Topic Introduction

Today, we’re diving into a trio of classic tree problems that share a powerful and versatile technique: **tree path sum traversal**. If you’ve worked with binary trees before, you’ve probably traversed them in different ways — pre-order, in-order, post-order — but when you add a *sum constraint* to the paths, things get interesting.

What Are Path Sums in Trees?

A *path sum* in a binary tree is simply the sum of node values along a path from one node to another. Often, interview questions focus on *root-to-leaf* paths, but sometimes, any path (downward, not necessarily from root) is fair game. The goal is typically to check if certain sums can be formed, to find all such paths, or to count how many exist.

Why Is This Useful in Interviews?

- **Recursive thinking:** These problems sharpen your recursion and backtracking skills.
- **Pattern recognition:** They often involve traversing the tree while carrying a piece of information (the path sum so far).
- **Optimization:** Recognizing how to prune paths or optimize searches can separate a good solution from a great one.

Example (Not One of the Main Problems)

Suppose you want to check if there is a root-to-leaf path in this tree that sums to 10:

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

A root-to-leaf path is any path starting at the root and ending at a leaf. Here, the path 5 -> 2 -> 1 sums to 8 (not 10), but 5 -> 2 -> 7 sums to 14. So, the answer would be "no" for sum = 10.

The Three Path Sum Problems

Let’s look at three LeetCode favorites:

- **Path Sum ([LeetCode 112](https://leetcode.com/problems/path-sum/))**: Does *any* root-to-leaf path have a given sum?
- **Path Sum II ([LeetCode 113](https://leetcode.com/problems/path-sum-ii/))**: Return *all* root-to-leaf paths where the sum equals the target.
- **Path Sum III ([LeetCode 437](https://leetcode.com/problems/path-sum-iii/))**: Count the *number* of paths (not necessarily root-to-leaf, but must go downward) that sum to the target.

Why group these? All three are about exploring tree paths to satisfy a sum constraint. The first asks "does one exist?", the second "which ones exist?", and the third "how many exist even if the path starts/ends anywhere?" They build on each other in complexity and technique.

Let’s start with the most basic, build our intuition, and work up!

Problem 1: Path Sum ([LeetCode 112](https://leetcode.com/problems/path-sum/))

Problem Statement (Rephrased)

Given a binary tree and a target sum, determine whether the tree has a *root-to-leaf* path such that adding up all the values along the path equals the target sum.

- *A leaf is a node with no children.*

Example

```
Input:
    5
   / \
  4   8
     / \
    13  4
         \
          1
Target sum = 9

Output: False

Explanation:
Possible root-to-leaf paths:
- 5 -> 4 = 9 (leaf, but 4 has no children; that's a leaf, so this is True! Correction
here.)
Actually, 5->4 is a root-to-leaf path and sums to 9. So, Output: True.
```

Thought Process

- At each node, subtract the node's value from the target sum.
- If you reach a leaf and the remaining sum is zero, you've found a valid path.
- Otherwise, recursively check left and right children.

Tip: Try sketching the tree and tracking the sum as you traverse.

Another Test Case

```
Input:
    1
   / \
  2   3
Target sum = 5
Output: False
```

Brute Force Approach

Check every root-to-leaf path and sum their values. This is already what recursion does. Time complexity: O(N), where N = number of nodes (since you visit every node once).

Optimal Approach

Recursion is the optimal approach here; it’s clean and intuitive.

- **Core Pattern:** Recursively traverse, subtracting node values from the target.
- **Step-by-step:**
  - Base case: If the node is null, return False.
  - If it’s a leaf, check if node value equals remaining sum.
  - Otherwise, recurse on left and right children with updated sum.

Python Solution

```
class Solution:
    def hasPathSum(self, root, targetSum):
        # Helper function to perform recursion
        if not root:
            return False

        # If the node is a leaf
        if not root.left and not root.right:
            return root.val == targetSum

        # Recursively check left and right subtrees
        left_has = self.hasPathSum(root.left, targetSum - root.val)
        right_has = self.hasPathSum(root.right, targetSum - root.val)

        return left_has or right_has
```

Time Complexity: O(N) — where N = number of nodes

Space Complexity: O(H) — H = height of tree (due to recursion stack; H = logN for balanced, N for worst case)

Code Explanation

- The function checks if the current node is null (base case).
- If at a leaf, it checks if the sum matches.
- Otherwise, it recurses left and right, subtracting the current node's value from the sum.
- Returns True if any path matches.

Trace Example

Suppose:

```
Input:
    5
   / \
  4   8
Target sum = 9
```

- At root (5), targetSum = 9.
- After subtracting 5: targetSum = 4.
- Go left to 4 (which is a leaf), subtract 4: targetSum = 0.
- Since leaf and targetSum == 0, return True.

Try This Test Case

```
    1
   / \
  2   3
Target sum = 3
```

- Does a root-to-leaf path sum to 3?

