Topic Introduction

Today's Concept: Reconstructing Binary Trees from Traversal Arrays

If you're prepping for coding interviews, you've probably seen "construct a binary tree from traversals" more than once. These problems test your ability to deeply understand tree traversals and apply divide-and-conquer thinking—two skills that are gold in technical interviews.

What is Tree Traversal?

A tree traversal is an order in which you visit every node in a binary tree. The three main types are:

- **Preorder (Root, Left, Right)**
- **Inorder (Left, Root, Right)**
- **Postorder (Left, Right, Root)**

You can think of these as different ways to "tour" a tree, each revealing unique information about its structure.

Why is this Useful?

Interviewers love these problems because:

- They test recursion, indices, and array manipulation.
- They require you to "reverse engineer" the tree, given only the breadcrumbs of traversal order.
- They highlight your ability to reason about subarrays, recursion boundaries, and pointer math.

When to Use This Concept?

Whenever you're asked to reconstruct a binary tree from its traversal arrays (like preorder + inorder, or inorder + postorder), this technique is your go-to. It's also handy in serialization/deserialization and advanced tree algorithms.

Simple Example (not from our target problems):

Suppose you have a preorder traversal `[3, 9, 20]` and a tree with only three nodes. If you also had the inorder traversal `[9, 3, 20]`, you could deduce:

- Preorder's first element is always the root (3).
- Inorder tells you nodes to the left of 3 are in the left subtree (9), those to the right are in the right subtree (20).
- You can recursively repeat this for subarrays.

Today, we'll break this idea down using three classic LeetCode problems—each with a different traversal combo.

Why These 3 Problems Belong Together

All three problems challenge you to reconstruct a binary tree using two traversal arrays, but with different combinations (preorder + inorder, inorder + postorder, preorder + postorder). They each require you to:

- Understand how each traversal order encodes the tree's structure.
- Recursively build left and right subtrees by slicing the arrays.
- Master divide-and-conquer recursion and index trickery.

Let’s use this as a springboard to deepen your tree skills.

Problem 1: Construct Binary Tree from Preorder and Inorder Traversal

Problem link: LeetCode 105

Your Task (in plain English):

Given two arrays:

- `preorder` (root, left, right)
- `inorder` (left, root, right)

Both contain all values from a binary tree (no duplicates). Reconstruct and return the original binary tree.

Example Input/Output:

Input:

```
preorder = [3,9,20,15,7]
inorder = [9,3,15,20,7]
```

Output:

```
    3
   / \
  9  20
    / \
   15  7
```

How Should You Think About This?

- Preorder: First element is always the root.
- Inorder: The root splits the array into left and right subtrees.
- Recursively apply this to subarrays.

Pen-and-paper is your friend here! Draw the tree as you go.

Try This Test Case:

```
preorder = [1,2,4,5,3,6]
inorder = [4,2,5,1,3,6]
```

Can you sketch the tree?

Brute-force Idea:

At each step, scan the inorder array to find the root. Slice left/right subarrays and recurse.

- Time: O(N^2) (if you copy arrays every time).
- Space: O(N^2) (if you create lots of subarrays).

Optimal Approach:

- Use indices to avoid array copies.

- Precompute a hashmap from value to index in inorder for fast lookups.
- Use recursion with boundaries.

Step-by-step Logic:

- Preorder's first element is root.
- Find root's index in inorder via hashmap.
- Elements to the left in inorder are in the left subtree; right are in the right subtree.
- Track how many elements are in the left subtree to adjust preorder indices.
- Recursively repeat for 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 buildTree(preorder, inorder):
    # Map from value to its index in inorder for O(1) lookup
    idx_map = {val: i for i, val in enumerate(inorder)}

    def helper(pre_left, pre_right, in_left, in_right):
        if pre_left > pre_right:
            return None

        # Root is always preorder[pre_left]
        root_val = preorder[pre_left]
        root = TreeNode(root_val)

        # Find root in inorder
        in_root_idx = idx_map[root_val]

        # Number of nodes in left subtree
        left_size = in_root_idx - in_left

        # Recursively build left and right
        root.left = helper(pre_left + 1, pre_left + left_size, in_left, in_root_idx - 1)
            root.right = helper(pre_left + left_size + 1, pre_right, in_root_idx + 1,
in_right)

        return root
```

```
    return helper(0, len(preorder) - 1, 0, len(inorder) - 1)
```

Time Complexity: O(N) (each node is visited once)

Space Complexity: O(N) (for hashmap and recursion stack)

Explanation of Each Part:

- `idx_map`: Quickly finds the index of any value in the inorder array.
- `helper`: Recursively constructs the tree using index boundaries, not array slices (very efficient).
- `pre_left`, `pre_right`: Boundaries in preorder.
- `in_left`, `in_right`: Boundaries in inorder.
- Base case: If there are no elements to construct, return None.

Trace (Example):

With `preorder = [3,9,20,15,7]`, `inorder = [9,3,15,20,7]`:

- First call: root is 3 (preorder[0]).
- Inorder: 9 is left subtree, 15,20,7 is right.
- Build left/right recursively using indices.

Try This Dry-Run:

