Topic Introduction

Today's focus is on **tree path enumeration**—a classic pattern in binary tree problems. If you've ever traced your finger along every possible route from the root of a tree to its leaves, you've already done the core of this technique!

What is tree path enumeration?

At its heart, this concept is about systematically listing or processing all possible paths from the root of a binary tree to its leaves. You might be collecting these paths, checking some property for each, or aggregating results.

How does it work?

Typically, you use **depth-first search (DFS)**—either recursively or with an explicit stack—to explore every branch. At each step, you track the path you've taken so far. When you reach a leaf node, you process the full path (e.g., print it, sum its values, etc.).

Why is it useful in interviews?

These problems are favorites because they test your understanding of recursion, backtracking, and how to "carry state" through a tree traversal. Interviewers love to see if you can manage path data, handle base cases, and write clean recursive code.

Simple Example (but not one of today's problems):

Suppose you have a tree:

```
   1
  / \
 2   3
```

All root-to-leaf paths are:

- [1, 2]
- [1, 3]

A common interview question might ask you to print all such paths, or sum their values.

Introducing the Trio: Why These Problems?

Let's look at three related LeetCode problems:

- **Binary Tree Paths**: Find all root-to-leaf paths and return them as strings.
- **Sum Root to Leaf Numbers**: Each path forms a number; sum all such numbers.
- **Path Sum II**: Find all root-to-leaf paths where the sum of values equals a target.

All three require enumerating paths from root to leaves, but each asks you to process those paths differently. They are grouped because mastering their shared pattern unlocks a whole category of tree problems.

Problem 1: Binary Tree Paths

PrepLetter: Binary Tree Paths and similar

Problem Statement (Rephrased):

Binary Tree Paths (LeetCode 257)

Given the root of a binary tree, return all root-to-leaf paths as strings in the format "node1->node2->...->nodeN".

Example Input/Output:

Input:

```
    1
   / \
  2   3
   \
    5
```

Output: `["1->2->5", "1->3"]`

Walkthrough:

- One path: 1 -> 2 -> 5, so string is "1->2->5"
- Another path: 1 -> 3, so string is "1->3"

Thought Process:

Imagine tracing every path from the root to the leaves, writing down the node values you pass through. When you reach a leaf, you convert your list to a string and add it to your results.

Try on Paper:

Given this tree:

```
    4
   / \
  9   0
 /
5
```

What are the root-to-leaf paths?

Brute-force Approach:

Recursively visit every path, collecting node values. At each leaf, join the path into a string.

Time Complexity: O(N^2), since you visit each node and may copy the path (up to O(N) length) at each leaf.

Optimal Approach:

We'll use a recursive DFS. At each call:

- Add the node's value to the current path.
- If it's a leaf, convert the path to a string and add to results.
- Otherwise, recurse left and right.
- Backtrack by removing the last value after recursive calls.

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

class Solution:
    def binaryTreePaths(self, root):
        result = []

        def dfs(node, path):
            if not node:
                return
            # Add current node to path
            path.append(str(node.val))
            # If it's a leaf, join path and add to result
            if not node.left and not node.right:
                result.append("->".join(path))
            else:
                # Recurse on children
                dfs(node.left, path)
                dfs(node.right, path)
            # Backtrack
            path.pop()

        dfs(root, [])
        return result
```

Time Complexity:

- O(N^2): For N nodes, each path could be up to N in length, and you build a string at each leaf.

Space Complexity:

- O(N * L), where L is the average path length (for storing all paths); also O(H) for call stack, where H is tree height.

Explanation of Parts:

- `dfs(node, path)`: Walks the tree, maintaining the current path.
- `path.append(...)`: Adds the current value.
- At a leaf, `"->".join(path)` converts path list to string.
- After exploring children, `path.pop()` removes the last value (backtracking).

Trace of Test Case:

Input:

```
   1
  / \
 2   3
  \
   5
```

- Start at 1: path = [1]

- Go left to 2: path = [1,2]
- 2 has right child 5: path = [1,2,5] (leaf) => add "1->2->5"
- Backtrack to 2, then 1
- Go right to 3: path = [1,3] (leaf) => add "1->3"

Another Test Case to Try:

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

What are the root-to-leaf paths? Try working it out!

Self-attempt Encouragement:

Take a moment to write this out, or try coding it yourself before reviewing the solution.

Problem 2: Sum Root to Leaf Numbers

Problem Statement (Rephrased):

Sum Root to Leaf Numbers (LeetCode 129)

Each path from root to leaf forms a number by concatenating the node values. Return the sum of all these numbers.

Example Input/Output:

Input:

