Daily Coding Problem #115

Problem

This problem was asked by Google.

Given two non-empty binary trees `s` and `t`, check whether tree `t` has exactly the same structure and node values with a subtree of `s`. A subtree of `s` is a tree consists of a node in `s` and all of this node's descendants. The tree `s` could also be considered as a subtree of itself.

Solution

We can solve this problem by solving the subproblem of whether the tree `t` has the same structure and node values of any subtree of `s`. We can implement a helper function `is_equal(s, t)` that checks whether the two trees have the same structure and values. Then, for each node `u` in `s`, we return `True` if `is_equal(u, t)` for any `u`, otherwise we return `False`. The implementation of `is_equal` is straightforward, and takes `O(min(M, N))` time in the worst case, where `M` and `N` are the number of nodes in trees `u` and `t`, respectively.

Once we have the `is_equal(u, t)` function, we can write a recursion for the overall answer. We should return `True` if the current node is equal to `t`, or if either of the subtrees is equal to `t`. The base case should handle an empty subtree.

```
def is_subtree(s, t):
    def is_equal(s, t):
        if s is None and t is None:
            return True
        if s is None or t is None:
            return False
        if s.val != t.val:
            return False
        return is_equal(s.left, t.left) and is_equal(s.right, t.right)

    if s is None:
        return False
    if is_equal(s, t):
        return True
    return is_subtree(s.left, t) or is_subtree(s.right, t)
```

Overall, the time complexity of this algorithm is `O(M * N)` in the worst case, where `M` is the number of nodes in tree `s` and `N` is the number of nodes in tree `t`, since we would perform up to `N` calls to `is_subtree`. The space complexity of this solution is `O(M)`, since the depth of the recursion can go up to the number of nodes of tree `s` in the worst case.

Another way we can solve this problem is by encoding both trees using a pre-order traversal (with `null` markers). We return whether the string representation of subtree `s` is found within the string representation of tree `t`. Since we are using a pre-order traversal, subtrees of `t` will each be found in one contiguous substring, enabling us to search for the substring from the traversal of tree `s`. We must mark `null` pointers during our traversal, as pre-order traversals without these can be ambiguous in how the tree should be reconstructed. We also need to wrap the start and end of the traversal string, to avoid edge cases such as `12` and `1`.

```
def is_subtree(s, t):
    def preorder(root):
        traversal = []
        stack = [root]
        while stack:
            n = stack.pop()
            if n is None:
                traversal.append('.')  # null marker
                continue
            else:
                traversal.append(str(n.val))

            stack.append(n.right)
```

```
                stack.append(n.left)
            return ',' + ','.join(traversal) + ','  # Wrap result

        s_str = preorder(s)
        t_str = preorder(t)
        return t_str in s_str
```

The `preorder(root)` function takes `O(N)` time, and up to `O(N)` space in the worst case. The overall runtime depends on how the method checking for whether the substring is contained is implemented. The Python 3 implementation uses a variant of the Boyer–Moore string search algorithm , which runs in `O(M + N)` time in the average case, and `O(M * N)` in the worst case.

Finally, we could also check for subtree quality by using a hashing algorithm. One frequently-used data structure for comparing trees using hashing is a Merkle tree.