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

Problem 2: Path Sum II ([LeetCode 113](https://leetcode.com/problems/path-sum-ii/))

Problem Statement (Rephrased)

Given a binary tree and a target sum, return **all root-to-leaf paths** where the sum of the node values equals the target sum.

Example

```
Input:
    5
   / \
  4   8
     / \
    13  4
       / \
      5   1
Target sum = 22

Output: [
  [5,4,13],
  [5,8,4,5]
]
```

How is this different?

- Instead of "does one path exist?", now we want **all** root-to-leaf paths that sum to the target.
- We must track the actual path, not just the sum.

Brute Force

- Explore all root-to-leaf paths, record their sums, and collect those matching the target.
- Time complexity: O(N), but since we have to store every valid path, output size can be large.

Optimal Approach

- Use recursion with a path list to track current path.
- When at a leaf, if the sum matches, add a *copy* of path to results.
- Backtrack: remove the current node from path after recursion.

Step-by-step:

- At each node, add value to current path.
- If leaf and sum matches, append path to result.
- Recurse left and right with updated sum.
- Backtrack (pop value from path before returning).

Pseudocode

```
function pathSumII(root, targetSum):
    result = []
    dfs(node, currentPath, remainingSum):
        if node is null:
            return
        add node.val to currentPath
        if node is leaf and node.val == remainingSum:
            add copy of currentPath to result
        else:
            dfs(node.left, currentPath, remainingSum - node.val)
            dfs(node.right, currentPath, remainingSum - node.val)
        remove last element from currentPath  // backtrack
    dfs(root, [], targetSum)
    return result
```

Example Trace

Input:

```
    1
   / \
  2   3
Target sum = 3

Paths:
- 1 -> 2: sum = 3 (valid)
- 1 -> 3: sum = 4 (invalid)

Output: [[1,2]]
```

Try This Test Case

```
    2
   / \
  3   4
 /     \
5       6
Target sum = 10

Possible paths:
- 2->3->5: sum = 10 (valid)
- 2->4->6: sum = 12 (invalid)
```

Complexity

- **Time:** O(N), but can be more if many valid paths (since output size can be large)
- **Space:** O(H) for recursion, plus O(K*L) for storing K paths of average length L

Problem 3: Path Sum III ([LeetCode 437](https://leetcode.com/problems/path-sum-iii/))

Problem Statement (Rephrased)

Given a binary tree and a target sum, count the **number of paths** that sum to the target.

- **Paths can start and end anywhere, but must go downward (parent to child).**

What's Different?

- Paths don't have to start at the root or end at a leaf.
- Can start at any node, as long as the path is downward.

Brute Force

- For every node, try all downward paths starting at that node, summing up values.
- For each node, O(N) work, total O(N^2).

Optimal Approach

- Use a hash map to store running prefix sums.
- As you traverse, keep track of the sum from root to current node.
- For each node, check how many times (current_sum - target) has occurred before.
- This uses the *prefix sum* pattern within a tree.

Pseudocode

```
function pathSumIII(root, targetSum):
    prefixSums = {0: 1}
    count = 0

    dfs(node, currSum):
        if node is null:
            return
        currSum += node.val
        count += prefixSums.get(currSum - targetSum, 0)
        prefixSums[currSum] += 1
        dfs(node.left, currSum)
        dfs(node.right, currSum)
        prefixSums[currSum] -= 1  // backtrack

    dfs(root, 0)
    return count
```

Example

Input:

```
    10
   /  \
  5   -3
 / \    \
3   2    11
Target sum = 8

Valid paths:
- 5->3
- 5->2->1
- -3->11
Total: 3 paths
```

Try This Test Case

```
    1
   / \
  2   3
Target sum = 3

Possible paths:
- 1->2: sum = 3
- 3: sum = 3
Total: 2 paths
```

Nudge:

How does prefix sum improve from brute force? Can you spot the recursion pattern and how the hash map prevents recomputation?

Complexity

- **Time:** O(N)
- **Space:** O(H) for recursion + O(N) for hash map

Summary and Next Steps

Today, you tackled three related tree path sum problems:

- **Path Sum:** Simple existence check, root-to-leaf, using recursion.
- **Path Sum II:** Collect all such paths, requiring backtracking.
- **Path Sum III:** Count all paths (not just root-to-leaf), using prefix sums and hash maps.

Key Patterns:

- Recursive traversal with sum tracking.
- Backtracking to manage paths.
- Prefix sum with hash maps for counting arbitrary downward paths.

Traps to Avoid:

- Forgetting to check only root-to-leaf in the first two.
- Not backtracking the path or hash map state.
- Mixing up node vs. path constraints.

Action List

- Solve all 3 problems on your own — even the one with code provided here!
- Try Problem 2 and 3 using a different approach (e.g., iterative, or with explicit stack).
- Explore related problems like "Sum Root to Leaf Numbers" or "Binary Tree Paths".
- Compare your solutions with others — pay attention to edge cases and recursion/backtracking style.
- If you get stuck, review the recursion trees and dry-run with pen and paper.

Keep practicing, and you'll find these patterns start to click! Happy coding!