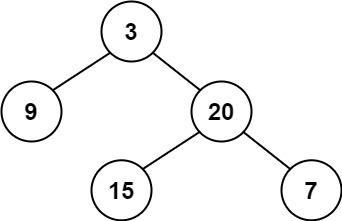
# Question

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.

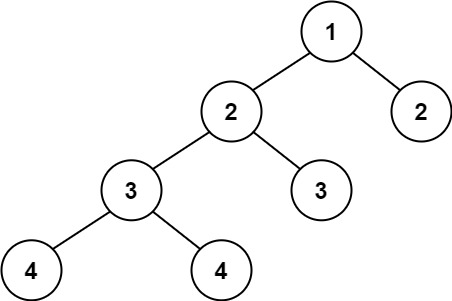
**Example 1:**



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

**Output:** true

**Example 2:**



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

**Output:** false

**Example 3:**

**Input:** root = []

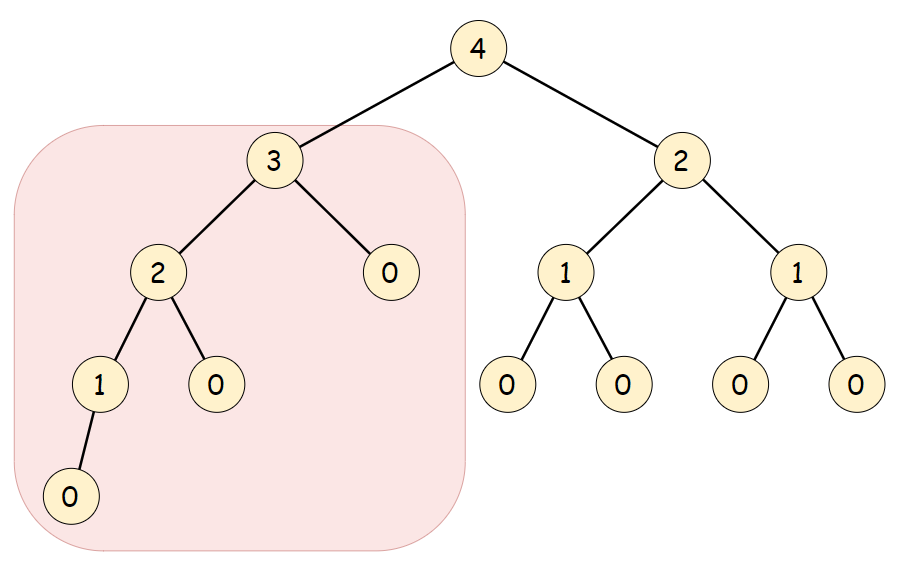
**Output:** true

**Constraints:**

* The number of nodes in the tree is in the range [0, 5000].
* -104 <= Node.val <= 104

# Solution

Given the definition of a balanced tree we know that a tree *T* is not balanced if and only if there is some node *p*∈*T* such that ∣height(*p*.*left*)−height(*p*.*right*)∣>1. The tree below has each node is labeled by its height, as well as the unbalanced subtree highlighted.

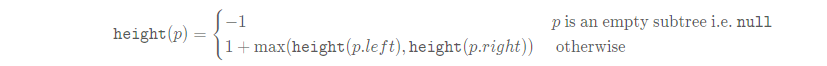


The balanced subtree definition hints at the fact that we should treat each subtree as a subproblem. The question is: in which order should we solve the subproblems?

#### **Approach 1: Top-down recursion**

**Algorithm**

First we define a function \texttt{height}height such that for any node *p*∈*T*



Now that we have a method for determining the height of a tree, all that remains is to compare the height of every node's children. A tree T*T* rooted at r*r* is balanced if and only if the height of its two children are within 1 of each other and the subtrees at each child are also balanced. Therefore, we can compare the two child subtrees' heights then recurse on each one.

isBalanced(root):

if (root == NULL):

return true

if (abs(height(root.left) - height(root.right)) > 1):

return false

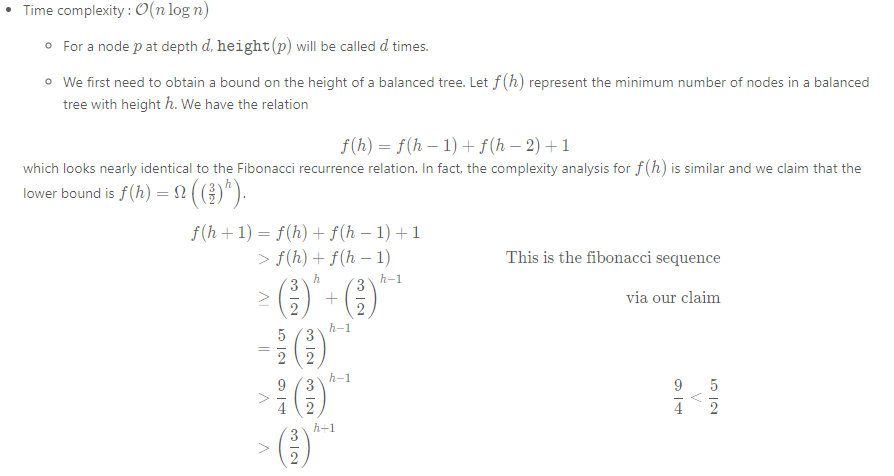
else:

