Topic Introduction

Today's PrepLetter dives into a classic family of interview problems: **tree comparison and matching using recursion**.

While you probably already know what a binary tree is, let's quickly define our focus: **Tree Comparison** means examining two (or more) trees to determine if they share a certain structure or content. In coding interviews, this often means checking if two trees are identical, if one is a subtree of another, or if a tree is symmetric (mirrored).

How does it work?
At the heart of these problems is the *recursive structure* of trees. Each tree node can itself be considered a root of a subtree. This recursive nature means problems can often be solved by writing a function that checks the current node, and then recursively checks its children.

When and why is this useful?
Tree comparison comes up often in interviews because it tests your grasp of recursion, edge case handling (like `None` nodes), and your ability to think in terms of tree structure rather than flat data. It's also a great way to show you can break a problem down into smaller, similar subproblems—an essential programming skill!

Quick Example (not using our target problems):
Suppose you want to check if two trees are mirror images of each other. For each node, you'd need to compare the left subtree of one with the right subtree of the other, and vice versa, recursively.

Let's get hands-on with three classic problems that all use this recursive tree matching concept!

Why These 3 Problems?

- **Same Tree:** Checks if two trees are exactly the same (structure and values).
- **Symmetric Tree:** Checks if a tree is a mirror of itself.
- **Subtree of Another Tree:** Checks if one tree is a subtree of another.

They share a recursive "structure matching" approach, but each adds its own twist. Starting with Same Tree builds your foundational recursive thinking. Symmetric Tree is a clever variation. Subtree of Another Tree is the most challenging, as it requires checking for matches at any node, not just the root.

Let's tackle them in that order for the smoothest learning curve!

Problem 1: Same Tree

Problem link: Same Tree (LeetCode 100)

Rephrased Statement:
Given two binary trees, return `True` if they are *identical*—meaning they have the same structure and each corresponding node has the same value. Otherwise, return `False`.

Example Input/Output:

```
Tree p:      1
           /   \
          2     3

Tree q:      1
           /   \
          2     3

Output: True
```

Thought Process:

- Trees are the same if both roots are the same, and their left and right subtrees are also the same.
- If one node is None and the other isn't, they cannot be the same.
- If both nodes are None, they're considered the same at this branch.

Try on paper:

Draw out two trees and compare node by node. Are both structures identical? Do all their values match?

Additional test case:

```
Tree p:      1
           /
          2

Tree q:      1
           /   \
          2     3

Output: False
```

Brute-force approach:

Traverse both trees in any order (preorder, inorder, postorder), and compare nodes at the same positions.

Time Complexity: O(N), where N is the number of nodes (since every node must be compared).

Optimal Approach:

Use recursion—a function that compares the current nodes, and then recursively compares left and right children.

Step-by-step Logic:

- If both nodes are None, return True.
- If only one of the nodes is None, return False.
- If values are different, return False.
- Otherwise, recursively check left and right subtrees.

Python Solution

```
# Definition for a binary tree node.
```

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def isSameTree(p, q):
    # Base case: both nodes are None
    if not p and not q:
        return True
    # If only one is None, trees differ
    if not p or not q:
        return False
    # Values differ
    if p.val != q.val:
        return False
    # Recursively compare left and right subtrees
    return isSameTree(p.left, q.left) and isSameTree(p.right, q.right)
```

Time Complexity: O(N)

Space Complexity: O(H) — where H is the height of the tree, for the recursion stack.

Code Explanation:

- The function checks the two current nodes.
- If both are None, they're equal at this branch.
- If only one is None or values differ, return False.
- Otherwise, keep comparing left and right subtrees.

Test Case Trace:

```
Tree p:      1
           /   \
          2     3

Tree q:      1
           /   \
          2     3
```

- `isSameTree(1,1)` → values equal, check children.
- `isSameTree(2,2)` → values equal, check children (both None).
- `isSameTree(3,3)` → values equal, check children (both None).
- All match, returns True.

Try this test case yourself:

```
Tree p:      1
           /   \
```

```
          2     1

Tree q:      1
           /   \
          1     2

Expected Output: False
```

Take a moment to solve this on your own before jumping into the solution!

Problem 2: Symmetric Tree

Problem link: Symmetric Tree (LeetCode 101)

Problem Statement Rephrased:

Given a binary tree, return `True` if it is symmetric around its center (i.e., it's a mirror of itself). Otherwise, return `False`.

How is this similar/different?

Like Same Tree, we're comparing nodes recursively. The twist: instead of comparing the left child of one tree to the left child of the other, we compare the left child of one to the right child of the other (and vice versa).

Example Input/Output:

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

Output: True
```

