Topic Introduction

Welcome to your daily coding interview prep! Today, we're diving again into a classic and versatile algorithmic technique: **the two pointers method**. This pattern is a staple in array problems and is beloved by both interviewers and programmers for its elegance and efficiency.

What is the Two Pointers Technique?
We know you already know this, still touching the surface of it, Won't hurt.
The two pointers technique involves using two separate indices that move through the array (often at different speeds or from different directions) to solve problems around searching, sorting, or counting. It shines when working with **sorted arrays** and is perfect for scenarios where you want to efficiently find pairs or triplets that meet certain criteria.

How does it work?

Imagine you have an array of numbers, and you want to find two numbers that sum up to a target. By sorting the array and placing one pointer at the start and another at the end, you can adjust the pointers based on whether the sum is too small or too large. No need to check every possible pair!

When should you use it in interviews?

- When the problem asks for a pair/triplet/quadruplet with a certain sum or property.
- When the array is sorted or can be sorted.
- When brute force is too slow (O(N^2) or worse).

Simple Example:
Suppose you are given `[1, 2, 4, 6, 8, 9]` and asked: "Does any pair sum to 10?"

- Start with pointers at `1` (left) and `9` (right).
- `1 + 9 = 10` — found in one step!

Now, let's see this in action with a family of related problems: **3Sum** (Yes! again!), **3Sum Closest**, and **3Sum Smaller**. All three build upon the two pointers pattern, extending it from pairs to triplets.

Problem 1: 3Sum Smaller

Link: 3Sum Smaller (LeetCode 259)

Problem Statement (in simple words):
Given an array of integers and an integer `target`, count the number of triplets `i, j, k` (with `i < j < k`) such that `nums[i] + nums[j] + nums[k] < target`.

Example:
Input: `nums = [-2, 0, 1, 3]`, `target = 2`
Output: `2`
Why? The two triplets that sum to less than 2 are:

- `(-2, 0, 1)` sum = -1

- `(-2, 0, 3)` sum = 1

What's different here?

Unlike the usual "find the exact sum" problem, here we want **all triplets whose sum is less than target**. So we need to count, not just find.

Try this example with pen and paper:

`nums = [0, 2, 3], target = 6` — How many triplets sum to less than 6?

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

Solution: Two Pointers after Sorting

Since we need all **ordered triplets** whose sum is less than target, brute force would take O(N^3) time. But here's where two pointers shine!

Steps:

- **Sort the array** (so we can reason about sums).
- For each index `i` (`0` to `n-3`):
  - Set two pointers: `left` at `i+1`, `right` at `n-1`.
  - While `left < right`:
    - Compute `total = nums[i] + nums[left] + nums[right]`
    - If `total < target`, then ALL triplets between `left` and `right` with the current `i` and `left` will work (because increasing `right` will only increase the sum).
      - So, add `(right - left)` to the count.
      - Move `left` rightward.
    - Else, move `right` leftward (to try smaller sums).

Python Code:

```
def threeSumSmaller(nums, target):
    nums.sort()
    count = 0
    n = len(nums)
    for i in range(n - 2):
        left, right = i + 1, n - 1
        while left < right:
            total = nums[i] + nums[left] + nums[right]
            if total < target:
                # All combinations (i, left, any in [left+1, right]) work
                count += (right - left)
                left += 1
            else:
```

```
                right -= 1
    return count
```

Time & Space Complexity:

- **Time:** O(N^2) — outer loop and inner while loop together.
- **Space:** O(1) extra (ignoring sorting).

Code Explanation:

- We sort first to use the two pointers logic.
- For each first element, we look for all valid pairs after it.
- If the sum is too big, move `right` left to try a smaller number.
- If the sum is small enough, all combinations between `left` and `right` are valid, so we add those in bulk.

Need practice? Try running this on:
`nums = [1, 1, -2, 0], target = 2` — How many triplets work?

Did you know? This problem can also be solved with a brute-force three nested loops approach, but it's much less efficient. Give that a shot for comparison after reading!

Let's step up the challenge. Next, let's see how a small tweak leads to a new problem.

Problem 2: 3Sum Closest

Link: 3Sum Closest (LeetCode 16)

Problem Statement:
Given an array of integers `nums` and an integer `target`, find the sum of three integers in `nums` such that the sum is **closest** to `target`. Return this sum.

Example:
Input: `nums = [-1, 2, 1, -4]`, `target = 1`
Output: `2`
Why? The closest sum is `2` (`-1 + 2 + 1 = 2`).

How is this different from 3Sum Smaller?
Instead of counting all triplets with sum less than a target, now you must find **the triplet with the sum closest to the target** (either above or below).

Try this example yourself:
`nums = [1, 1, 1, 0], target = 100` — What should the function return?

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

Solution: Two Pointers after Sorting

Just like before, we sort and use two pointers — but now we keep track of the **closest sum** found so far.

Steps:

- **Sort the array.**
- For each index `i` (`0` to `n-3`):
  - Set pointers: `left` at `i+1`, `right` at `n-1`.
  - While `left < right`:
    - Compute `total = nums[i] + nums[left] + nums[right]`
    - If `total` is closer to `target` than our previous best, update our closest.
    - If `total < target`, move `left` rightward (to increase sum).
    - If `total > target`, move `right` leftward (to decrease sum).
    - If `total == target`, return early (can't get closer!).

