,3]

Output: true

Example 2:

Input: p = [1,2], q = [1,null,2]

Output: false

Constraints:

The number of nodes in both trees is in the range [0, 100].

-104 <= Node.val <= 104

Implementation/Code:

```
#include <iostream>
using namespace std;
```

```
// 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:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if (!p && !q) return true;
```

```
        if (!p || !q || p->val != q->val) return false;
        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};
```

```
// Helper function to create a tree
TreeNode* createTree1() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    return root;
}
```

```
TreeNode* createTree2() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    return root;
}
```

```
int main()
{
    TreeNode* p = createTree1();
    TreeNode* q = createTree2();
    Solution sol;
    cout << (sol.isSameTree(p, q) ? "true" : "false") << endl;
    return 0;
}
```

Output:



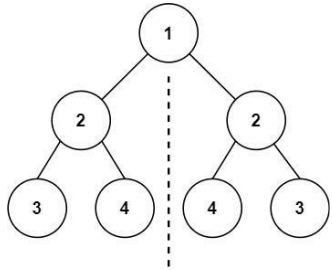
```
input
true

...Program finished with exit code 0
Press ENTER to exit console.
```

7. Symmetric Tree

(Easy)

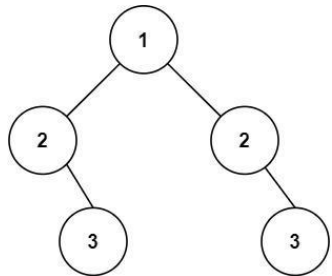
Example 1:



Input: root = [1,2,2,3,4,4,3]

Output: true

Example 2:



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

Output: false

Constraints:

The number of nodes in the tree is in the range [1, 1000].

-100 <= Node.val <= 100

Implementation/Code:

```
#include <iostream>
using namespace std;
```

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

private:
    bool isMirror(TreeNode* t1, TreeNode* t2) {
        if (!t1 && !t2) return true;
        if (!t1 || !t2 || t1->val != t2->val) return false;
        return isMirror(t1->left, t2->right) && isMirror(t1->right, t2->left);
    }
};

TreeNode* createTree() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(2);
    root->left->left = new TreeNode(3);
    root->left->right = new TreeNode(4);
    root->right->left = new TreeNode(4);
    root->right->right = new TreeNode(3);
    return root;
}

int main() {
    TreeNode* root = createTree();
    Solution sol;
    cout << (sol.isSymmetric(root) ? "true" : "false") << endl;
    return 0;
}
```


Output:

```

input
true

...Program finished with exit code 0
Press ENTER to exit console.

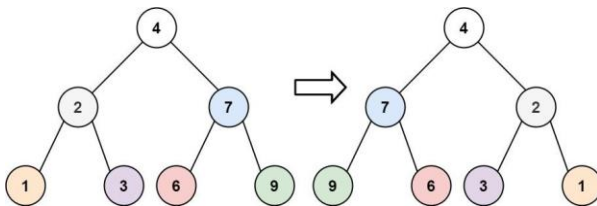
```

8. Invert Binary Tree

(Easy)

Given the root of a binary tree, invert the tree, and return its root.

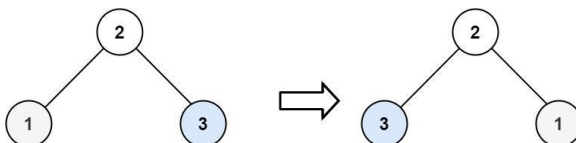
Example 1:



Input: root = [4,2,7,1,3,6,9]

Output: [4,7,2,9,6,3,1]

Example 2:



Input: root = [2,1,3]

Output: [2,3,1]

Constraints:

The number of nodes in the tree is in the range [0, 100].

-100 <= Node.val <= 100

Implementation/Code:

```
#include <iostream>
using namespace std;

// 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) {}
};

