Mastering Two Pointers: The Secret Sauce Behind Pair, Triplet, and Quad Sum Problems

Welcome to your very first PrepLetter! Today, we're diving into a technique that's a true interview favorite: **the Two Pointers pattern**. If you're prepping for technical interviews, you'll encounter this strategy everywhere from arrays to strings, from pairs to quadruplets.

What is the Two Pointers Technique?

The Two Pointers technique involves using two indices (pointers) that move through the data structure (usually an array) in a coordinated way. Typically, you'll set one pointer at the start and another at the end, moving them towards each other depending on the problem's requirements.

Why Use Two Pointers?

- **Efficiency**: It can reduce your algorithm from O(n^2) to O(n) or O(n^2) from O(n^3).
- **Ideal For**: Sorted arrays and problems involving sums, subarrays, or searching for relationships between elements.

Simple Example

Suppose you have a sorted array: `[1, 2, 4, 7, 11, 15]` and you want to find if any two numbers sum to 15.

- Start with left pointer `l = 0` and right pointer `r = 5` (the ends).
- Check `arr[l] + arr[r]`. If it's less than 15, move `l` right. If it's more, move `r` left. If equal, you found the pair!

This approach is the backbone for a family of popular LeetCode problems: **Two Sum**, **3Sum**, and **4Sum**. Each increases in complexity, but the heart of the solution remains the same: fix some numbers and use two pointers to find the rest.

Let's level up, step by step.

Problem 1: Two Sum II – Input Array Is Sorted

Two Sum II – LeetCode #167

Problem Statement (In Plain English)

Given a sorted array of integers and a target value, find the indices (1-based) of two numbers that add up to the target. Assume exactly one solution exists, and you cannot use the same element twice.

Example

Input: `numbers = [2, 7, 11, 15]`, `target = 9`

Output: `[1, 2]`

Explanation: 2 + 7 = 9, and those are at indices 1 and 2 (1-based).

Another Example (Try This Yourself!)

Input: `numbers = [1, 3, 4, 6, 8, 10]`, `target = 14`

What should the output be? Grab a pen and paper before reading on!

Pause and Solve

Take a moment to try solving this on your own before reading the solution.

Solution: Two Pointers in Action

Since the array is sorted, we can use the classic two pointers approach:

- Start with `left` at the beginning and `right` at the end.
- If `numbers[left] + numbers[right]` is less than target, move `left` right.
- If it's more, move `right` left.
- If equal, return the indices (remember to add 1 since it's 1-based).

Python Code

```
def twoSum(numbers, target):
    left, right = 0, len(numbers) - 1

    while left < right:
        curr_sum = numbers[left] + numbers[right]

        if curr_sum == target:
            return [left + 1, right + 1]  # 1-based indexing!
        elif curr_sum < target:
            left += 1
        else:
            right -= 1
    return []
```

Time and Space Complexity

- **Time**: O(n) – Each pointer moves at most n steps.

- **Space**: O(1) – No extra space used.

How It Works (Step-by-Step)

For `numbers = [1, 3, 4, 6, 8, 10]`, `target = 14`:

- `left = 0 (1)`, `right = 5 (10)`: 1+10 = 11 < 14, move left.
- `left = 1 (3)`, `right = 5 (10)`: 3+10 = 13 < 14, move left.
- `left = 2 (4)`, `right = 5 (10)`: 4+10 = 14, found! Return [3, 6].

Challenge

Did you know this could also be solved with a hash map (dictionary)? Once you finish this article, try that method for practice!

Problem 2: 3Sum

3Sum – LeetCode #15

Problem Statement (In Plain English)

Given an array of integers, find all unique triplets (three numbers) in the array which sum to zero. The solution set must not contain duplicate triplets.

Example

Input: `nums = [-1, 0, 1, 2, -1, -4]`
Output: `[[-1, -1, 2], [-1, 0, 1]]`

Triplets must not repeat – order doesn't matter.

Another Example (Try This!)

Input: `nums = [-2, 0, 1, 1, 2]`
What unique triplets sum to zero? Give it a try on paper!

Pause and Solve

Take a moment to try solving this on your own before reading the solution.

Solution: Extending Two Pointers to Triplets

How does this differ from Two Sum?

• Now, for each number, you fix it, and search the rest of the array for two numbers that sum to `-nums[i]` (the negative of your fixed number).

Process:

• Sort the array (important for two pointers and duplicate handling).

• Loop through each index `i` (except the last two).

• For each `i`, set left and right pointers at `i+1` and end.

• Move pointers as in Two Sum, looking for pairs that sum to `-nums[i]`.

• Skip duplicates to ensure unique triplets.

Python Code

```
def threeSum(nums):
    nums.sort()
    res = []
    n = len(nums)

    for i in range(n):
        if i > 0 and nums[i] == nums[i - 1]:
            continue  # Skip duplicate fixed element

        left, right = i + 1, n - 1
        target = -nums[i]

        while left < right:
            curr_sum = nums[left] + nums[right]
            if curr_sum == target:
                res.append([nums[i], nums[left], nums[right]])

                # Skip duplicate second and third numbers
                while left < right and nums[left] == nums[left + 1]:
                    left += 1
                while left < right and nums[right] == nums[right - 1]:
                    right -= 1

                left += 1
                right -= 1
            elif curr_sum < target:
                left += 1
            else:
                right -= 1
    return res
```

Complex Step Explanation

- The duplicate skipping ensures no repeated triplets.
- Sorting enables us to skip and move pointers efficiently.

