Topic Introduction

Welcome back, PrepLetter reader! Today, we're diving into a family of problems that often appear in coding interviews, each with a twist: the **Two Sum** pattern. While you may already know the classic "find two numbers that add up to a target," what if the numbers are in a sorted array? Or a Binary Search Tree? Or streaming in, one at a time? These variations test how well you adapt a core idea to new data structures.

The Two Sum Pattern:
Given a collection of numbers, can you quickly determine if any two of them sum to a specific value?
This is a classic use-case for hashing, sorting, two-pointers, and data structure design.

How does it work?
At its heart, Two Sum is about efficiently checking for pairs. The techniques you use depend on the data structure:

- **Hashing**: Store what you've seen so far to check for complements in O(1) time.
- **Two Pointers**: Walk from both ends of a sorted array, closing in on the answer.
- **Tree Traversal**: Explore all nodes in a BST, leveraging its sorted nature.

Why is it useful in interviews?
Interviewers love this pattern because it tests for:

- Understanding of hash maps and sets
- Efficient traversal of arrays and trees
- Ability to adapt algorithms to new constraints

Mastering these opens the door to a huge class of problems!

Quick Example (not from the target problems):
Suppose you have the array `[1, 4, 6, 8]` and target `10`.
Hashing approach:

- Check if (target - current) is in the set.
- For `1`, check `9` (not in set), add `1`.
- For `4`, check `6` (not in set), add `4`.
- For `6`, check `4` (yes! found: `4` + `6` = `10`).

Why Group These Three Problems?

All three are variations on the Two Sum theme, each using a different data structure:

- **Two Sum IV - Input is a BST**: Find if any two nodes in a Binary Search Tree sum to a target. Requires tree traversal and knowledge of BST properties.
- **Two Sum III - Data structure design**: Design a structure that supports adding numbers and checking for any pair that sums to a target. Tests your ability to blend data structures and algorithms.
- **Two Sum II - Input array is sorted**: The array is sorted, so you can use the efficient two-pointer approach.

We'll start with the BST, since it blends traversal and Two Sum logic, then move to the data structure design, and finally the sorted array case.

Problem 1: Two Sum IV – Input is a BST

[LeetCode Link](https://leetcode.com/problems/two-sum-iv-input-is-a-bst/)

Problem Statement (Rephrased)

Given the root of a Binary Search Tree (BST) and an integer target, determine if there exist two nodes in the tree whose values add up to the target.

Example

Input:
BST:

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

Target: `9`

Output:
`True` (Because 2 + 7 = 9)

Walkthrough

- Start at the root.
- For each node, ask: "Is there another node in the BST whose value is (target - current node's value)?"
- We need to avoid using the same node twice!

Additional Test Case

BST:

```
    1
     \
      3
     /
    2
```

Target: `4`
Expected Output: `True` (1 + 3 = 4)

Brute-Force Approach

- Traverse every pair of nodes and check their sums.
- Time: O(N^2), where N is the number of nodes.

Optimal Approach

Core Pattern:

Use a HashSet to store values we've seen. As we traverse the tree, for each node, check if (target - node.val) is in the set. If yes, we found a pair.

Step-by-Step:

- Initialize an empty HashSet.
- Traverse the tree (DFS or BFS).
- At each node:
  - If (target - node.val) is in the set, return True.
  - Otherwise, add node.val to the set.
- If traversal ends without finding a pair, return False.

Take a moment to try this on pen and paper! Can you walk through the example above?

Another Test Case

BST:

```
    2
   / \
  1   3
```

Target: `6`

Expected Output: `False`

Python Solution

```
class Solution:
    def findTarget(self, root, k):
        # Helper function for DFS traversal
        def dfs(node, seen):
            if not node:
                return False
            # Check if complement is in seen
            if k - node.val in seen:
                return True
            seen.add(node.val)
            # Continue to left and right children
            return dfs(node.left, seen) or dfs(node.right, seen)

        # Set to keep track of seen values
        return dfs(root, set())
```

Time Complexity:

- O(N), where N is the number of nodes (each node visited once).

Space Complexity:

- O(N), for the HashSet storing visited values.

Code Explanation

- The `dfs` function traverses the tree.
- For each node, we check if the required complement is already in `seen`.
- If found, we return `True` immediately.
- Otherwise, add current value to `seen` and continue traversing.

Trace Example

BST:

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

Target: `9`

- Start at `5`. `9-5=4`, not in set. Add `5`.
- Visit `3`. `9-3=6`, not in set. Add `3`.
- Visit `2`. `9-2=7`, not in set. Add `2`.
- Visit `4`. `9-4=5`, `5` is in set! Return `True`.

Test Case for You

BST:

```
    1
     \
      2
       \
        3
```

Target: `5`

What should the output be?

