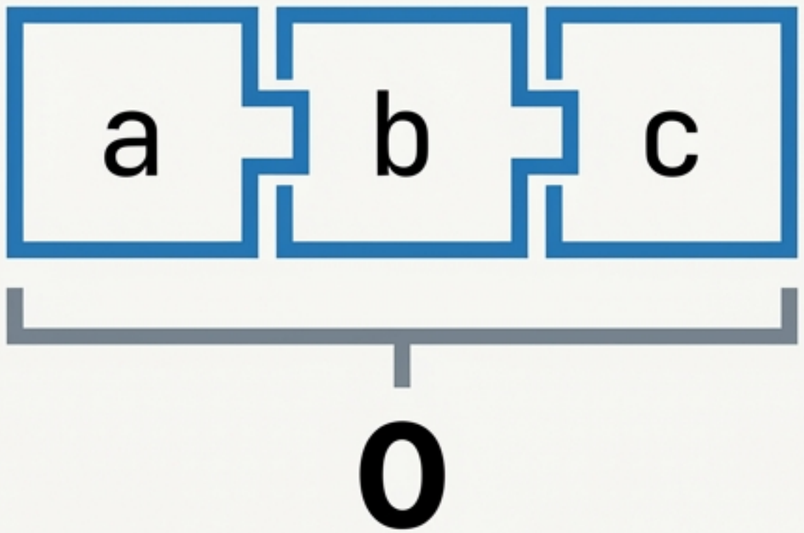
NeetCode Pattern: Two Pointers

Mastering 3 Sum

The Two-Pointer Approach

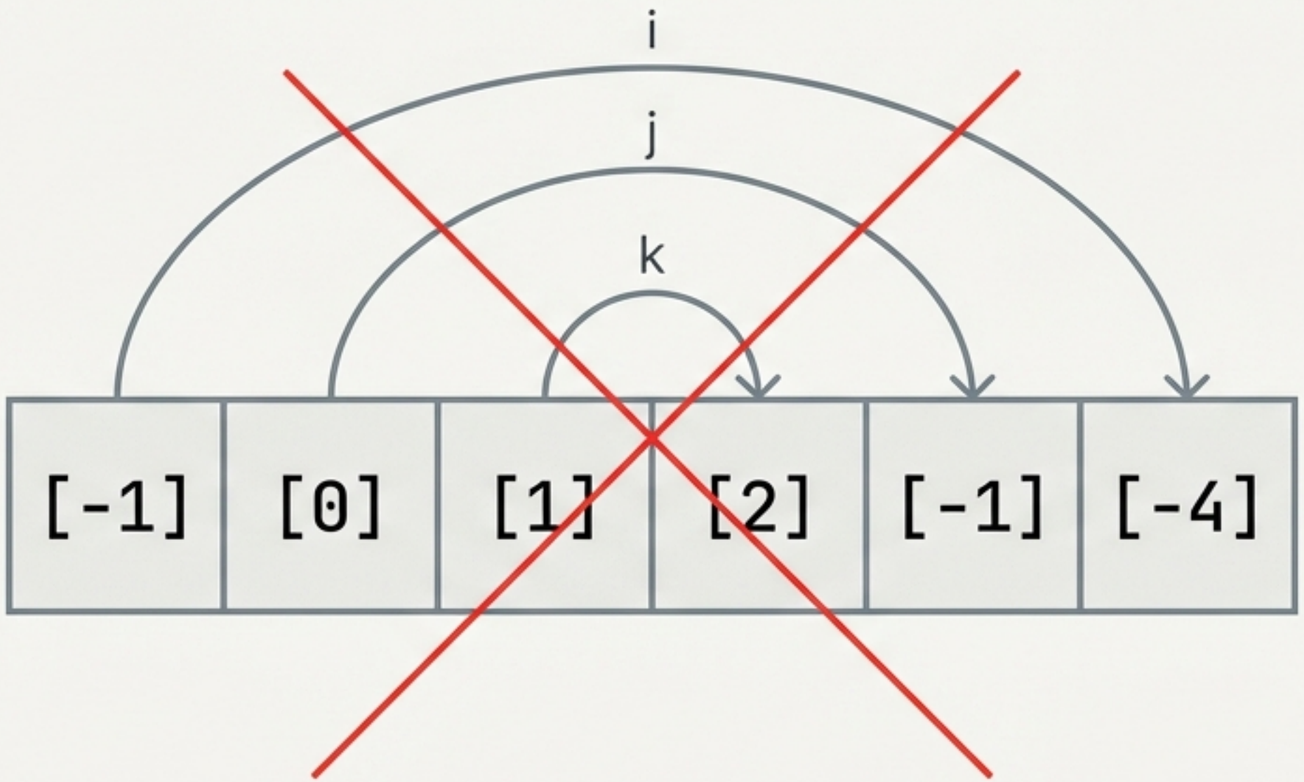


Finding unique triplets efficiently without the **O(N³) brute force trap.**

The Brute Force Bottleneck

THE GOAL

Find all unique [a, b, c] where a + b + c = 0.