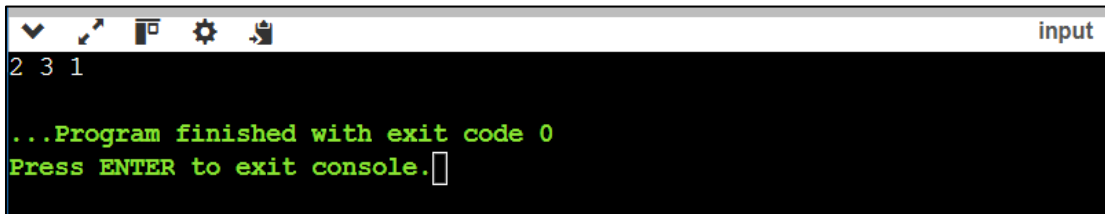
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (!root) return nullptr;
        swap(root->left, root->right);
        invertTree(root->left);
        invertTree(root->right);
        return root;
    }
};

// Helper function to create a tree
TreeNode* createTree() {
    TreeNode* root = new TreeNode(2);
    root->left = new TreeNode(1);
    root->right = new TreeNode(3);
    return root;
}

void printTree(TreeNode* root) {
    if (!root) return;
    cout << root->val << " ";
```

```
    printTree(root->left);  
    printTree(root->right);  
}  
  
int main() {  
    TreeNode* root = createTree();  
    Solution sol;  
    TreeNode* inverted = sol.invertTree(root);  
    printTree(inverted);  
    return 0;  
}
```

Output:

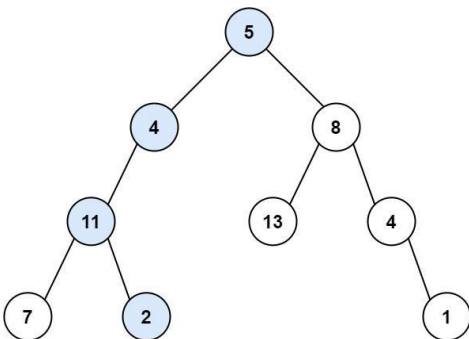


9. Path Sum

(Easy)

Given a binary tree and a sum, return **true** if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. Return **false** if no such path can be found.

Example 1:

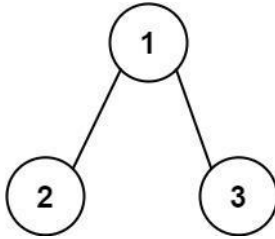


Input: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22

Output: true

Explanation: The root-to-leaf path with the target sum is shown.

Example 2:



Input: root = [1,2,3], targetSum = 5

Output: false

Explanation: There are two root-to-leaf paths in the tree:

(1 --> 2): The sum is 3.

(1 --> 3): The sum is 4.

There is no root-to-leaf path with sum = 5.

Example 3:

Input: root = [], targetSum = 0

Output: false

Explanation: Since the tree is empty, there are no root-to-leaf paths.

Implementation/Code:

```
#include <iostream>
using namespace std;

// Definition for a binary tree node.
struct TreeNode
{
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
}
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
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:
    bool hasPathSum(TreeNode* root, int targetSum) {
        if (!root) return false;
        if (!root->left && !root->right) return root->val == targetSum;
        return hasPathSum(root->left, targetSum - root->val) || hasPathSum(root->right,
targetSum - root->val);
    }
};

// Helper function to create a tree
TreeNode* createTree()
{
    TreeNode* root = new TreeNode(5);
    root->left = new TreeNode(4);
    root->right = new TreeNode(8);
    root->left->left = new TreeNode(11);
    root->right->left = new TreeNode(13);
    root->right->right = new TreeNode(4);
    root->left->left->left = new TreeNode(7);
    root->left->left->right = new TreeNode(2);
    root->right->right->right = new TreeNode(1);
    return root;
}

int main()
{
    TreeNode* root = createTree();
    Solution sol;
    cout << (sol.hasPathSum(root, 22) ? "true" : "false") << endl;
    return 0;
}
```

Output:

