<u>DOMAIN WINTER CAMP</u>

<u>DAY - 6</u>

Student Name: Kethrin Naharwal **UID:** 22BCS12653

Branch: BE-CSE **Section:** 22BCS_FL_IOT-603

DSA Questions(Trees)

Very Easy:

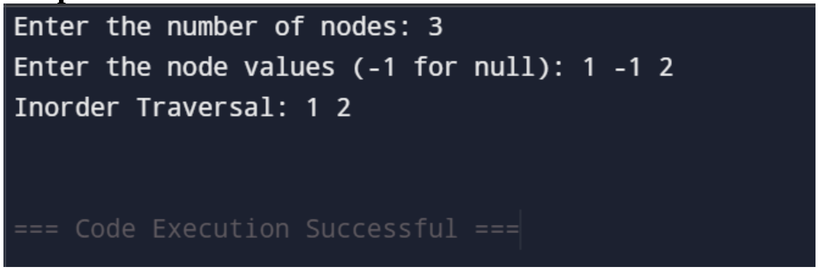
1. Binary Tree Inorder Traversal

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

Input:

```
#include <iostream>
#include <vector>
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) {}
};

// Helper function for inorder traversal
void inorderHelper(TreeNode* root, vector<int>& result) {
    if (root == nullptr) return;
    inorderHelper(root->left, result);  // Traverse left subtree
    result.push_back(root->val);       // Visit node
    inorderHelper(root->right, result); // Traverse right subtree
}

// Function to return the inorder traversal
vector<int> inorderTraversal(TreeNode* root) {
    vector<int> result;
    inorderHelper(root, result);
    return result;
}

// Utility function to build a binary tree
TreeNode* buildTree(vector<int>& nodes, int index) {
    if (index >= nodes.size() || nodes[index] == -1) return nullptr;
    TreeNode* root = new TreeNode(nodes[index]);
    root->left = buildTree(nodes, 2 * index + 1);  // Left child
    root->right = buildTree(nodes, 2 * index + 2); // Right child
    return root;
}

int main() {
    int n;
    cout << "Enter the number of nodes: ";
    cin >> n;

    cout << "Enter the node values (-1 for null): ";
    vector<int> nodes(n);
    for (int i = 0; i < n; i++) {
        cin >> nodes[i];
    }

    TreeNode* root = buildTree(nodes, 0); // Build the tree
    vector<int> result = inorderTraversal(root);

    cout << "Inorder Traversal: ";
    for (int val : result) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

Output:



```
Enter the number of nodes: 3
Enter the node values (-1 for null): 1 -1 2
Inorder Traversal: 1 2


=== Code Execution Successful ===
```

2. Count Complete Tree Nodes

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 2h nodes inclusive at the last level h.

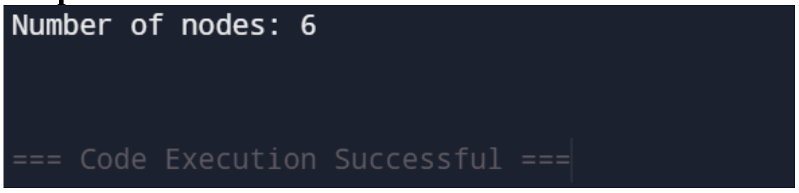
Design an algorithm that runs in less than O(n) time complexity.

Solution:

```
#include <iostream>
#include <cmath>
```

```
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 countNodes(TreeNode* root) {
        if (!root) return 0;

        // Helper function to compute tree height
        auto getHeight = [](TreeNode* node) -> int {
            int height = 0;
            while (node) {
                ++height;
                node = node->left;
            }
            return height;
        };

        int leftHeight = getHeight(root->left);
        int rightHeight = getHeight(root->right);

        if (leftHeight == rightHeight) {
            // Left subtree is a full binary tree
            return (1 << leftHeight) + countNodes(root->right);
        } else {
            // Right subtree is a full binary tree
            return (1 << rightHeight) + countNodes(root->left);
        }
    }
};