```
preorder = [1,2,4,5,3,6]
inorder = [4,2,5,1,3,6]
```

Draw the tree step by step!

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

Did you know this technique is the backbone for many tree serialization/deserialization problems? Try applying it there next!

Problem 2: Construct Binary Tree from Inorder and Postorder Traversal

Problem link: LeetCode 106

What's Different?

- You now have `inorder` (left, root, right) and `postorder` (left, right, root).
- The root is the *last* element of postorder, not the first.

Example Input/Output:

```
inorder = [9,3,15,20,7]
postorder = [9,15,7,20,3]
```

Output:

```
    3
   / \
  9  20
     / \
    15  7
```

Brute-force:

As before, scan to find the root in inorder, slice arrays.

Time: O(N^2), Space: O(N^2).

Optimal Approach:

- Use a hashmap for quick index lookup in inorder.
- Use recursion with indices.
- The root is postorder[post_right].
- Left subtree: elements to the left in inorder.
- Right subtree: elements to the right in inorder.
- Right subtree comes *before* the root in postorder (important difference!).

Step-by-Step:

- Pick postorder[post_right] as root.
- Find root's index in inorder.
- Compute size of left/right subtrees.
- Recursively build left and right subtrees.

Pseudocode:

```
function buildTree(inorder, postorder):
    idx_map = map from value to index in inorder

    function helper(in_left, in_right, post_left, post_right):
        if in_left > in_right:
            return null

        root_val = postorder[post_right]
        root = TreeNode(root_val)

        in_root_idx = idx_map[root_val]
        left_size = in_root_idx - in_left

         root.left = helper(in_left, in_root_idx - 1, post_left, post_left + left_size -
1)
        root.right = helper(in_root_idx + 1, in_right, post_left + left_size, post_right
- 1)

        return root

    return helper(0, len(inorder) - 1, 0, len(postorder) - 1)
```

Example Dry-Run:

```
inorder = [2,1,3]
postorder = [2,3,1]
```

- Root is 1 (postorder[-1]).
- Left: [2], Right: [3].

Another Test Case:

```
inorder = [4,2,5,1,3,6]
postorder = [4,5,2,6,3,1]
```

Sketch the tree step by step!

Time Complexity: O(N)

Space Complexity: O(N)

How is this similar/different from Problem 1?

- Both use divide-and-conquer and index math.
- Difference: which traversal gives you the root, and how you split postorder/preorder arrays.

Problem 3: Construct Binary Tree from Preorder and Postorder Traversal

Problem link: LeetCode 889

What's Different or More Challenging?

- You have `preorder` (root, left, right) and `postorder` (left, right, root).
- Without inorder, the tree is not always uniquely determined! (Assume full binary tree for this problem.)

Example Input/Output:

```
preorder = [1,2,4,5,3,6,7]
postorder = [4,5,2,6,7,3,1]
```

Output:

```
      1
    / \
   2   3
  / \ / \
 4  5 6 7
```

Brute-force:

Try all splits, but this is inefficient.

Optimal Approach:

- Use preorder to get root.
- Next element in preorder is left child.
- Find the left child's index in postorder to determine the size of the left subtree.
- Recursively build left and right subtrees.

Pseudocode:

```
function constructFromPrePost(preorder, postorder):
```

```
    if not preorder: return null
    root = TreeNode(preorder[0])
    if len(preorder) == 1: return root

    left_root_val = preorder[1]
    idx = index of left_root_val in postorder

    left_size = idx + 1

    root.left = constructFromPrePost(preorder[1:left_size+1], postorder[0:left_size])
    root.right = constructFromPrePost(preorder[left_size+1:], postorder[left_size:-1])

    return root
```

Key Steps:

- Root is preorder[0].
- Left subtree size is found by locating preorder[1] in postorder.
- Split both arrays accordingly.

Example Dry-Run:

```
preorder = [1,2,3]
postorder = [3,2,1]
```

- Root: 1
- Left root: 2, found at index 1 in postorder => left subtree size is 2.

Try This:

```
preorder = [1,2,4,5,3,6]
postorder = [4,5,2,6,3,1]
```

Can you reconstruct the tree?

Time Complexity: O(N^2) (since we search for index each time, can be improved with hashmap)

Space Complexity: O(N) (recursion)

Reflect:

- Notice the pattern: always use the traversal that gives you the root to guide recursion.
- Think about how adding a hashmap to postorder could improve lookup speed.

Summary and Next Steps

Today, you learned how to reconstruct binary trees from:

- Preorder + Inorder
- Inorder + Postorder
- Preorder + Postorder

All three rely on:

- Understanding what each traversal reveals.
- Using recursion and array index math (not slices!).
- Hashmaps for quick index lookup (when possible).

Key Patterns:

- Always start with the traversal that tells you the root.
- Use the other traversal to find subtree boundaries.
- Avoid unnecessary array copying by using indices.

Common Mistakes:

- Accidentally copying arrays instead of using indices.
- Forgetting to use a hashmap for O(1) lookups.
- Off-by-one errors in index math.
- Assuming uniqueness when it may not exist (especially with preorder+postorder).

Action List

- Solve all three problems on your own, even the one with code provided.
- Try implementing problem 3 with a hashmap for postorder to improve speed.
- Challenge: Can you derive iterative (non-recursive) versions?
- Practice drawing the recursion tree on paper for each test case.
- Compare your solutions with others to spot edge cases and style improvements.
- If you get stuck, review the patterns and step through the process with smaller inputs.

Keep practicing—these divide-and-conquer tree problems are classic for a reason. Master them, and you'll be ready for almost any tree question in an interview!