110. Balanced Binary Tree

Oct. 17, 2019 | 21.7K views

Previous Next

Average Rating: 4.09 (23 votes)

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:

Given the following tree `[3,9,20,null,null,15,7]`:

```
    3
   / \
  9  20
    /  \
   15   7
```

Return true.

Example 2:

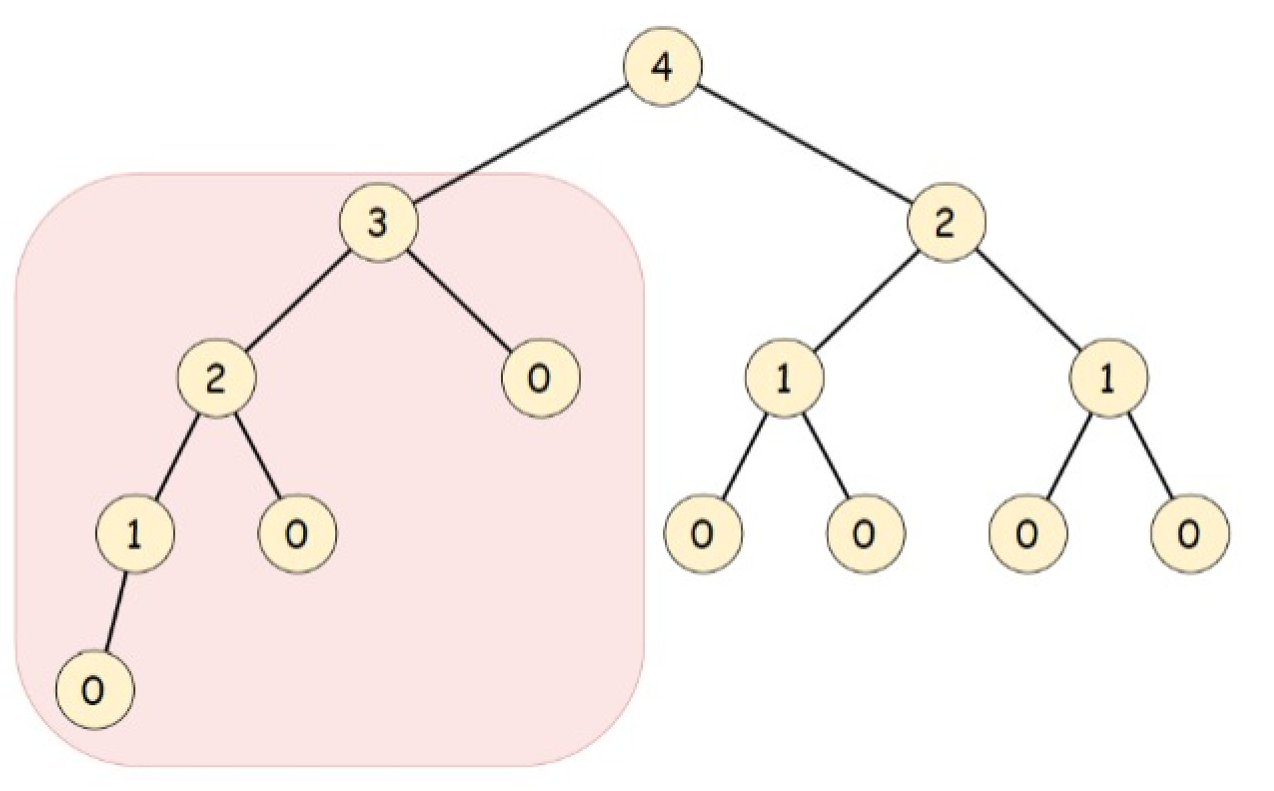
Given the following tree `[1,2,2,3,3,null,null,4,4]`:

```
       1
      / \
     2   2
    / \
   3   3
  / \
 4   4
```

Return false.