return isBalanced(root.left) && isBalanced(root.right)

|  |
| --- |
| class Solution {  // Recursively obtain the height of a tree. An empty tree has -1 height  private int height(TreeNode root) {  // An empty tree has height -1  if (root == null) {  return -1;  }  return 1 + Math.max(height(root.left), height(root.right));  }  public boolean isBalanced(TreeNode root) {  // An empty tree satisfies the definition of a balanced tree  if (root == null) {  return true;  }  // Check if subtrees have height within 1. If they do, check if the  // subtrees are balanced  return Math.abs(height(root.left) - height(root.right)) < 2  && isBalanced(root.left)  && isBalanced(root.right);  }  }; |

1 / 31

**Complexity Analysis**



Therefore, the height h*h* of a balanced tree is bounded O(log1.5​(*n*)). With this bound we can guarantee that \texttt{height}height will be called on each node \mathcal{O}(\log n)O(log*n*) times.

* + If our algorithm didn't have any early-stopping, we may end up having \mathcal{O}(n^2)O(*n*2) complexity if our tree is skewed since height is bounded by \mathcal{O}(n)O(*n*). However, it is important to note that we stop recursion as soon as the height of a node's children are not within 1. In fact, in the skewed-tree case our algorithm is bounded by \mathcal{O}(n)O(*n*), as it only checks the height of the first two subtrees.
* Space complexity : \mathcal{O}(n)O(*n*). The recursion stack may contain all nodes if the tree is skewed.

**Fun fact**: f(n) = f(n-1) + f(n-2) + 1*f*(*n*)=*f*(*n*−1)+*f*(*n*−2)+1 is known as a [Fibonacci meanders](http://oeis.org/wiki/User:Peter_Luschny/FibonacciMeanders) sequence.

#### **Approach 2: Bottom-up recursion**

**Intuition**

In approach 1, we perform redundant calculations when computing \texttt{height}height. In each call to \texttt{height}height, we require that the subtree's heights also be computed. Therefore, when working top down we will compute the height of a subtree once for every parent. We can remove the redundancy by first recursing on the children of the current node and then using their computed height to determine whether the current node is balanced.

**Algorithm**

We will use the same \texttt{height}height defined in the first approach. The bottom-up approach is a reverse of the logic of the top-down approach since we first check if the child subtrees are balanced before comparing their heights. The algorithm is as follows:

Check if the child subtrees are balanced. If they are, use their heights to determine if the current subtree is balanced as well as to calculate the current subtree's height.

1 / 32

|  |
| --- |
| **// Utility class to store information from recursive calls**  **final class TreeInfo {**  **public final int height;**  **public final boolean balanced;**  **public TreeInfo(int height, boolean balanced) {**  **this.height = height;**  **this.balanced = balanced;**  **}**  **}**  **class Solution {**  **// Return whether or not the tree at root is balanced while also storing**  **// the tree's height in a reference variable.**  **private TreeInfo isBalancedTreeHelper(TreeNode root) {**  **// An empty tree is balanced and has height = -1**  **if (root == null) {**  **return new TreeInfo(-1, true);**  **}**  **// Check subtrees to see if they are balanced.**  **TreeInfo left = isBalancedTreeHelper(root.left);**  **if (!left.balanced) {**  **return new TreeInfo(-1, false);**  **}**  **TreeInfo right = isBalancedTreeHelper(root.right);**  **if (!right.balanced) {**  **return new TreeInfo(-1, false);**  **}**  **// Use the height obtained from the recursive calls to**  **// determine if the current node is also balanced.**  **if (Math.abs(left.height - right.height) < 2) {**  **return new TreeInfo(Math.max(left.height, right.height) + 1, true);**  **}**  **return new TreeInfo(-1, false);**  **}**  **public boolean isBalanced(TreeNode root) {**  **return isBalancedTreeHelper(root).balanced;**  **}**  **};** |

**Complexity Analysis**

* Time complexity : \mathcal{O}(n)O(*n*)

For every subtree, we compute its height in constant time as well as compare the height of its children.

* Space complexity : \mathcal{O}(n)O(*n*). The recursion stack may go up to \mathcal{O}(n)O(*n*) if the tree is unbalanced.