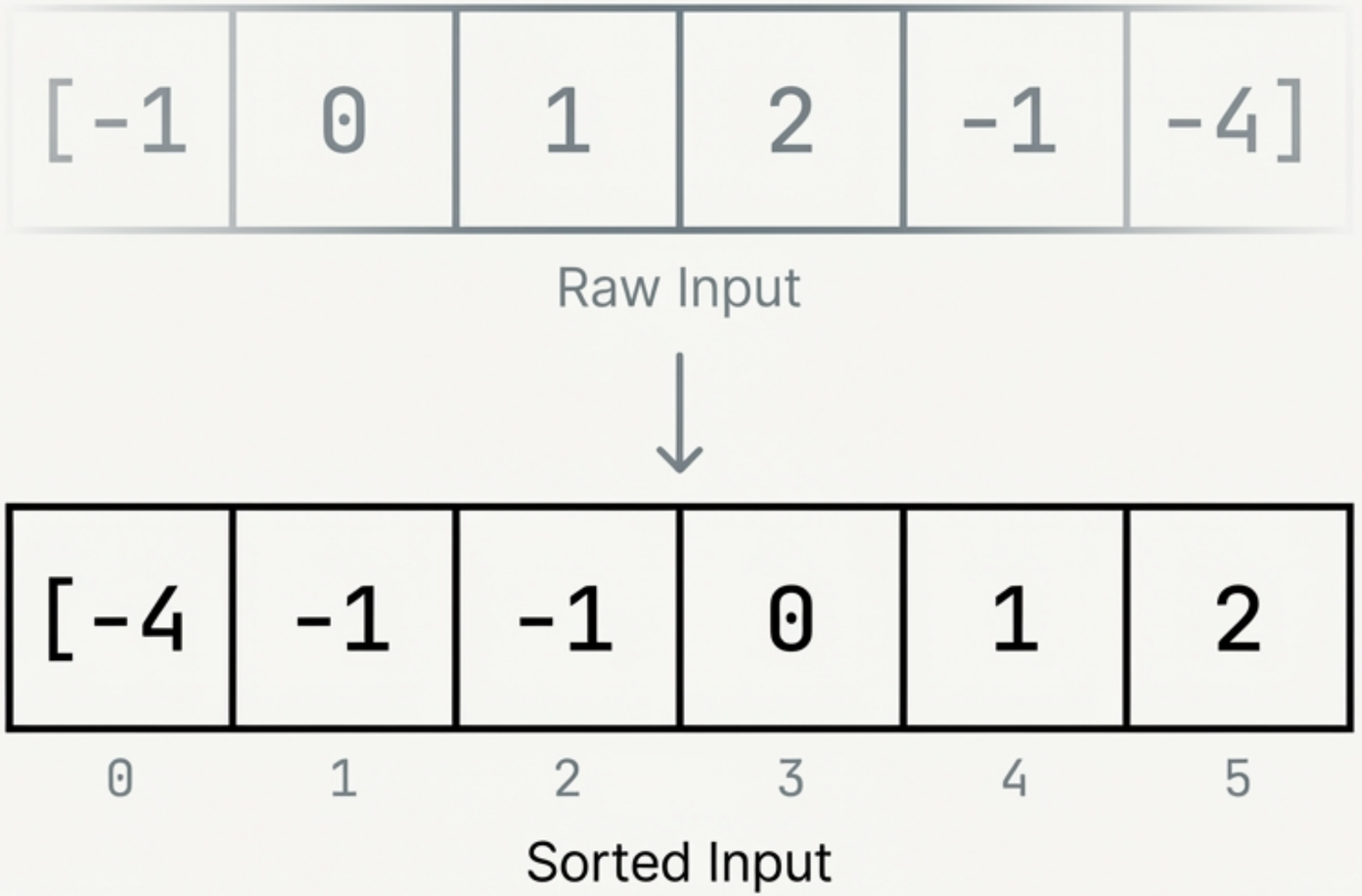


THE TRAP

Checking every combination requires three nested loops.