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 $p \in T$ such that $|\texttt{height}(p.left) - \texttt{height}(p.right)| > 1$. 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 `height` such that for any node $p \in T$

$$\texttt{height}(p) = \begin{cases} -1 & p \text{ is an empty subtree i.e. } \texttt{null} \\ 1 + \max(\texttt{height}(p.left), \texttt{height}(p.right)) & \text{otherwise} \end{cases}$$

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 rooted at 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)
```

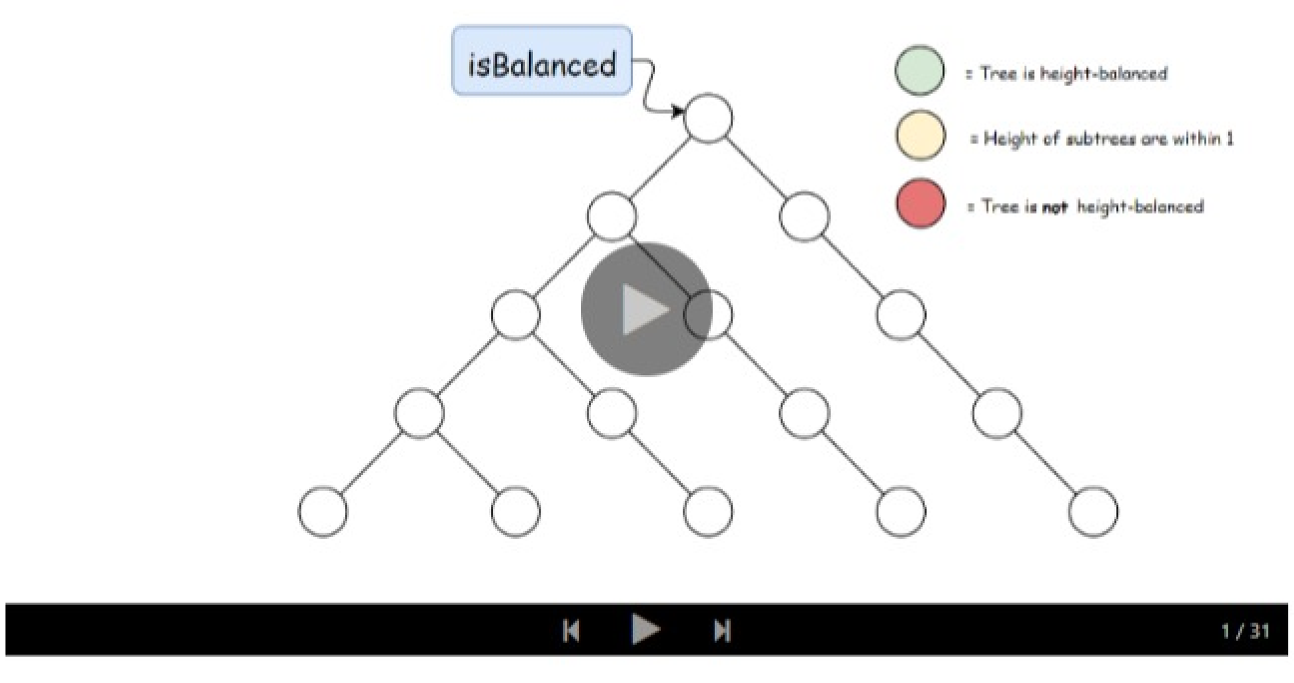
C++ Java Python Copy

```python
class Solution:
    # Compute the tree's height via recursion
    def height(self, root: TreeNode) -> int:
        # An empty tree has height -1
        if not root:
            return -1
        return 1 + max(self.height(root.left), self.height(root.right))

    def isBalanced(self, root: TreeNode) -> bool:
        # An empty tree satisfies the definition of a balanced tree
        if not root:
            return True

