Daily Coding Problem #93

Problem

This problem was asked by Apple.

Given a tree, find the largest tree/subtree that is a BST.

Given a tree, return the size of the largest tree/subtree that is a BST.

Solution

One way we can solve this problem is by using the solution to the "determine whether a tree is a valid binary search tree" problem with modifications.

Let's recap the properties of a valid BST. Each node of a BST has the following properties:

- A node's left subtree contains only nodes with keys less than the nodes' key.
- A node's right subtree contains only nodes with keys greater than the nodes' key.
- Both the left and right subtrees must be valid BSTs.

Our solution to `is_bst()` looked like this:

```
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.key = key

def is_bst(root):
    def is_bst_helper(root, min_key, max_key):
        if root is None:
            return True
        if root.key <= min_key or root.key >= max_key:
            return False
        return is_bst_helper(root.left, min_key, root.key) and \
               is_bst_helper(root.right, root.key, max_key)

    return is_bst_helper(root, float('-inf'), float('inf'))
```

We can use this method at each node in our tree, starting at the leaves and returning the size of the tree upwards:

```
def size(root):
    if root is None:
        return 0
    return size(root.left) + size(root.right) + 1
```

```
def largest_bst_subtree(root):
    def helper(root):
        # Returns a tuple of (size, root) of the largest subtree.
        if is_bst(root):
            return (size(root), root)
        return max(helper(root.left), helper(root.right), key=lambda x: x[0])

    return helper(root)[1]
```

The time complexity of this solution is `O(N^2)` in the worse case, since we are doing an `O(N)` traversal for each of the nodes in the tree.

We can improve upon this solution by using a single method to check the validity and find the size of a subtree. To do this, we can revisit the definition of a BST in our `is_bst()` method. Instead of passing the range of valid keys down to the children of the current node, we can return the range of valid keys up to the parent. At the current node, we check whether the key is *less than* the `max_key` of the left subtree or *greater than* the `min_key` of the right subtree. In this way, we can determine both the size and validity in a bottom-up fashion.

```
def largest_bst_subtree(root):
    max_size = 0
    max_root = None
    def helper(root):
        # Returns a tuple of (size, min_key, max_key) of the subtree.
        nonlocal max_size
        nonlocal max_root
        if root is None:
            return (0, float('inf'), float('-inf'))
        left = helper(root.left)
        right = helper(root.right)
        if root.key > left[2] and root.key < right[1]:
```

```
            size = left[0] + right[0] + 1
            if size > max_size:
                max_size = size
                max_root = root
            return (size, min(root.key, left[1]), max(root.key, right[2]))
        else:
            return (0, float('-inf'), float('inf'))

    helper(root)
    return max_root
```

Our solution now has a worst-case time complexity of `O(N)`, where `N` is the number of nodes in the tree.