```

input
true

...Program finished with exit code 0
Press ENTER to exit console.

```

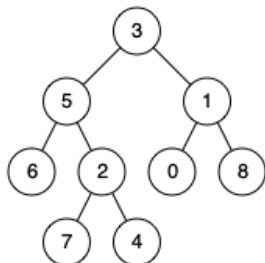
10. Lowest Common Ancestor of a Binary Tree

(Medium)

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

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:

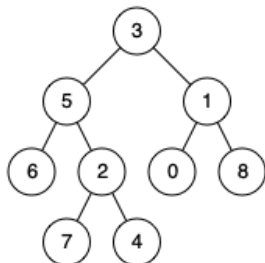


Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

Explanation: The LCA of nodes 5 and 1 is 3.

Example 2:



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

Output: 5

Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

Example 3:

Input: root = [1,2], p = 1, q = 2

Output: 1

Constraints:

- The number of nodes in the tree is in the range [2, 105].
- $-109 \leq \text{Node.val} \leq 109$
- All Node.val are unique.
- $p \neq q$
- p and q will exist in the tree.

Implementation/Code:

```
#include <iostream>
using namespace std;

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root || root == p || root == q) return root;
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);
        return left && right ? root : (left ? left : right);
    }
};
```

```
// Main function
int main() {
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(5);
    root->right = new TreeNode(1);
    root->left->left = new TreeNode(6);
    root->left->right = new TreeNode(2);
    root->right->left = new TreeNode(0);
    root->right->right = new TreeNode(8);

    Solution sol;
    TreeNode* lca = sol.lowestCommonAncestor(root, root->left, root->right);
    cout << lca->val << endl; // Expected: 3
    return 0;
}
```

Output:



```
3
...Program finished with exit code 0
Press ENTER to exit console.
```

11. Sum Root to Leaf Numbers

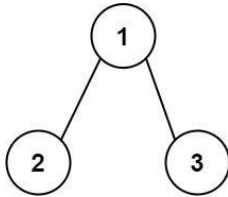
(Medium)

You are given the root of a binary tree containing digits from 0 to 9 only. Each root-to-leaf path in the tree represents a number. A leaf node is a node with no children.

For example, the root-to-leaf path 1 -> 2 -> 3 represents the number 123.

Return the total sum of all root-to-leaf numbers. Test cases are generated so that the answer will fit in a 32-bit integer.

Example 1:



Input: root = [1,2,3]

Output: 25

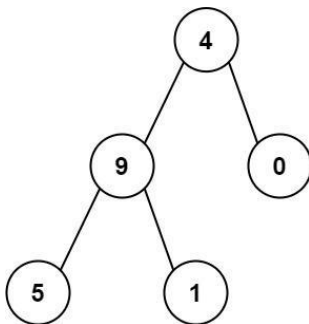
Explanation:

The root-to-leaf path 1->2 represents the number 12.

The root-to-leaf path 1->3 represents the number 13.

Therefore, sum = 12 + 13 = 25.

Example 2:



Input: root = [4,9,0,5,1]

Output: 1026

Explanation:

The root-to-leaf path 4->9->5 represents the number 495.

The root-to-leaf path 4->9->1 represents the number 491.

The root-to-leaf path 4->0 represents the number 40.

Therefore, sum = 495 + 491 + 40 = 1026.

Constraints:

The number of nodes in the tree is in the range [1, 1000].

0 <= Node.val <= 9

The depth of the tree will not exceed 10.

Implementation/Code:



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
#include <iostream>
using namespace std;

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

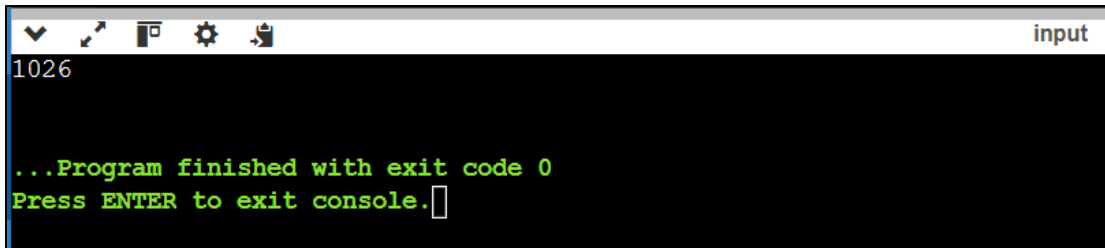
class Solution {
public:
    int sumNumbers(TreeNode* root) {
        return dfs(root, 0);
    }

private:
    int dfs(TreeNode* node, int currentSum) {
        if (!node) return 0;
        currentSum = currentSum * 10 + node->val;
        if (!node->left && !node->right) return currentSum;
        return dfs(node->left, currentSum) + dfs(node->right, currentSum);
    }
};

// Main function
int main() {
    TreeNode* root = new TreeNode(4);
    root->left = new TreeNode(9);
    root->right = new TreeNode(0);
    root->left->left = new TreeNode(5);
    root->left->right = new TreeNode(1);

    Solution sol;
    cout << sol.sumNumbers(root) << endl; // Expected: 1026
    return 0;
}
```

