Daily Coding Problem #8

Problem

This problem was asked by Google.

A unival tree (which stands for "universal value") is a tree where all nodes under it have the same value.

Given the root to a binary tree, count the number of unival subtrees.

For example, the following tree has 5 unival subtrees:

```
   0
  / \
 1   0
    / \
   1   0
  / \
 1   1
```

Solution

To start off, we should go through some examples.

```
  a
 / \
a   a
   /\
  a  a
      \
       A
```

This tree has 3 unival subtrees: the two 'a' leaves, and the one 'A' leaf. The 'A' leaf causes all its parents to not be counted as a unival tree.

```
  a
 / \
c   b
   /\
  b  b
       \
        b
```

This tree has 5 unival subtrees: the leaf at 'c', and every 'b'.

We can start off by first writing a function that checks whether a tree is unival or not. Then, perhaps we could use this to count up all the nodes in the tree.

To check whether a tree is a unival tree, we must check that every node in the tree has the same value. To start off, we could define an `is_unival` function that takes in a root to a tree. We would do this recursively with a helper function. Recall that a leaf qualifies as a unival tree.

```
def is_unival(root):
    return unival_helper(root, root.value)

def unival_helper(root, value):
    if root is None:
        return True
    if root.value == value:
        return unival_helper(root.left, value) and unival_helper(root.right, value)
    return False
```

And then our function that counts the number of subtrees could simply use that function:

```
def count_unival_subtrees(root):
    if root is None:
        return 0
    left = count_unival_subtrees(root.left)
    right = count_unival_subtrees(root.right)
    return 1 + left + right if is_unival(root) else left + right
```

However, this runs in O(n^2) time. For each node of the tree, we're evaluating each node in its subtree again as well. We can improve the runtime by starting at the leaves of the tree, and keeping track of the unival subtree count and value as we percolate back up. This should evaluate each node only once, making it run in O(n) time.

```
def count_unival_subtrees(root):
    count, _ = helper(root)
    return count
```

```python
# Also returns number of unival subtrees, and whether it is itself a unival subtree.
def helper(root):
    if root is None:
        return 0, True

    left_count, is_left_unival = helper(root.left)
    right_count, is_right_unival = helper(root.right)
    total_count = left_count + right_count

    if is_left_unival and is_right_unival:
        if root.left is not None and root.value != root.left.value:
            return total_count, False
        if root.right is not None and root.value != root.right.value:
            return total_count, False
        return total_count + 1, True
    return total_count, False
```