Additional test case:

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

Output: False
```

Brute-force:

Could flatten the left and right subtrees and check if one is the reverse of the other.

Time Complexity: Still O(N).

Optimal Approach:

Use a helper function to compare the left and right subtrees as mirror images.

Step-by-step Logic:

- Define a function `isMirror(left, right)`:
  - If both nodes are None, return True.
  - If only one is None, return False.
  - If values differ, return False.
  - Recursively check `left.left` vs `right.right` and `left.right` vs `right.left`.

Pseudocode:

```
function isSymmetric(root):
    if not root: return True
    return isMirror(root.left, root.right)

function isMirror(left, right):
    if not left and not right: return True
    if not left or not right: return False
    if left.val != right.val: return False
    return isMirror(left.left, right.right) and isMirror(left.right, right.left)
```

Test Case Trace:

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

- Compare 2 vs 2 (both sides of root).
- Compare 3 vs 3 and 4 vs 4 recursively.
- All match, returns True.

Try this test case yourself:

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

Expected Output: False
```

Time Complexity: O(N)

Space Complexity: O(H) for recursion.

Problem 3: Subtree of Another Tree

Problem link: Subtree of Another Tree (LeetCode 572)

Problem Statement Rephrased:

Given two binary trees `root` and `subRoot`, return `True` if `subRoot` is a subtree of `root` (there exists a node in `root` such that the subtree rooted at that node is identical to `subRoot`).

What's different?

This time, you must check *every* node in `root` to see if it matches the root of `subRoot`, and if so, check if the subtrees match (just like Same Tree). This combines both previous ideas and is the trickiest of the three.

Example Input/Output:

```
root:      3
          /   \
         4     5
        / \
       1   2

subRoot:    4
           / \
          1   2

Output: True
```

Brute-force:

At every node in `root`, use Same Tree comparison to check if the subtree matches `subRoot`.

Time Complexity: O(M * N), where M is the number of nodes in `root`, N in `subRoot`.

Optimal Approach:

Still involves checking every node, but uses recursion efficiently.

Step-by-step Logic:

- For the current node in `root`:
  - If the subtree rooted here is the same as `subRoot`, return True.
  - Else, recursively check left and right subtrees.

Pseudocode:

```
function isSubtree(root, subRoot):
    if not root: return False
    if isSameTree(root, subRoot): return True
    return isSubtree(root.left, subRoot) or isSubtree(root.right, subRoot)

function isSameTree(p, q):
    ... (as before)
```

Test Case Trace:

```
root:      3
          /   \
         4     5
```

```
      / \
     1   2

subRoot:    4
           / \
          1   2

- At root 3: isSameTree(3,4)? False
- Go left: isSameTree(4,4)? True (subtrees match)
- Return True.
```

Try this case:

```
root:      3
         /   \
        4     5
       / \
      1   2
           \
            0

subRoot:    4
           / \
          1   2

Expected Output: False
```

Time Complexity: O(M * N)

Space Complexity: O(H), H = height of root.

Reflect:

There are more advanced solutions (e.g., tree serialization and KMP string searching) that can reduce the average time, but the recursive solution is the most interview-friendly and helps you master the concept.

Summary and Next Steps

These three problems all center on **recursively comparing trees**. You've seen how the basic technique (Same Tree) evolves into mirror comparison (Symmetric Tree) and then into searching for a subtree anywhere in a larger tree (Subtree of Another Tree).

Key Patterns and Insights:

- When comparing trees, always check base cases: both nodes None (True), only one None (False), values differ (False).
- Recursive structure makes these problems elegant and test your recursion skills.
- Symmetry often means comparing the "opposite" child nodes.
- Subtree problems may require reusing your Same Tree logic.

Common Mistakes:

- Forgetting to check for None before accessing node values.
- Mixing up which children to compare in symmetric or subtree checks.
- Not handling all base cases.

Action List

- Solve all 3 problems on your own—even the one with code provided.
- Try implementing the Symmetric Tree and Subtree problems iteratively or using BFS for practice.
- Explore related problems, like "Invert Binary Tree" or "Maximum Depth of Binary Tree."
- Compare your solution with others for new insights or different styles.
- If you get stuck, review the recursive patterns and dry-run with small trees on paper.

Keep practicing—you’re not just mastering tree comparison, but recursion and structural thinking for all kinds of coding interviews!