// Example Usage
int main() {
    // Tree: [1,2,3,4,5,6]
    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);

    Solution solution;
    cout << "Number of nodes: " << solution.countNodes(root) << endl; // Output: 6

    return 0;
}
```

Output:



```
Number of nodes: 6



=== Code Execution Successful ===
```

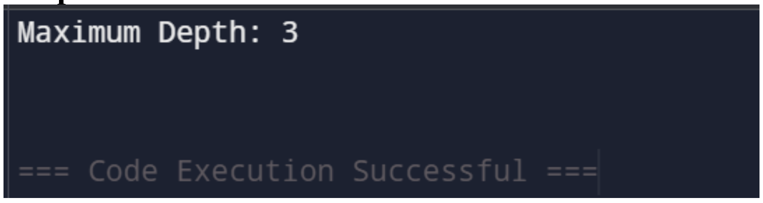
3. Binary Tree - Find Maximum Depth

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.

Solution:

```
#include <iostream>
#include <algorithm>
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 maxDepth(TreeNode* root) {
        if (!root) return 0; // Base case: empty tree has depth 0
        int leftDepth = maxDepth(root->left);   // Recurse on left subtree
        int rightDepth = maxDepth(root->right); // Recurse on right subtree
        return 1 + max(leftDepth, rightDepth);  // Add 1 for the current node
    }
};

// Example Usage
int main() {
    // Tree: [3, 9, 20, null, null, 15, 7]
    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);

    Solution solution;
    cout << "Maximum Depth: " << solution.maxDepth(root) << endl; // Output: 3

    return 0;
}
```

Output:



```
Maximum Depth: 3


=== Code Execution Successful ===
```

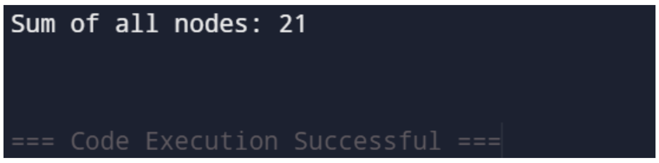
5. Binary Tree - Sum of All Nodes

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

Solution:

```
#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) {}
};

// Function to calculate the sum of all nodes in a binary tree
int sumOfAllNodes(TreeNode* root) {
    if (root == nullptr) return 0; // Base case: if the node is null, return 0
    return root->val + sumOfAllNodes(root->left) + sumOfAllNodes(root->right);
}

// Helper function to create a sample binary tree
TreeNode* createSampleTree() {
    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 = createSampleTree();
    cout << "Sum of all nodes: " << sumOfAllNodes(root) << endl;
    return 0;
}
```

Output:



```
Sum of all nodes: 21


=== Code Execution Successful ===
```

Easy:

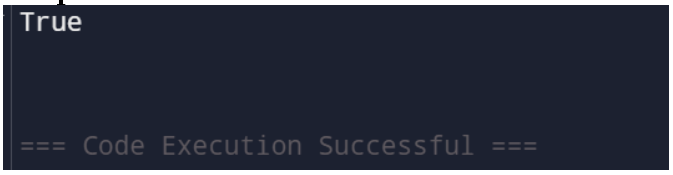
1. Same Tree

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

Solution:

```
#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:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if (!p && !q) return true;    // Both nodes are null
        if (!p || !q) return false;   // One node is null, the other is not
        if (p->val != q->val) return false; // Values don't match
        // Check left and right subtrees recursively
        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};

int main() {
    // Tree p: [1, 2, 3]
    TreeNode* p = new TreeNode(1);
    p->left = new TreeNode(2);
    p->right = new TreeNode(3);

    // Tree q: [1, 2, 3]
    TreeNode* q = new TreeNode(1);
    q->left = new TreeNode(2);
    q->right = new TreeNode(3);