Output:



```
1026

...Program finished with exit code 0
Press ENTER to exit console.
```

12. Binary Tree Maximum Path Sum

(Hard)

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.

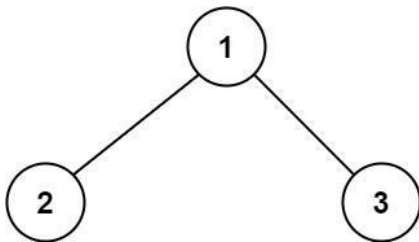
A node can only appear in the sequence at most once.

Note that the path does not need to pass through the root.

The path sum of a path is the sum of the node's values in the path.

Given the root of a binary tree, return the maximum path sum of any non-empty path.

Example 1:

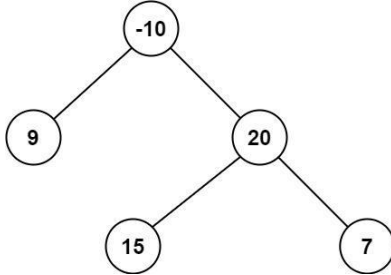


Input: root = [1,2,3]

Output: 6

Explanation: The optimal path is 2 -> 1 -> 3 with a path sum of $2 + 1 + 3 = 6$.

Example 2:



Input: root = [-10,9,20,null,null,15,7]

Output: 42

Explanation: The optimal path is 15 -> 20 -> 7 with a path sum of $15 + 20 + 7 = 42$.

Constraints:

The number of nodes in the tree is in the range $[1, 3 * 10^4]$.

