LeetCode Difficult Problems Notes

Contents

1 Tree Problems

1.1 110. Balanced Binary Tree

Problem: Given a binary tree, determine if it is height-balanced. A height-balanced binary tree is defined as a binary tree in which the height of the left and right subtree of every node differs in height by no more than 1.

Approach: Use a bottom-up approach with a helper function that returns the height of the subtree. If any subtree is unbalanced, propagate -1 upwards.

```cpp
/**
 * 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) {}
 * };
 */

// This class checks if a given binary tree is height-balanced.
class Solution {
public:
    /**
     * The main function to check if a tree is balanced.
     * A balanced tree is defined as one where the heights
     * of the two child subtrees of any node never differ by more than one.
     */
    bool isBalanced(TreeNode* root) {
        // If checkHeight returns -1, the tree is not balanced.
        return checkHeight(root) != -1;
    }

    /**
     * Helper function to compute the height of the tree.
     * Returns -1 immediately if an unbalanced subtree is found.
     */
    int checkHeight(TreeNode* root) {
        // An empty subtree is balanced, and its height is 0.
        if (root == nullptr) {
            return 0;
        }

        // Recursively check the height of the left subtree.
        int leftHeight = checkHeight(root->left);
        // If left subtree is unbalanced, propagate failure upwards (-1).
        if (leftHeight == -1) {
            return -1;
        }

        // Recursively check the height of the right subtree.
        int rightHeight = checkHeight(root->right);
        // If right subtree is unbalanced, propagate failure upwards (-1).
        if (rightHeight == -1) {
            return -1;
```

```
48          }
49
50          // If the current node is unbalanced (heights differ by more than 1),
                report unbalanced.
51          if (abs(leftHeight - rightHeight) > 1) {
52              return -1;
53          }
54
55          // If balanced, return height of current subtree.
56          // Height is 1 + maximum height of the two subtrees.
57          return 1 + max(leftHeight, rightHeight);
58      }
59  };
60
61  /*
62    Explanation of the -1 "signal":
63    -------------------------------
64    The function uses -1 as a special value to indicate that
65    an unbalanced subtree has been found. Once -1 is returned
66    by any recursion, all further ancestors also return -1,
67    which efficiently stops further unnecessary checks.
68  */
```

Time Complexity: O(n) - we visit each node once
Space Complexity: O(h) - where h is the height of the tree (worst case O(n) for skewed trees)