    Solution solution;
    cout << (solution.isSameTree(p, q) ? "True" : "False") << endl;

    return 0;
}
```

Output:



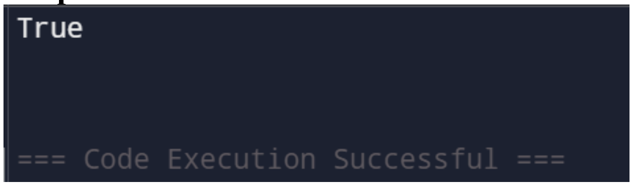
```
True


=== Code Execution Successful ===
```

2. Symmetric Tree

Input:

```
#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:
    bool isMirror(TreeNode* t1, TreeNode* t2) {
        if (!t1 && !t2) return true;    // Both subtrees are null
        if (!t1 || !t2) return false;   // One subtree is null, the other is not
        if (t1->val != t2->val) return false; // Values don't match
        // Check mirroring in the subtrees
        return isMirror(t1->left, t2->right) && isMirror(t1->right, t2->left);
    }

    bool isSymmetric(TreeNode* root) {
        if (!root) return true;  // An empty tree is symmetric
        return isMirror(root->left, root->right);
    }
};

int main() {
    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);

    Solution solution;
    cout << (solution.isSymmetric(root) ? "True" : "False") << endl;

    return 0;
}
```

Output:



```
True


=== Code Execution Successful ===
```

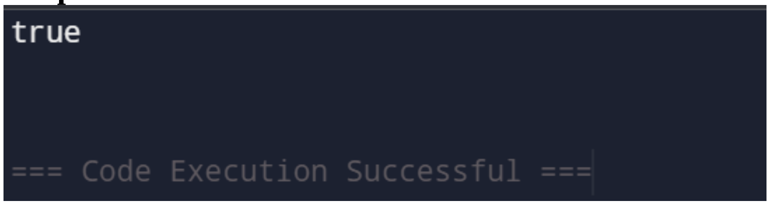
5. Path Sum

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.

Solution:

```
#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(NULL), right(NULL) {}
};

class Solution {
public:
    bool hasPathSum(TreeNode* root, int targetSum) {
        if (!root) {
            return false;  // If the tree is empty
        }

        // If we reach a leaf node, check if the sum matches
        if (!root->left && !root->right) {
            return targetSum == root->val;
        }

        // Recursively check the left and right subtrees with the updated sum
        return hasPathSum(root->left, targetSum - root->val) ||
               hasPathSum(root->right, targetSum - root->val);
    }
};

int main() {
    // Example 1
    TreeNode* root1 = new TreeNode(5);
    root1->left = new TreeNode(4);
    root1->right = new TreeNode(8);
    root1->left->left = new TreeNode(11);
    root1->left->left->left = new TreeNode(7);
    root1->left->left->right = new TreeNode(2);
    root1->right->left = new TreeNode(13);
    root1->right->right = new TreeNode(4);
    root1->right->right->right = new TreeNode(1);

    Solution solution;
    cout << boolalpha;  // To print true/false instead of 1/0
    cout << solution.hasPathSum(root1, 22) << endl;

    return 0;
}
```

Output:



```
true


=== Code Execution Successful ===
```

Medium:

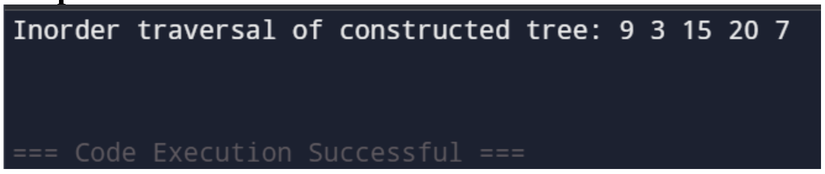
1. Construct Binary Tree from Preorder and Inorder Traversal

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

Solution:

```
#include <iostream>
#include <unordered_map>
#include <vector>
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:
    unordered_map<int, int> inorderIndexMap; // Map to store index of values in
inorder

    TreeNode* buildTreeHelper(vector<int>& preorder, int preStart, int preEnd,
                              vector<int>& inorder, int inStart, int inEnd) {
        if (preStart > preEnd || inStart > inEnd) return nullptr;

