110. Balanced Binary Tree

<https://leetcode.com/problems/balanced-binary-tree/>

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as:

a binary tree in which the left and right subtrees of *every* node differ in height by no more than 1.

**Example 1:**

A picture containing text, clipart

Description automatically generated

**Input:** root = [3,9,20,null,null,15,7]

**Output:** true

**Brute Force:**

Solution 1: Top-Down Recursion

We check to see whether the tree is balanced strictly according to the definition of a balanced binary tree:

* the difference between the heights of the two sub trees are not greater than 1.
* both the left sub tree and right sub tree are also balanced.

With the helper function depth(), we could easily write the code:

public class Solution {

public bool isBalanced(TreeNode root) {

if (root == NULL) return true;

int left=depth(root.left);

int right=depth(root.right);

return (Math.abs(left - right) <= 1) &&

(isBalanced(root->left) && isBalanced(root->right));

}

public int depth (TreeNode root) {

if (root == NULL) return 0;

return max (depth(root.left), depth (root.right)) + 1;

}

};

For the current node root, calling depth() for its left and right children actually has to access all of its children, thus the complexity is O(N).

Each call to depth will take O(N) time because it visits all nodes.

We call depth for each node in the tree, therefore the runtime is O(N^2).

**Optimize: Bottom-Up Recursion**

This solution uses postorder DFS in a bottom-up recursive way.

Instead of calling depth() explicitly for each child node from top to bottom, we first go as deep as possible and work our way backwards then return the height of the current node we are working with.

This negates the duplicate work we were doing in Solution 1.

When the subtree of the current node is balanced, the function dfsHeight() returns a non-negative value as the height.

Otherwise -1 is returned.

According to the leftHeight and rightHeight of the two children, the parent node could check if the sub tree is balanced, and decides its return value.

class Solution {

public boolean isBalanced(TreeNode root) {

if(maxHeight(root) == -1) return false;

return true;

}

public int maxHeight(TreeNode root)

{

if(root == null) return 0;

int leftHeight = maxHeight(root.left);

if(leftHeight == -1) return -1;

int rightHeight = maxHeight(root.right);

if(rightHeight == -1) return -1;

if(Math.abs(leftHeight - rightHeight) > 1) return -1;

return 1 + Math.max(leftHeight, rightHeight);

}

}

This takes O(N) time and O(H) space