Daily Coding Problem #247

Problem

This problem was asked by PayPal.

Given a binary tree, determine whether or not it is height-balanced. A height-balanced binary tree can be defined as one in which the heights of the two subtrees of any node never differ by more than one.

Solution

When analyzing tree problems, the first tool in our toolkit should be recursion.

Suppose we are looking at a tree like this:

```
      3
   4     7
8      5   9
```

Starting at the root node 3, we can first check that its subtrees, starting at 4 and 7, have heights that differ by no more than one. Then we must proceed to each subtree, performing the same task. If after visiting all nodes we determine that the height difference property is always satisfied, we know that the tree is balanced.

To solve this recursively, our base case will be when the root does not exist, in which case

To solve this recursively, our base case will be when the root does not exist, in which case we can say the tree is vacuously balanced. Otherwise, we will ensure that the subtree height difference is less than or equal to one, and call our function again on the subtrees.

```
def get_height(root):
    if not root:
        return 0
    else:
        return 1 + max(get_height(root.left), get_height(root.right))

def is_balanced(root):
    if not root:
        return True
    return is_balanced(root.left) and \


           is_balanced(root.right) and \
           abs(get_height(root.left) - get_height(root.right)) <= 1
```

By calling `is_balanced` twice at each node, we are creating two subproblems of size approximately `N / 2`, and using up to `O(N)` time for our call to `get_height`. Using the master theorem), we can define the time complexity of this recursion as `T(N) = 2 * T(N / 2) + O(N)`. As in merge sort, this resolves to `O(N log N)`.

We can improve on this by combining the two recursive functions into one helper function. When traversing the tree we calculate the height of each node to be one more than the maximum height of its subtrees. If at any point we find that a subtree starting at a given node is unbalanced, or that two subtree heights differ by more than one, we return `-1`.

```
def helper(root):
    if root is None:
        return 0

    left = helper(root.left)
    if left == -1:
        return -1

    right = helper(root.right)
    if right == -1:
        return -1
```

```
        if abs(left - right) > 1:
            return -1

        return max(left, right) + 1

    def is_balanced(root):
        return helper(root) != -1
```

Our helper function generates two subproblems at each node of size `N / 2`, but the time required to combine these subproblems is only `O(1)`, as we are only taking their maximum. Again by the master theorem, we can express the time complexity as `T(N) = 2 * T(N / 2) + O(1)`, which resolves to `O(N)`.