$-1000 \leq \text{Node.val} \leq 1000$

Implementation/Code:

```
#include <iostream>
#include <algorithm>
#include <climits>
using namespace std;

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    int maxPathSum(TreeNode* root) {
        int maxSum = INT_MIN;
```

```
        maxGain(root, maxSum);
        return maxSum;
    }

private:
    int maxGain(TreeNode* node, int& maxSum)
    {
        if (!node) return 0;

        // Recursively calculate the maximum gain from left and right subtrees
        int leftGain = max(maxGain(node->left, maxSum), 0);
        int rightGain = max(maxGain(node->right, maxSum), 0);

        // The current path sum including the current node
        int currentPathSum = node->val + leftGain + rightGain;

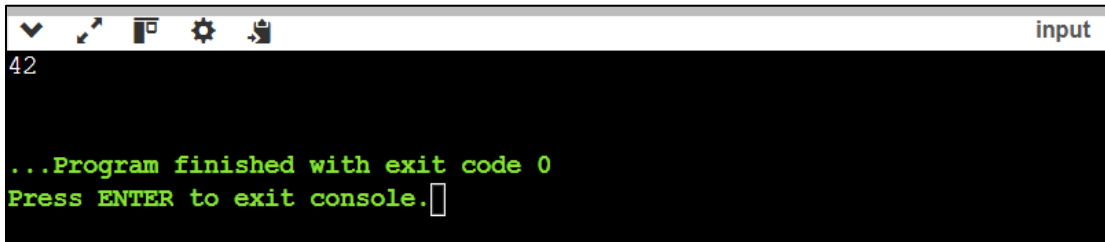
        // Update the maximum path sum if the current path sum is greater
        maxSum = max(maxSum, currentPathSum);

        // Return the maximum gain the current node contributes to its parent
        return node->val + max(leftGain, rightGain);
    }
};

// Main function
int main()
{
    TreeNode* root = new TreeNode(-10);
    root->left = new TreeNode(9);
    root->right = new TreeNode(20);
    root->right->left = new TreeNode(15);
    root->right->right = new TreeNode(7);

    Solution sol;
    cout << sol.maxPathSum(root) << endl; // Expected: 42
    return 0;
}
```

Output:



```
42

...Program finished with exit code 0
Press ENTER to exit console.
```

13. Kth Smallest Element in a BST (Binary Search Tree) (Hard)

Given a binary search tree (BST), write a function to find the kth smallest element in the tree.

Example 1:

Input: root = [3,1,4,null,2], k = 1

Output: 1

Explanation: The inorder traversal of the BST is [1, 2, 3, 4], and the 1st smallest element is 1.

Example 2:

Input: root = [5,3,6,2,4,null,null,1], k = 3

Output: 3

Explanation: The inorder traversal of the BST is [1, 2, 3, 4, 5, 6], and the 3rd smallest element is 3.

Constraints:

The number of nodes in the tree is in the range [1, 1000].

$-10^4 \leq \text{Node.val} \leq 10^4$.

Implementation/Code:

```
#include <iostream>
```

```
#include <stack>
```

```
using namespace std;
```

```
// Definition for a binary tree node.
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
struct TreeNode {  
    int val;  
    TreeNode* left;  
    TreeNode* right;  
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}  
};
```

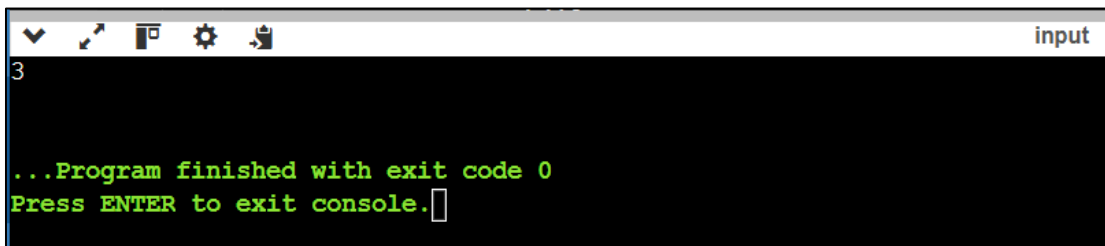
```
class Solution {  
public:  
    int kthSmallest(TreeNode* root, int k) {  
        stack<TreeNode*> stk;  
        TreeNode* current = root;  
        int count = 0;  
  
        while (current || !stk.empty()) {  
            // Traverse the left subtree  
            while (current) {  
                stk.push(current);  
                current = current->left;  
            }  
  
            // Visit the node  
            current = stk.top();  
            stk.pop();  
            count++;  
            if (count == k) return current->val;  
  
            // Traverse the right subtree  
            current = current->right;  
        }  
  
        return -1; // This line should never be reached  
    }  
};
```

```
// Main function  
int main() {  
    TreeNode* root = new TreeNode(5);
```

```
root->left = new TreeNode(3);
root->right = new TreeNode(6);
root->left->left = new TreeNode(2);
root->left->right = new TreeNode(4);
root->left->left->left = new TreeNode(1);
```

```
Solution sol;
int k = 3;
cout << sol.kthSmallest(root, k) << endl; // Expected: 3
return 0;
}
```