```
   1
  / \
 2   3
```

Paths: 1->2 forms 12, 1->3 forms 13.

Output: 25

How is this similar?

Again, we need to enumerate all root-to-leaf paths. But now, instead of collecting their values as strings, we interpret each path as a number and sum them.

Brute-force:

List every path, convert each to a number, then sum.

Time: O(N^2) (collecting all paths).

Optimal Approach:

Instead, as we traverse, carry along the current number formed so far. At each node, multiply the previous number by 10 and add the node's value.

When you reach a leaf, add the complete number to the total sum.

Step-by-step Logic:

- At each recursive call, pass down the number formed so far.

- At a leaf, add that number to your total.
- Recurse to both children.

Pseudocode:

```
function sumNumbers(node, currentNumber):
    if node is null:
        return 0
    currentNumber = currentNumber * 10 + node.val
    if node is a leaf:
        return currentNumber
    return sumNumbers(node.left, currentNumber) + sumNumbers(node.right, currentNumber)
```

Example Trace:

Input:

```
    4
   / \
  9   0
 / \
5   1
```

Paths:

- 4->9->5: 495
- 4->9->1: 491
- 4->0: 40

Sum: 495 + 491 + 40 = 1026

Try This Test Case:

```
    2
   / \
  0   3
       \
        7
```

What numbers do you get?

Time Complexity:

O(N), as each node is visited once.

Space Complexity:

O(H) for recursion stack.

Walkthrough of Example:

- At 4, number = 4
- At 9, number = 49
- At 5, number = 495 (leaf) => add 495
- At 1, number = 491 (leaf) => add 491
- At 0, number = 40 (leaf) => add 40

Total sum = 495 + 491 + 40 = 1026

Try dry-running with your own test case!

Problem 3: Path Sum II

Problem Statement (Rephrased):

Path Sum II (LeetCode 113)

Given a binary tree and a target sum, return all root-to-leaf paths where each path's sum equals the target.

How is this different?

Instead of converting paths to strings or numbers, you check the sum of values along each root-to-leaf path. If it equals the target, add the path to your result list.

Added challenge:

You must keep track of the current path (like in problem 1) and also the running sum (like in problem 2).

Optimal Approach:

DFS with backtracking:

- At each node, add its value to the current path and running sum.
- If it's a leaf and sum equals target, add path to result.
- Else, recurse on children.
- Backtrack after recursion.

Pseudocode:

```
function pathSum(node, targetSum, currentPath, currentSum):
    if node is null:
        return
    add node.val to currentPath
    currentSum += node.val
    if node is a leaf and currentSum == targetSum:
        add a copy of currentPath to result
    else:
        pathSum(node.left, targetSum, currentPath, currentSum)
        pathSum(node.right, targetSum, currentPath, currentSum)
    remove last from currentPath  // backtrack
```

Example Input:

```
    5
   / \
  4   8
 /   / \
11  13  4
/  \    / \
7    2  5  1
```

Target sum: 22

Paths:

- 5->4->11->2 (5+4+11+2=22)
- 5->8->4->5 (5+8+4+5=22)

Output:

```
[[5,4,11,2],[5,8,4,5]]
```

Try This Test Case:

```
   1
  / \
 2   3
```

Target sum: 3

What paths sum to 3?

Time Complexity:

O(N^2): Each path can be up to O(N) in size, and you may copy it at each leaf.

Space Complexity:

O(N^2) for storing paths, O(H) for the call stack.

Try implementing this!

Notice the pattern: DFS with path tracking and backtracking. What would you change if you needed only the count, not the paths themselves?

Summary and Next Steps

Recap:

These three problems are all about enumerating root-to-leaf paths in a binary tree, but each tweaks what you do with each path:

- **Binary Tree Paths**: Collect and format all paths.
- **Sum Root to Leaf Numbers**: Convert paths to numbers and sum.
- **Path Sum II**: Collect only paths that sum to a target.

Key Patterns & Insights:

- **DFS with path tracking** is essential.
- **Backtracking** (removing the last node after recursion) keeps your path data correct.
- Carrying extra state (like running sum or number) avoids recomputing.
- Be careful about when to add to results: only at leaves!

Common Mistakes:

- Forgetting to backtrack (leading to wrong paths).
- Not checking for leaf nodes correctly.
- Mutating shared lists or variables—make copies as needed.

Action List

- Try coding all three problems yourself, including the one with code above, without peeking.
- For Problems 2 and 3, try a different technique (e.g., iterative DFS with a stack).
- Explore related problems: e.g., path sum I (just check if one path exists), or finding the longest root-to-leaf path.
- Compare your solutions with others: How do they handle edge cases? Is their recursion clean?
- Don’t worry if some test cases trip you up. Keep practicing and reflecting!

Happy coding! You’re not just learning tree traversal—you’re building the muscle for tackling all types of recursive and path-based problems. Keep going!