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

Problem 2: Two Sum III – Data structure design

[LeetCode Link](https://leetcode.com/problems/two-sum-iii-data-structure-design/)

Problem Statement (Rephrased)

Design a class with two methods:

- `add(number)`: Add a number to an internal data structure.
- `find(value)`: Return true if any two numbers in the data structure sum to `value`.

Example

```
ds = TwoSum()
ds.add(1)
ds.add(3)
ds.add(5)
print(ds.find(4)) # True (1+3)
print(ds.find(7)) # False
```

Additional Test Case

Add: 0, 0

Find(0) -> True

Brute-Force Approach

- For `find(value)`, check every pair in the list.
- Time: O(N) for `add`, O(N^2) for `find`.

Optimal Approach

Similarities:

Still checking for pairs that sum to a target, but now we must support efficient repeated queries.

Optimal Pattern:

- Use a HashMap to count occurrences of each number.

- For `add(number)`: Increment count in HashMap.
- For `find(value)`:
    - For each unique number, compute complement = value - number.
    - If complement == number, ensure count >= 2.
    - If complement != number, check if complement exists.

Step-by-Step (Pseudocode):

```
class TwoSum:
    Initialize: num_counts = empty hashmap

    add(number):
        increment num_counts[number] by 1

    find(value):
        for each num in num_counts:
            complement = value - num
            if complement == num:
                if num_counts[num] >= 2:
                    return True
            else:
                if complement in num_counts:
                    return True
        return False
```

Example Trace

Add: 1, 3, 5

Find(4):

- num: 1, complement: 3 (exists) => True

Find(7):

- num: 1, complement: 6 (not in map)
- num: 3, complement: 4 (not in map)
- num: 5, complement: 2 (not in map)

Return False

Another Test Case

Add: 2, 2, 3

Find(4): Should return True (2+2)

Time and Space Complexity

- `add(number)`: O(1)
- `find(value)`: O(N), where N is the number of unique elements
- Space: O(N), for storing numbers

Try implementing this yourself, and dry-run with the above test case!

Problem 3: Two Sum II – Input array is sorted

[LeetCode Link](https://leetcode.com/problems/two-sum-ii-input-array-is-sorted/)

Problem Statement (Rephrased)

Given a sorted array of integers and a target, return the 1-based indices of the two numbers that add up to the target.

Example

Input: `numbers = [2, 7, 11, 15]`, target = 9
Output: `[1, 2]` (2 + 7 = 9)

What's New or Tricky?

- The input is sorted!
- Must return indices (1-based).

Brute-Force

- For each pair, check sum.
- Time: O(N^2)

Optimal Approach

Core Pattern:
Use the **two-pointer** technique.

Pseudocode:

```
left = 0
right = len(numbers) - 1
while left < right:
    current_sum = numbers[left] + numbers[right]
    if current_sum == target:
        return [left + 1, right + 1]
    elif current_sum < target:
        left += 1
    else:
        right -= 1
return []
```

Example Trace

numbers = [2, 7, 11, 15], target = 9

- left=0 (2), right=3 (15): 2+15=17 > 9, move right
- left=0, right=2 (11): 2+11=13 > 9, move right
- left=0, right=1 (7): 2+7=9, return [1,2]

Another Test Case

numbers = [1, 2, 3, 4], target = 8
What should be returned?

Time and Space Complexity

- Time: O(N)
- Space: O(1)

Try implementing this approach and test with the above case!

Tip: Reflect on how the sorted property made the two-pointer strategy possible here. What would you do if the array wasn't sorted?

Summary and Next Steps

Recap:
Today, you saw how the core Two Sum pattern adapts to different data structures:

- **BST**: Use a set during DFS/BFS to check for complements.
- **Data structure design**: Use a hashmap to count elements for efficient add/find.
- **Sorted array**: Use the two-pointer method for O(N) time and O(1) space.

Key Insights:
- Hashing is versatile for "seen before?" checks.
- Two-pointer only works when data is sorted or can be traversed in order.
- Data structure design problems often test your ability to balance operation costs.

Common Pitfalls:
- Forgetting to avoid using the same element twice.

- Not leveraging sorted/BST properties for efficiency.
- Confusing 0-based and 1-based indexing.

Action List

- Solve all 3 problems on your own, even the BST one with code provided.
- Try the Two Sum II problem as if the array is **not** sorted: can you adapt your approach?
- Implement Two Sum III both with a HashMap and with a brute-force approach; compare efficiencies.
- For the BST problem, try solving it with in-order traversal and two pointers (advanced!).
- Share your code or thoughts with a friend or mentor and discuss any tricky cases.
- Remember: Practice, review, and compare. Getting stuck is part of the learning process!

You're getting stronger with every variation you tackle. Keep practicing, and Two Sum (and its many cousins) will become second nature!