Complexity: $O(N^3)$

Step 1: Sort the Input

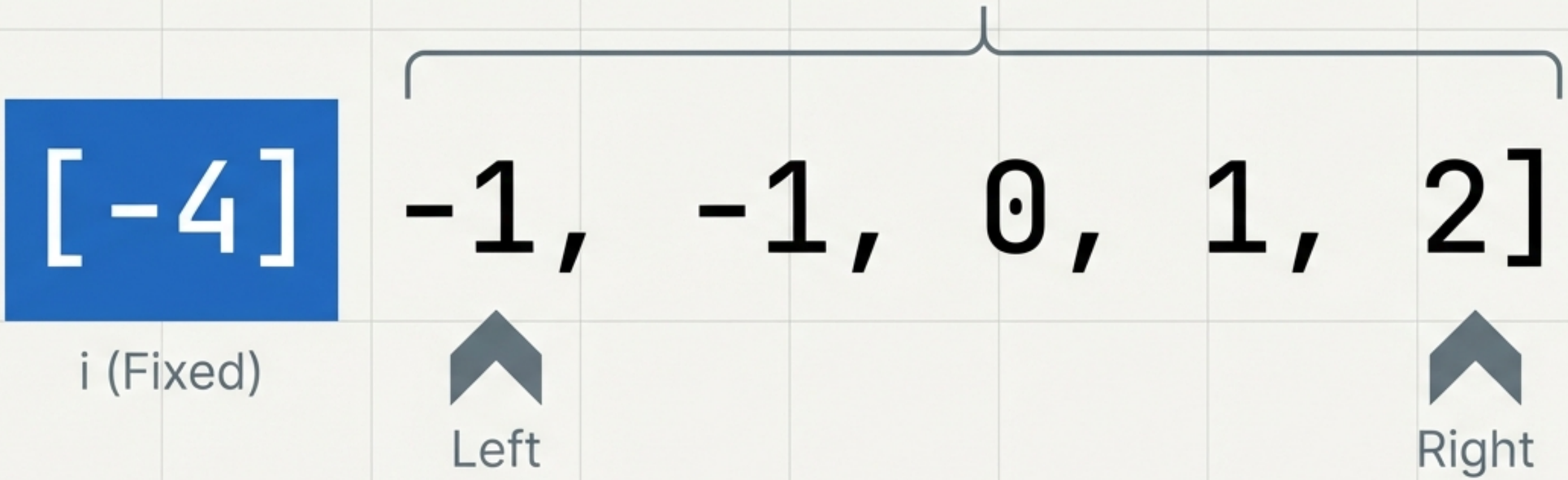



```
nums.sort()
```

Sorting unlocks predictable decision-making. If the sum is too small, move right. If too large, move left.

Eliminates the need for a third loop.