Output:



```
input
3
...Program finished with exit code 0
Press ENTER to exit console.
```

14. Count Paths That Can Form a Palindrome in a Tree *(Very Hard)*

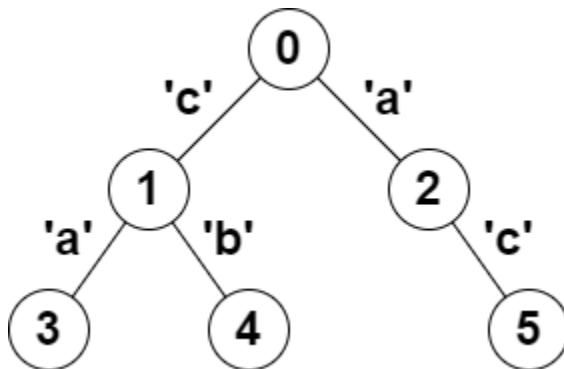
You are given a tree (i.e. a connected, undirected graph that has no cycles) rooted at node 0 consisting of n nodes numbered from 0 to $n - 1$. The tree is represented by a 0-indexed array `parent` of size n , where `parent[i]` is the parent of node i . Since node 0 is the root, `parent[0] == -1`.

You are also given a string `s` of length n , where `s[i]` is the character assigned to the edge between i and `parent[i]`. `s[0]` can be ignored.

Return the number of pairs of nodes (u, v) such that $u < v$ and the characters assigned to edges on the path from u to v can be rearranged to form a palindrome.

A string is a palindrome when it reads the same backwards as forwards.

Example 1:



Input: parent = [-1,0,0,1,1,2], s = "acaabc"

Output: 8

Explanation: The valid pairs are:

- All the pairs (0,1), (0,2), (1,3), (1,4) and (2,5) result in one character which is always a palindrome.
- The pair (2,3) result in the string "aca" which is a palindrome.
- The pair (1,5) result in the string "cac" which is a palindrome.
- The pair (3,5) result in the string "acac" which can be rearranged into the palindrome "acca".

Example 2:

Input: parent = [-1,0,0,0,0], s = "aaaaa"

Output: 10

Explanation: Any pair of nodes (u,v) where $u < v$ is valid.

Constraints:

$n == \text{parent.length} == \text{s.length}$

$1 \leq n \leq 105$

$0 \leq \text{parent}[i] \leq n - 1$ for all $i \geq 1$

$\text{parent}[0] == -1$

parent represents a valid tree.

s consists of only lowercase English letters.