Python Code:

```
def threeSumClosest(nums, target):
    nums.sort()
    n = len(nums)
    closest = float('inf') # Start with infinity as the "worst" closest sum
    for i in range(n - 2):
        left, right = i + 1, n - 1
        while left < right:
            total = nums[i] + nums[left] + nums[right]
            # If this sum is closer to target than previous closest, update it
            if abs(total - target) < abs(closest - target):
                closest = total
            if total < target:
                left += 1
            elif total > target:
                right -= 1
            else:
                # If exactly equal, this is the best possible
                return total
    return closest
```

Time & Space Complexity:

- **Time:** O(N^2)
- **Space:** O(1) extra (ignoring sorting).

Code Walkthrough:

- The outer loop picks the "first number" of the triplet.
- The inner loop explores possible pairs with two pointers.
- We keep updating `closest` as we go.
- If we find an exact match, we return immediately.

Test it yourself:
Try: `nums = [0, 2, 1, -3], target = 1` — What should the function return?

Notice the similarity?
We use the same loop structure as in 3Sum Smaller, but the update logic is different (we track the closest sum instead of counting).

For experts:
Did you know you could generalize this idea to kSum Closest using recursion and two pointers? Try to implement that after this article!

Now, let's look at the classic that started it all... We know you already learned this, but the other two problems was crying that they don't wanna take the photo without their friend!

Problem 3: 3Sum

Link: 3Sum (LeetCode 15)

Problem Statement:
Given an array `nums`, return all **unique triplets** `[nums[i], nums[j], nums[k]]` such that `i < j < k` and `nums[i] + nums[j] + nums[k] == 0`.

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

What's different now?
This time, we want **all unique triplets** that sum exactly to zero (not closest, not less than). No duplicates allowed in the result.

Give this example a try:
`nums = [0, 0, 0, 0]` — What should the function return?

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

Solution: Two Pointers after Sorting + Skipping Duplicates

We use the same two pointer structure, but with extra care to **avoid duplicate triplets**.

Steps:

- **Sort the array.**
- For each index `i` (`0` to `n-3`):
  - If `i > 0` and `nums[i] == nums[i-1]`, skip (to avoid duplicate triplets).
  - Set pointers: `left = i+1`, `right = n-1`.
  - While `left < right`:
    - Compute `total = nums[i] + nums[left] + nums[right]`
    - If `total == 0`, add `[nums[i], nums[left], nums[right]]` to result.
      - Move `left` right and skip duplicates.
      - Move `right` left and skip duplicates.
    - If `total < 0`, move `left` right.
    - If `total > 0`, move `right` left.

Python Code:

```
def threeSum(nums):
    nums.sort()
    n = len(nums)
    res = []
    for i in range(n - 2):
        # Skip duplicate values for i
        if i > 0 and nums[i] == nums[i - 1]:
            continue
        left, right = i + 1, n - 1
        while left < right:
            total = nums[i] + nums[left] + nums[right]
            if total == 0:
                res.append([nums[i], nums[left], nums[right]])
                # Move left and right to next distinct values
                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 total < 0:
                left += 1
            else:
                right -= 1
    return res
```

Time & Space Complexity:

- **Time:** O(N^2) — same as before, but with extra skip steps.

- **Space:** O(1) extra (ignoring sorting and output).

Code Explanation:

- The key difference here: we **skip over duplicate values** to ensure unique triplets.
- After finding a valid triplet, we advance both pointers past duplicates before continuing.
- The logic for moving pointers is identical to the previous problems.

Step-by-step on `[0, 0, 0, 0]`:

- After sorting: `[0, 0, 0, 0]`
- i = 0, left = 1, right = 3.
- 0 + 0 + 0 = 0. Add `[0, 0, 0]` to result.
- Skip left and right over duplicates. End.

Try this yourself:

`nums = [-2, 0, 1, 1, 2]` — What are all the unique triplets?

Did you notice?

While a hash set or dictionary can sometimes be used to track duplicates, this two pointers + sort method is usually better for interviews. Try to implement a hash set solution yourself!

Summary and Next Steps

Congratulations! You've just tackled three closely related problems using the versatile **two pointers** pattern:

- **3Sum Smaller** — count all triplets less than a target.
- **3Sum Closest** — find the triplet closest to a target.
- **3Sum** — find all unique triplets that sum to a target.

They all use the **sort + two pointers structure**, but differ in:

- What we do when we find a valid sum (count, compare, or collect).
- How we handle duplicates (especially for 3Sum).
- Whether we seek all, closest, or exact matches.

Key patterns/variations to watch for:

- Sorting is usually the first step for two pointer problems.
- Adjust which pointer moves based on how the sum compares to the target.
- Watch out for duplicate handling!
- These ideas generalize to kSum, subarray problems, and more.

Common pitfalls:

- Forgetting to skip duplicates in 3Sum.
- Not updating pointers correctly (risk of infinite loop).
- Mishandling the count logic in 3Sum Smaller (remember, all pairs between left and right work if the sum is small enough).

Action List:

- **Solve all three problems yourself** — code them from scratch without looking!
- **Try more two pointers problems** — e.g., 2Sum, 4Sum, Two Sum II, Container With Most Water.
- **Review other people's solutions** — see how others handle tricky duplicates or edge cases.
- **Practice explaining your approach** — being able to clearly explain your reasoning is just as important as solving.
- **If you get stuck on complex problems, that's okay!** The more you practice, the more natural these patterns will feel.
- **Experiment:** Try implementing one of these problems using a different approach (like hash sets or recursion) to deepen your understanding.

Keep practicing and remember: patterns like two pointers are your friends. With each problem you solve, you’re building strong problem-solving muscles. Happy coding!