        // The first element in preorder is the root
        int rootVal = preorder[preStart];
        TreeNode* root = new TreeNode(rootVal);

        // Find the position of the root in inorder
        int inRootIndex = inorderIndexMap[rootVal];
        int leftSubtreeSize = inRootIndex - inStart;

        // Recursively build the left and right subtrees
        root->left = buildTreeHelper(preorder, preStart + 1, preStart + leftSubtreeSize,
                                     inorder, inStart, inRootIndex - 1);
        root->right = buildTreeHelper(preorder, preStart + leftSubtreeSize + 1, preEnd,
                                      inorder, inRootIndex + 1, inEnd);

        return root;
    }

    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        // Build a map for quick lookup of indices in inorder
        for (int i = 0; i < inorder.size(); ++i) {
            inorderIndexMap[inorder[i]] = i;
        }

        return buildTreeHelper(preorder, 0, preorder.size() - 1,
                               inorder, 0, inorder.size() - 1);
    }
};

// Helper function to print the tree (inorder traversal)
void printInorder(TreeNode* root) {
    if (!root) return;
    printInorder(root->left);
    cout << root->val << " ";
    printInorder(root->right);
}

// Example Usage
int main() {
    Solution solution;

    // Input example 1
    vector<int> preorder = {3, 9, 20, 15, 7};
    vector<int> inorder = {9, 3, 15, 20, 7};

    TreeNode* root = solution.buildTree(preorder, inorder);

    cout << "Inorder traversal of constructed tree: ";
    printInorder(root); // Output: 9 3 15 20 7
    cout << endl;

    return 0;
}
```

Output:



```
Inorder traversal of constructed tree: 9 3 15 20 7



=== Code Execution Successful ===
```

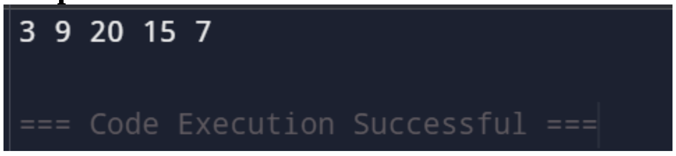
2. Construct Binary Tree from Inorder and Postorder Traversal

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.

Input:

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

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

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

private:
    TreeNode* build(vector<int>& inorder, vector<int>& postorder, int inStart, int
inEnd, int postIndex) {
        if (inStart > inEnd) return nullptr;

        int rootVal = postorder[postIndex];
        TreeNode* root = new TreeNode(rootVal);

        int inRootIndex = find(inorder.begin(), inorder.end(), rootVal) - inorder.begin();

        root->right = build(inorder, postorder, inRootIndex + 1, inEnd, postIndex - 1);
        root->left = build(inorder, postorder, inStart, inRootIndex - 1, postIndex - (inEnd
- inRootIndex) - 1);

        return root;
    }
};

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

int main() {
    vector<int> inorder = {9, 3, 15, 20, 7};
    vector<int> postorder = {9, 15, 7, 20, 3};

    Solution sol;
    TreeNode* root = sol.buildTree(inorder, postorder);

    printTree(root);

    return 0;
}
```

Output:



```
3 9 20 15 7


=== Code Execution Successful ===
```

4. Sum Root to Leaf Numbers

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.

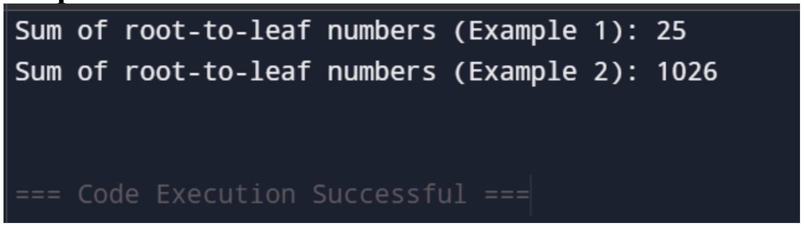
For example, the root-to-leaf path 1 -> 2 -> 3 represents the number 123.
Return the total sum of all root-to-leaf numbers. Test cases are generated so that the answer will fit in a 32-bit integer.