Implementation/Code:

```
#include <iostream>
```

```
#include <vector>
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
#include <unordered_map>
#include <bitset>
using namespace std;

class Solution {
public:
    int countPalindromePaths(vector<int>& parent, string s) {
        int n = parent.size();
        vector<vector<int>>> tree(n);
        for (int i = 1; i < n; ++i) {
            tree[parent[i]].push_back(i);
        }

        unordered_map<int, int> freq;
        freq[0] = 1; // Base case: empty path
        int result = 0;

        dfs(0, 0, tree, s, freq, result);
        return result;
    }

private:
    void dfs(int node, int mask, vector<vector<int>>>& tree, string& s, unordered_map<int,
int>& freq, int& result) {
        // Update the mask by flipping the bit corresponding to the current character
        mask ^= (1 << (s[node] - 'a'));

        // Check for palindromic paths
        result += freq[mask]; // Exact match
        for (int i = 0; i < 26; ++i) {
            result += freq[mask ^ (1 << i)]; // One bit difference
        }

        // Update the frequency map
        freq[mask]++;

        // Recur for children
        for (int child : tree[node]) {
```

```
        dfs(child, mask, tree, s, freq, result);
    }

    // Backtrack
    freq[mask]--;
}

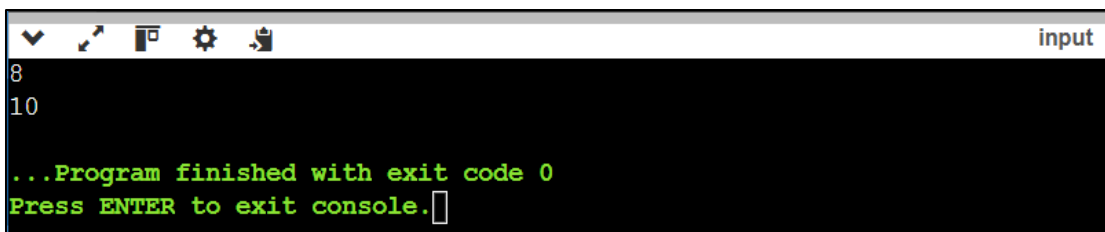
};

// Main function
int main() {
    Solution sol;

    vector<int> parent1 = {-1, 0, 0, 1, 1, 2};
    string s1 = "acaabc";
    cout << sol.countPalindromePaths(parent1, s1) << endl; // Expected: 8

    vector<int> parent2 = {-1, 0, 0, 0, 0};
    string s2 = "aaaaa";
    cout << sol.countPalindromePaths(parent2, s2) << endl; // Expected: 10

    return 0;
}
```

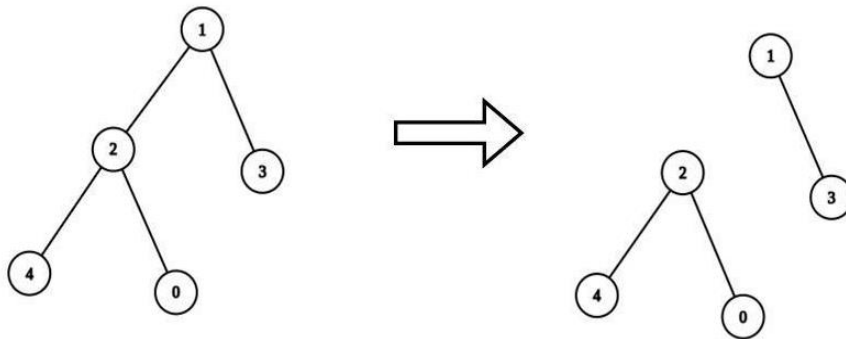
Output:**15. Maximum Number of K-Divisible Components****(Very Hard)**

There is an undirected tree with n nodes labeled from 0 to $n - 1$. You are given the integer n and a 2D integer array `edges` of length $n - 1$, where `edges[i] = [ai, bi]` indicates that there is an edge between nodes ai and bi in the tree.

You are also given a 0-indexed integer array `values` of length `n`, where `values[i]` is the value associated with the `i`th node, and an integer `k`. Return the maximum number of components in any valid split.

A valid split of the tree is obtained by removing any set of edges, possibly empty, from the tree such that the resulting components all have values that are divisible by `k`, where the value of a connected component is the sum of the values of its nodes.

Example 1:



Input: `n = 5`, `edges = [[0,2],[1,2],[1,3],[2,4]]`, `values = [1,8,1,4,4]`, `k = 6`