        # Check if subtrees have height within 1. If they do, check if the
        # subtrees are balanced
        return abs(self.height(root.left) - self.height(root.right)) < 2 \
            and self.isBalanced(root.left) \
            and self.isBalanced(root.right)

```



Complexity Analysis

- Time complexity : $\mathcal{O}(n \log n)$
  - For a node p at depth d, $\texttt{height}(p)$ will be called d times.
  - We first need to obtain a bound on the height of a balanced tree. Let $f(h)$ represent the minimum number of nodes in a balanced tree with height h. We have the relation

$$f(h) = f(h-1) + f(h-2) + 1$$

  which looks nearly identical to the Fibonacci recurrence relation. In fact, the complexity analysis for $f(h)$ is similar and we claim that the lower bound is $f(h) = \Omega\left(\left(\frac{3}{2}\right)^h\right)$.

  \(\begin{align} f(h+1) &= f(h) + f(h-1) + 1 \\ &> f(h) + f(h-1) & \qquad\qquad \text{This is the fibonacci sequence}\\ &\geq \left(\frac{3}{2}\right)^{h} + \left(\frac{3}{2}\right)^{h-1} & \text{via our claim} \\ &= \frac{5}{2} \left(\frac{3}{2}\right)^{h-1}\\ &> \frac{9}{4} \left(\frac{3}{2}\right)^{h-1} & \frac{9}{4} < \frac{5}{2}\\ &> \left(\frac{3}{2}\right)^{h+1} \end{align} \)

  Therefore, the height h of a balanced tree is bounded by $\mathcal{O}(\log_{1.5}(n))$. With this bound we can guarantee that `height` will be called on each node $\mathcal{O}(\log n)$ times.

  - If our algorithm didn't have any early-stopping, we may end up having $\mathcal{O}(n^2)$ complexity if our tree is skewed since height is bounded by $\mathcal{O}(n)$. 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 $\mathcal{O}(n)$, as it only checks the height of the first two subtrees.
- Space complexity : $\mathcal{O}(n)$. The recursion stack may contain all nodes if the tree is skewed.

Fun fact: $f(n) = f(n-1) + f(n-2) + 1$ is known as a Fibonacci meanders sequence.

Approach 2: Bottom-up recursion

Intuition

In approach 1, we perform redundant calculations when computing `height`. In each call to `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 `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.

C++ Java Python Copy

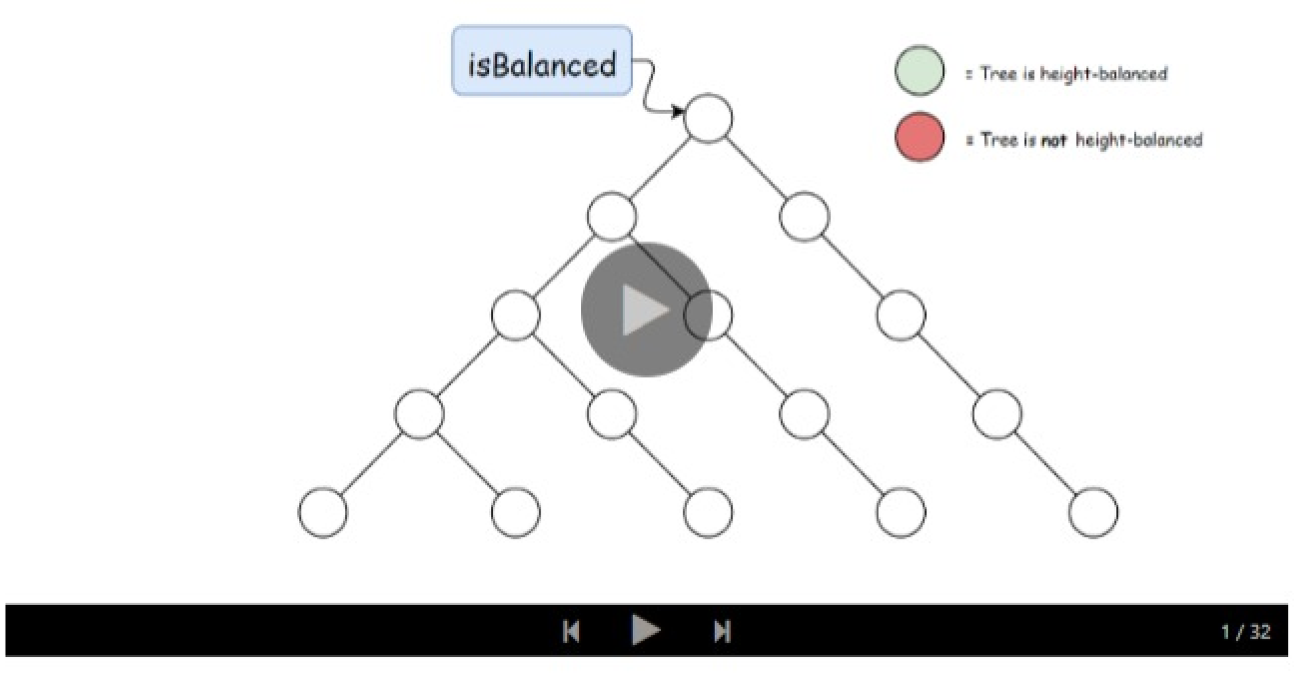
```python
class Solution:
    # Return whether or not the tree at root is balanced while also returning
    # the tree's height
    def isBalancedHelper(self, root: TreeNode) -> (bool, int):
        # An empty tree is balanced and has height -1
        if not root:
            return True, -1