A leaf node is a node with no children.

Input:

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

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

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);
    }
};

int main() {
    // Example 1
    TreeNode* root1 = new TreeNode(1);
    root1->left = new TreeNode(2);
    root1->right = new TreeNode(3);

    // Example 2
    TreeNode* root2 = new TreeNode(4);
    root2->left = new TreeNode(9);
    root2->right = new TreeNode(0);
    root2->left->left = new TreeNode(5);
    root2->left->right = new TreeNode(1);

    Solution solution;
    cout << "Sum of root-to-leaf numbers (Example 1): " <<
solution.sumNumbers(root1) << endl;
    cout << "Sum of root-to-leaf numbers (Example 2): " <<
solution.sumNumbers(root2) << endl;

    return 0;
}
```

Output:



```
Sum of root-to-leaf numbers (Example 1): 25
Sum of root-to-leaf numbers (Example 2): 1026


=== Code Execution Successful ===
```

<u>Hard:</u>

1. Binary Tree Right Side View

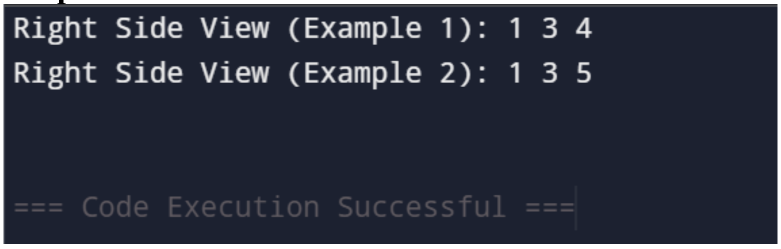
Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

Solution:

```
#include <iostream>
#include <vector>
#include <queue>
```

```
using namespace std;

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

class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
        vector<int> result;
        if (!root) return result;

        queue<TreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            int levelSize = q.size();
            for (int i = 0; i < levelSize; ++i) {
                TreeNode* node = q.front();
                q.pop();

                if (i == levelSize - 1) {
                    result.push_back(node->val);
                }

                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
        }

        return result;
    }
};

int main() {
    TreeNode* root1 = new TreeNode(1);
    root1->left = new TreeNode(2);
    root1->right = new TreeNode(3);
    root1->left->right = new TreeNode(5);
    root1->right->right = new TreeNode(4);

    TreeNode* root2 = new TreeNode(1);
    root2->left = new TreeNode(2);
    root2->right = new TreeNode(3);
    root2->left->left = new TreeNode(4);
    root2->right->right = new TreeNode(5);

    Solution solution;
    vector<int> result1 = solution.rightSideView(root1);
    vector<int> result2 = solution.rightSideView(root2);
    cout << "Right Side View (Example 1): ";
    for (int val : result1) cout << val << " ";
    cout << endl;

    cout << "Right Side View (Example 2): ";
    for (int val : result2) cout << val << " ";
    cout << endl;

    return 0;
}
```

Output:



```
Right Side View (Example 1): 1 3 4
Right Side View (Example 2): 1 3 5


=== Code Execution Successful ===
```

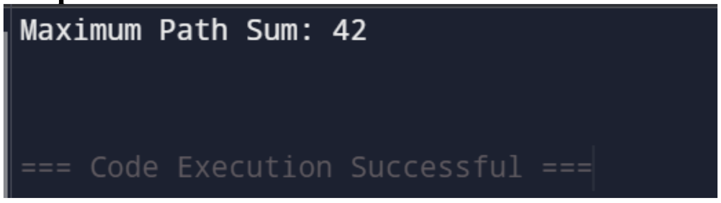
2. Binary Tree Maximum Path Sum

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.
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.

Solution:

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

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

class Solution {
public:
    int maxPathSum(TreeNode* root) {
        int maxSum = INT_MIN;
        maxPathSumHelper(root, maxSum);
        return maxSum;
    }

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

        int left = max(0, maxPathSumHelper(root->left, maxSum));  // Only add positive
contributions
        int right = max(0, maxPathSumHelper(root->right, maxSum)); // Only add
positive contributions

        int currentPathSum = root->val + left + right; // Path passing through the root

        maxSum = max(maxSum, currentPathSum);  // Update the global maximum path
sum

        return root->val + max(left, right);  // Return the max sum path including the
current node
    }
};

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 solution;
    int result = solution.maxPathSum(root);

    cout << "Maximum Path Sum: " << result << endl;

    return 0;
}
```