Output: 2

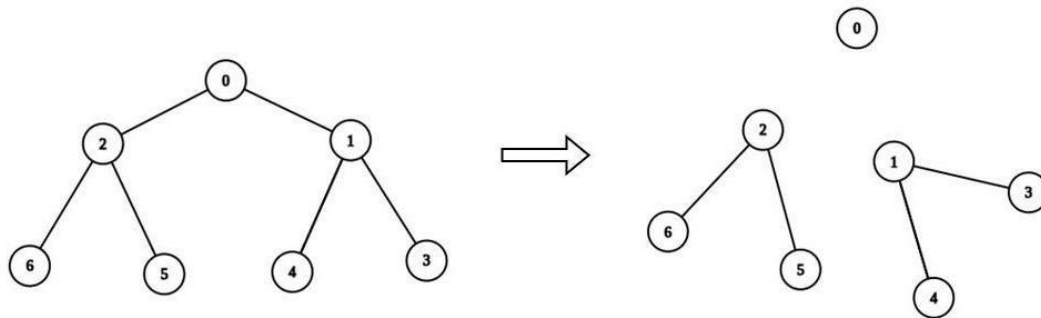
Explanation:

We remove the edge connecting node 1 with 2. The resulting split is valid because:

- The value of the component containing nodes 1 and 3 is `values[1] + values[3] = 12`.
- The value of the component containing nodes 0, 2, and 4 is `values[0] + values[2] + values[4] = 6`.

It can be shown that no other valid split has more than 2 connected components.

Example 2:



Input: $n = 7$, $\text{edges} = [[0,1],[0,2],[1,3],[1,4],[2,5],[2,6]]$, $\text{values} = [3,0,6,1,5,2,1]$, $k = 3$

Output: 3

Explanation:

We remove the edge connecting node 0 with 2, and the edge connecting node 0 with 1.

The resulting split is valid because:

- The value of the component containing node 0 is $\text{values}[0] = 3$.
- The value of the component containing nodes 2, 5, and 6 is $\text{values}[2] + \text{values}[5] + \text{values}[6] = 9$.
- The value of the component containing nodes 1, 3, and 4 is $\text{values}[1] + \text{values}[3] + \text{values}[4] = 6$.

It can be shown that no other valid split has more than 3 connected components.

Constraints:

$1 \leq n \leq 3 * 10^4$

$\text{edges.length} == n - 1$

$\text{edges}[i].\text{length} == 2$

$0 \leq a_i, b_i < n$

$\text{values.length} == n$

$0 \leq \text{values}[i] \leq 10^9$

$1 \leq k \leq 10^9$

Sum of values is divisible by k .

The input is generated such that edges represents a valid tree.

Implementation/Code:

```
#include <iostream> //Programming in C++
#include <vector>
using namespace std;

class Solution {
public:
    int maxKDivisibleComponents(int n, vector<vector<int>>& edges, vector<int>& values, int k) {
        vector<vector<int>> tree(n);
        for (auto& edge : edges) {
            tree[edge[0]].push_back(edge[1]);
            tree[edge[1]].push_back(edge[0]);
        }
    }
};
```

```
    }

    int components = 0;
    dfs(0, -1, tree, values, k, components);
    return components;
}

private:
    int dfs(int node, int parent, vector<vector<int>>& tree, vector<int>& values, int k,
    int& components) {
        int sum = values[node];
        for (int child : tree[node]) {
            if (child != parent) {
                sum += dfs(child, node, tree, values, k, components);
            }
        }

        if (sum % k == 0) {
            components++;
            return 0; // Reset sum for this component
        }

        return sum;
    }
};

int main() {
    Solution sol;

    int n1 = 5;
    vector<vector<int>> edges1 = {{0, 2}, {1, 2}, {1, 3}, {2, 4}};
    vector<int> values1 = {1, 8, 1, 4, 4};
    int k1 = 6;
    cout << sol.maxKDivisibleComponents(n1, edges1, values1, k1) << endl; // Expected:
2

    int n2 = 7;
    vector<vector<int>> edges2 = {{0, 1}, {0, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}};
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
vector<int> values2 = {3, 0, 6, 1, 5, 2, 1};  
int k2 = 3;  
cout << sol.maxKDivisibleComponents(n2, edges2, values2, k2) << endl; // Expected:  
3  
  
return 0;  
}
```

Output:

```
input  
2  
3  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```