Name: Kaushik Dhruv Alok

UID : 22BCS10210

Section : 901_KPIT(A)

1. 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.

Answer:

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

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

TreeNode* buildTree(const vector<int>& values)
   {  if (values.empty() || values[0] == -1) return
   NULL;

   TreeNode* root = new TreeNode(values[0]);
   queue<TreeNode*> q;
   q.push(root);

   int i = 1;
   while (!q.empty() && i < values.size())
      { TreeNode* node = q.front();
      q.pop();

      if (values[i] != -1) {
         node->left = new TreeNode(values[i]);
```

```
            q.push(node->left);
        }
        i++;

        if (i < values.size() && values[i] != -1)
            { node->right = new
            TreeNode(values[i]); q.push(node-
            >right);
        } i+
        +;
    }
    return root;
}

int getHeight(TreeNode* node)
    { int h = 0;
    while (node)
        { h++;
        node = node->left;
    }
    return h;
}

int countNodes(TreeNode* root)
    { if (!root) return 0;
    int lh = getHeight(root->left);
    int rh = getHeight(root->right);
    if (lh == rh)
        return (1 << lh) + countNodes(root->right);
    else
        return (1 << rh) + countNodes(root->left);
}

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

    vector<int> values(n);
    cout << "Enter the tree values in level order (-1 for NULL): ";
    for (int i = 0; i < n; i++) {
        cin >> values[i];
    }
```

```
TreeNode* root = buildTree(values);
```

```
    cout << "Total nodes: " << countNodes(root) << endl;

    return 0;
}
```

2. 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.

Answer:

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

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

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

    int rootVal = preorder[preStart];
    TreeNode* root = new TreeNode(rootVal);

    int inRoot = inMap[rootVal];
    int numsLeft = inRoot - inStart;

    root->left = buildTreeHelper(preorder, preStart + 1, preStart + numsLeft,
                                 inorder, inStart, inRoot - 1, inMap);
    root->right = buildTreeHelper(preorder, preStart + numsLeft + 1, preEnd,
                                  inorder, inRoot + 1, inEnd, inMap);

    return root;
}
```

```
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder)
    { unordered_map<int, int> inMap;
    for (int i = 0; i < inorder.size(); i++)
        { inMap[inorder[i]] = i;
    }
    return buildTreeHelper(preorder, 0, preorder.size() - 1, inorder, 0, inorder.size() - 1,
inMap);
}

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

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

    vector<int> preorder(n), inorder(n);
    cout << "Enter preorder traversal: ";
    for (int i = 0; i < n; i++) cin >> preorder[i];

    cout << "Enter inorder traversal: ";
    for (int i = 0; i < n; i++) cin >> inorder[i];

    TreeNode* root = buildTree(preorder, inorder);

    cout << "Tree in level order: ";
    printTree(root);
    cout << endl;

    return 0;
}
```

3. **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.

Answer:

```
#include <iostream>
#include <vector>
#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;
      maxGain(root, maxSum);
      return maxSum;
   }

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

      int leftGain = max(maxGain(node->left, maxSum), 0);
      int rightGain = max(maxGain(node->right, maxSum), 0);

      int currentPathSum = node->val + leftGain + rightGain;

      maxSum = max(maxSum, currentPathSum);

      return node->val + max(leftGain, rightGain);
   }
};

TreeNode* buildTree(const vector<int>& values)
   {  if (values.empty() || values[0] == -1) return
   NULL;
```

```
    TreeNode* root = new TreeNode(values[0]);
    vector<TreeNode*> nodes = {root};
    int index = 1;

    for (TreeNode* node : nodes)
        { if (!node) continue;

        if (index < values.size() && values[index] != -1) {
            node->left = new TreeNode(values[index]);
            nodes.push_back(node->left);
        }
        index++;

        if (index < values.size() && values[index] != -1) {
            node->right = new TreeNode(values[index]);
            nodes.push_back(node->right);
        }
        index++;
    }

    return root;
}

int main() {
    vector<int> values = {1, 2, 3};

    TreeNode* root = buildTree(values);
    Solution sol;

    cout << "Maximum Path Sum: " << sol.maxPathSum(root) << endl;

    return 0;
}
```

4. 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.

Answer:

```
#include <iostream>
#include <vector>
#include <unordered_map>
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> freqMap;
      freqMap[0] = 1;
      int result = 0;

      function<void(int, int)> dfs = [&](int node, int mask)
         { mask ^= (1 << (s[node] - 'a'));
         result += freqMap[mask];
         for (int i = 0; i < 26; i++) {
            result += freqMap[mask ^ (1 << i)];
         }
         freqMap[mask]++;
         for (int child : tree[node])
            { dfs(child, mask);
         }
         freqMap[mask]--;
      };

      dfs(0, 0);
      return result;
   }
};

int main() {
   vector<int> parent = {-1, 0, 0, 1, 1, 2};
   string s = "acaabc";

   Solution sol;
   cout << sol.countPalindromePaths(parent, s) << endl;
```

```
    return 0;
}
```

5. Number of Good Paths

There is a tree (i.e. a connected, undirected graph with no cycles) consisting of n nodes numbered from 0 to n - 1 and exactly n - 1 edges.
You are given a 0-indexed integer array vals of length n where vals[i] denotes the value of the ith node. You are also given a 2D integer array edges where edges[i] = [ai, bi] denotes that there exists an undirected edge connecting nodes ai and bi.
A good path is a simple path that satisfies the following conditions:
The starting node and the ending node have the same value.
All nodes between the starting node and the ending node have values less than or equal to the starting node (i.e. the starting node's value should be the maximum value along the path).
Return the number of distinct good paths.
Note that a path and its reverse are counted as the same path. For example, 0 -> 1 is considered to be the same as 1 -> 0. A single node is also considered as a valid path.

Answer:

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

class DSU
{ public:
    DSU(int n)
        { parent.resize(n);
        size.resize(n, 1);
        for (int i = 0; i < n; ++i)
            { parent[i] = i;
        }
    }

    int find(int x) {
        if (x != parent[x]) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
```

```
    void unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            if (size[rootX] > size[rootY])
                { parent[rootY] = rootX;
                size[rootX] += size[rootY];
            } else {
                parent[rootX] = rootY;
                size[rootY] += size[rootX];
            }
        }
    }

    int getSize(int x)
        { return
        size[find(x)];
    }

private:
    vector<int> parent, size;
};

class Solution
{ public:
    int countGoodPaths(vector<int>& vals, vector<vector<int>>& edges)
        { int n = vals.size();
        DSU dsu(n);
        vector<vector<int>> adj(n);
        for (auto& edge : edges) {
            adj[edge[0]].push_back(edge[1]);
            adj[edge[1]].push_back(edge[0]);
        }

        vector<pair<int, int>> nodes(n);
        for (int i = 0; i < n; ++i) {
            nodes[i] = {vals[i], i};
        }
        sort(nodes.begin(), nodes.end());

        long long result = 0;
        unordered_map<int, long long> countMap;

        for (auto& [value, node] : nodes)
```

```
{ countMap[dsu.find(node)]++;
```

```
            for (int neighbor : adj[node])
                { if (vals[neighbor] <= value)
                {
                    dsu.unionSets(node, neighbor);
                }
            }
            result += countMap[dsu.find(node)] - 1;
        }

        return result;
    }
};

int main()
    { Solution
    sol;
    vector<int> vals = {1, 3, 2, 1, 3};
    vector<vector<int>> edges = {{0, 1}, {0, 2}, {2, 3}, {2, 4}};
    cout << sol.countGoodPaths(vals, edges) << endl;

    return 0;
}
```