Output:



```
Maximum Path Sum: 42


=== Code Execution Successful ===
```

Very Hard :

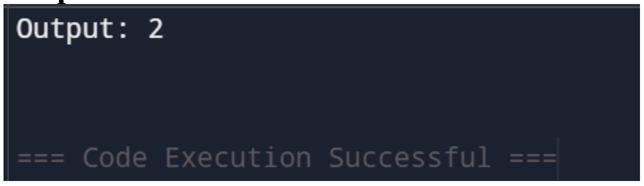
1. Count Paths That Can Form a Palindrome in a Tree

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.

Solution:
```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;

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

        unordered_map<int, int> freq;
        int result = 0;

        // DFS function to explore each path
        dfs(0, tree, s, freq, result);

        return result;
    }

private:
    void dfs(int node, vector<vector<int>>& tree, string& s, unordered_map<int,
int>& freq, int& result) {
        // Count the character for the current node
        int bit = 1 << (s[node] - 'a');
        freq[bit]++;

        // Count the number of valid pairs in the current state
        if (freq[bit] > 1) {
            result += freq[bit] - 1;
        }

        // Explore the tree recursively
        for (int child : tree[node]) {
            dfs(child, tree, s, freq, result);
        }

        // Backtrack and update the frequency map
        freq[bit]--;
    }
};

int main() {
    Solution solution;

    vector<int> parent = {-1, 0, 0, 1, 1, 2};
    string s = "acaabc";
    int n = parent.size();
    int result = solution.countPalindromePaths(n, parent, s);
    cout << "Output: " << result << endl;


    return 0;
}
```

Output:



```
Output: 2


=== Code Execution Successful ===
```

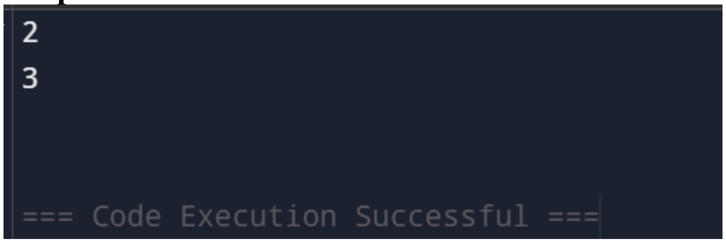
2. Maximum Number of K-Divisible Components

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 ith node, and an integer k.
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.
Return the maximum number of components in any valid split.

Solution:

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

class Solution {
public:
    int maxKDivisibleComponents(int n, vector<vector<int>>& edges, vector<int>&
values, int k) {
        // Create adjacency list for the tree
        adj.resize(n);
        for (auto& edge : edges) {
            adj[edge[0]].push_back(edge[1]);
            adj[edge[1]].push_back(edge[0]);
        }

        // Initialize visited array and the result
        visited.assign(n, false);
        result = 0;
        // Perform DFS to find the number of valid components
        dfs(0, k, values);

        return result;
    }

private:
    vector<vector<int>> adj;
    vector<bool> visited;
    int result;

    // DFS function to calculate the subtree sum and check if we can split
    int dfs(int node, int k, vector<int>& values) {
        visited[node] = true;
        int sum = values[node];  // Start with the value of the current node

        // Explore all neighbors
        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                sum += dfs(neighbor, k, values);
            }
        }