Time and Space Complexity

- **Time**: O(n^2) – Outer loop O(n), inner while loop O(n) in total due to pointer movement.
- **Space**: O(1) if we ignore output, otherwise O(k) for k triplets.

How It Works (Example Walkthrough)

For `nums = [-2, 0, 1, 1, 2]`:

- After sorting: `[-2, 0, 1, 1, 2]`
- For i=0 (`-2`), look for pairs summing to 2 (since -(-2) = 2):
    - left=1 (`0`), right=4 (`2`): 0+2=2, valid. Add `[-2, 0, 2]`.
    - Move pointers, skip duplicates.
- For i=1 (`0`), look for pairs summing to 0:
    - left=2 (`1`), right=4 (`2`): 1+2=3 > 0, move right.
    - left=2, right=3: 1+1=2 > 0, move right.
- For i=2 and so on, continue similarly.

Try this process with your example and see how duplicates are avoided.

Compare With Previous Problem

Notice how we just “fix” one element and use the same two pointers trick as in Two Sum! The key addition is duplicate avoidance.

Challenge

As now you guessed probably, this can also be solved using hash sets for O(n^2) time. Once you’re done, try implementing that variant!

Problem 3: 4Sum

4Sum – LeetCode #18

Problem Statement (In Plain English)

Given an array of integers and a target value, find all unique quadruplets (four numbers) in the array that add up to the target. Each

quadruplet must not repeat in the result set.

Example

Input: `nums = [1, 0, -1, 0, -2, 2]`, `target = 0`
Output: `[[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]`

Another Example (Try This!)

Input: `nums = [2, 2, 2, 2, 2]`, `target = 8`
What should the output be? Take a moment to try!

Pause and Solve

Take a moment to try solving this on your own before reading the solution.

Solution: The Pattern Grows

Building on the 3Sum pattern, we now **fix two numbers** and use two pointers to find the remaining pair that, with the fixed numbers, sums to the target.

Process:

- Sort the array.
- Loop through each index `i` (first fixed number).
- For each `i`, loop through `j = i+1` (second fixed number).
- For each `i, j`, set `left = j+1`, `right = end`.
- Use two pointers to find pairs that make the total sum equal to the target.
- Skip duplicates for all four numbers.

Python Code

```
def fourSum(nums, target):
    nums.sort()
    res = []
    n = len(nums)

    for i in range(n):
        if i > 0 and nums[i] == nums[i - 1]:
            continue  # Skip duplicate first number
        for j in range(i + 1, n):
            if j > i + 1 and nums[j] == nums[j - 1]:
                continue  # Skip duplicate second number
```

```
            left, right = j + 1, n - 1
            while left < right:
                curr_sum = nums[i] + nums[j] + nums[left] + nums[right]
                if curr_sum == target:
                    res.append([nums[i], nums[j], nums[left], nums[right]])
                    # Skip duplicates for left and right
                    while left < right and nums[left] == nums[left + 1]:
                        left += 1
                    while left < right and nums[right] == nums[right - 1]:
                        right -= 1
                    left += 1
                    right -= 1
                elif curr_sum < target:
                    left += 1
                else:
                    right -= 1
    return res
```

Time and Space Complexity

- **Time**: O(n^3) due to three nested loops (i, j, and the two pointers).
- **Space**: O(1), ignoring output.

How It Works

Let's walk through the example `nums = [2, 2, 2, 2, 2]`, `target = 8`:

- After sorting: `[2, 2, 2, 2, 2]`
- For i=0, j=1, left=2, right=4, sum is 2+2+2+2=8.
- Add `[2, 2, 2, 2]` to result.
- All other combinations are duplicates, and our duplicate checks skip them.

Compare With Previous Problem

Notice the pattern: we fix more numbers as the sum group size increases, but always end with two pointers to find the final pair. Duplicates become more important to handle, and the nesting increases.

Challenge

Can you imagine how 5Sum would work? Or, as an exercise, try implementing a recursive kSum based on this pattern!

And, did you know: Hashing can also be used to solve 4Sum (and kSum), though it's trickier. Give it a try after you're comfortable

with the two-pointer approach!

Summary and Next Steps

Let’s recap what we’ve learned:

- **Two Pointers** is a versatile and efficient technique for sum and searching problems, especially on sorted arrays.
- **Two Sum, 3Sum, and 4Sum** share a core pattern: fix some elements, then use two pointers to find the rest.
- As the group size grows, so does the nesting of loops and the importance of handling duplicates.
- Sorting is crucial for two pointers and for managing duplicates in these problems.

Key Patterns and Variations

- Always sort the input array for easier duplicate handling and pointer movement.
- For kSum problems, fix (k-2) numbers, then use Two Pointers for the remaining pair.
- Don’t forget to skip duplicates at every step to avoid repeated results.

Common Pitfalls

- Forgetting to handle duplicates- leads to repeated output.
- Not converting indices properly (especially in Two Sum II with 1-based indexing).
- Not sorting the array before using two pointers.

Your Action List

- **Solve all three problems yourself**- implement, debug, and test with multiple cases.
- **Explore other kSum variations**- can you generalize to 5Sum or kSum with recursion?
- **Check out other people’s solutions** — see how others handle edge cases and optimize further.
- **Try alternate techniques**- like hash maps for Two Sum, or hash sets for 3Sum.
- **Reflect on mistakes**- it’s totally normal to get stuck or make errors. The key is to keep learning and practicing!

Remember, mastering these problems is not just about memorizing solutions — it’s about recognizing patterns and building intuition. Every time you solve or even struggle with a problem, you’re leveling up.

Stay curious, keep coding, and you’ll be ready to tackle any interview array challenge that comes your way!