        # Check subtrees to see if they are balanced.
        leftIsBalanced, leftHeight = self.isBalancedHelper(root.left)
        if not leftIsBalanced:
            return False, 0
        rightIsBalanced, rightHeight = self.isBalancedHelper(root.right)
        if not rightIsBalanced:
            return False, 0

        # If the subtrees are balanced, check if the current tree is balanced
        # using their height
        return (abs(leftHeight - rightHeight) < 2), 1 + max(leftHeight, rightHeight)

    def isBalanced(self, root: TreeNode) -> bool:
        return self.isBalancedHelper(root)[0]

```



Complexity Analysis

- Time complexity : $\mathcal{O}(n)$

  For every subtree, we compute its height in constant time as well as compare the height of its children. * Space complexity : $\mathcal{O}(n)$. The recursion stack may go up to $\mathcal{O}(n)$ if the tree is unbalanced.

Rate this article:

Previous Next

Comments: 14

Sort By

Type comment here... (Markdown is supported)

Preview Post

dyckia ★ 199 November 25, 2019 6:54 PM

Here is a much simpler bottom up solution without the extra TreeInfo class.

```
// since the height of a tree is always greater than or equal to 0
// we use -1 as a flag to indicate if the subtree is not balanced
class Solution {
```

Read More

25 Share Reply

SHOW 8 REPLIES

chauhraj ★ 24 November 12, 2019 2:21 PM

In Solution 1 in Complexity Analysis,

> For a node p at depth d, `height(p)` will be called d times.

Why is this true? Isn't in the case of Binary Tree, one will visit the given node only once?

4 Share Reply

SHOW 1 REPLY

miaoyao ★ 32 May 11, 2020 1:33 PM

I use a global variable to control recursion. When we find one subtree is not balanced. We don't need to traversal the rest nodes. The time complexity and space complexity is O(n), because we may traversal the whole tree.

```
class Solution {
```

Read More

1 Share Reply

sriharik ★ 149 May 9, 2020 5:40 AM

There are two ways to solve this problem. One is a top down approach and the other is a bottom up approach.
The top down approach is trivial, simply get the height of the left and right subtree at each node and see if they differ by more than 1.

Read More

1 Share Reply

ab3rd ★ 5 January 26, 2020 10:29 PM

In top-down recursion, I feel the arguments for time complexity and space complexity would be the same. As a result, the space complexity should be O(nlog(n)) as well.

1 Share Reply

lenchen1112 ★ 973 January 5, 2020 4:22 PM

Clean Python 3 bottom-up approach:

```
class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        class NotBalanced(Exception):
```

Read More

1 Share Reply

silviuh1 ★ 0 2 days ago

elegant and easy solution

```
class Solution(object):
    def isBalanced(self, root) -> (int, bool):
        def helper(root):
```

Read More

0 Share Reply

user5064Z ★ 0 June 22, 2020 7:29 AM

Small note that helped me. if recursive calls before conditional check, then its bottom up. If recursive call after conditional check, its top down.

0 Share Reply

Username1604 ★ 42 June 18, 2020 12:52 AM

My 2 cents using bottom up, I used tuple for storing past validation result
and subtree height

```
class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
```

Read More

0 Share Reply

haiming1927 ★ 7 June 4, 2020 5:48 AM

This question should be the medium level....

0 Share Reply

‹ 1 2 ›