        // If the sum of the current component is divisible by k, we can split here
        if (sum % k == 0) {
            result++;  // We can create a valid component by cutting this subtree
            return 0;  // Reset the sum to 0 because we've "cut" the subtree
        }

        return sum;  // Return the sum of the current subtree
    }
};

int main() {
    Solution solution;

    // Test case 1
    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 << solution.maxKDivisibleComponents(n1, edges1, values1, k1) << endl;

    // Test case 2
    int n2 = 7;
    vector<vector<int>> edges2 = {{0, 1}, {0, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}};
    vector<int> values2 = {3, 0, 6, 1, 5, 2, 1};
    int k2 = 3;
    cout << solution.maxKDivisibleComponents(n2, edges2, values2, k2) << endl;
    return 0;
}
```

Output:



```
2
3


=== Code Execution Successful ===
```

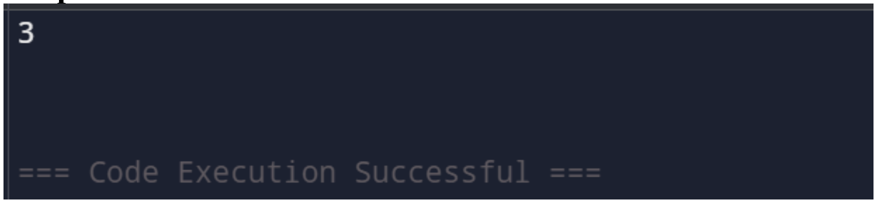
3. Count Number of Possible Root Nodes

Alice has an undirected tree with n nodes labeled from 0 to n - 1. The tree is represented as 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.
Alice wants Bob to find the root of the tree. She allows Bob to make several guesses about her tree. In one guess, he does the following:
Chooses two distinct integers u and v such that there exists an edge [u, v] in the tree.
He tells Alice that u is the parent of v in the tree.
Bob's guesses are represented by a 2D integer array guesses where guesses[j] = [uj, vj] indicates Bob guessed uj to be the parent of vj.
Alice being lazy, does not reply to each of Bob's guesses, but just says that at least k of his guesses are true.
Given the 2D integer arrays edges, guesses and the integer k, return the number of possible nodes that can be the root of Alice's tree. If there is no such tree, return 0.

Input:

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

class Solution {
public:
    int rootCount(int n, vector<vector<int>>& edges, vector<vector<int>>& guesses,
int k) {
        vector<vector<int>> tree(n);
        vector<int> correctGuesses(n, 0);
        unordered_map<int, unordered_map<int, bool>> guessMap;

        // Build the tree
        for (auto& edge : edges) {
            tree[edge[0]].push_back(edge[1]);
            tree[edge[1]].push_back(edge[0]);
        }

        // Mark all guesses as correct or incorrect
        for (auto& guess : guesses) {
            guessMap[guess[0]][guess[1]] = true;
        }
        // Use DFS to check all possible roots and count valid guesses
        int result = 0;
        for (int root = 0; root < n; ++root) {
            int count = 0;
            dfs(root, -1, tree, guessMap, count);
            if (count >= k) {
                result++;
            }
        }

        return result;
    }

private:
    void dfs(int node, int parent, vector<vector<int>>& tree, unordered_map<int,
unordered_map<int, bool>>& guessMap, int& count) {
        for (int child : tree[node]) {
            if (child != parent) {
                if (guessMap[node][child]) {
                    count++;
                }
                dfs(child, node, tree, guessMap, count);
            }
        }
    }
};

int main() {
    Solution solution;

    vector<vector<int>> edges = {{0, 1}, {1, 2}, {1, 3}, {4, 2}};
    vector<vector<int>> guesses = {{1, 3}, {0, 1}, {1, 0}, {2, 4}};
    int k = 3;
    int result = solution.rootCount(5, edges, guesses, k);
    cout << result << endl;


    return 0;
}
```

Output:



```
3



=== Code Execution Successful ===
```