The Strategy: Fix One, Solve Two

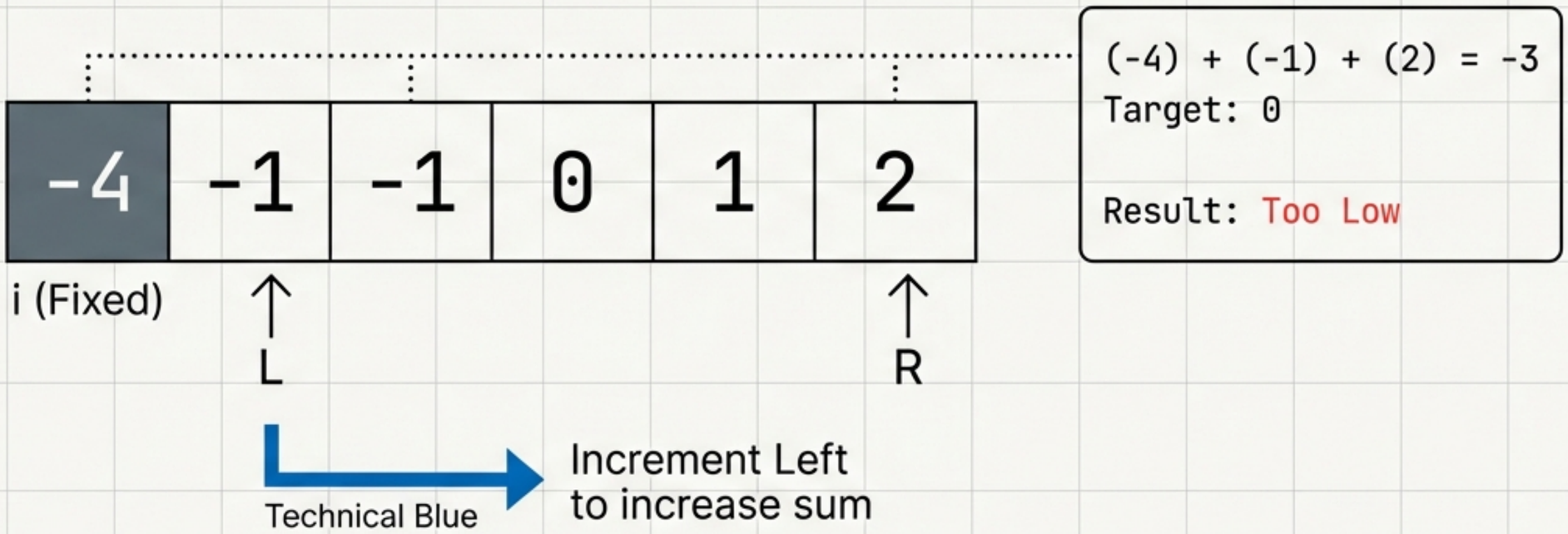


Iterate through ‘`i`’. For each step, the problem becomes finding two numbers in the remaining window that sum to `-nums[i]`.

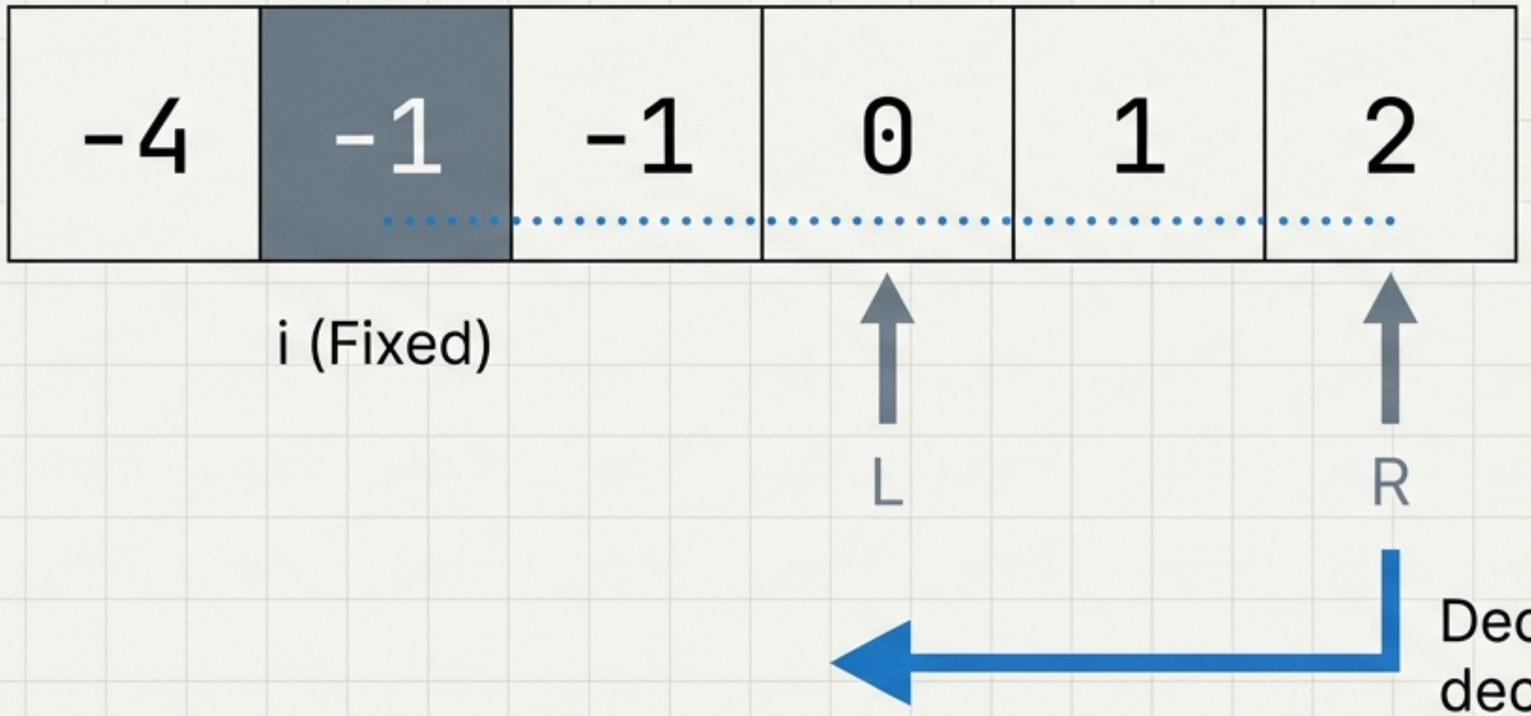
Transform the Equation

$$a + b + c = 0 \rightarrow b + c = -a$$

Scenario A: Sum is Too Small



Scenario B: Sum is Too Big

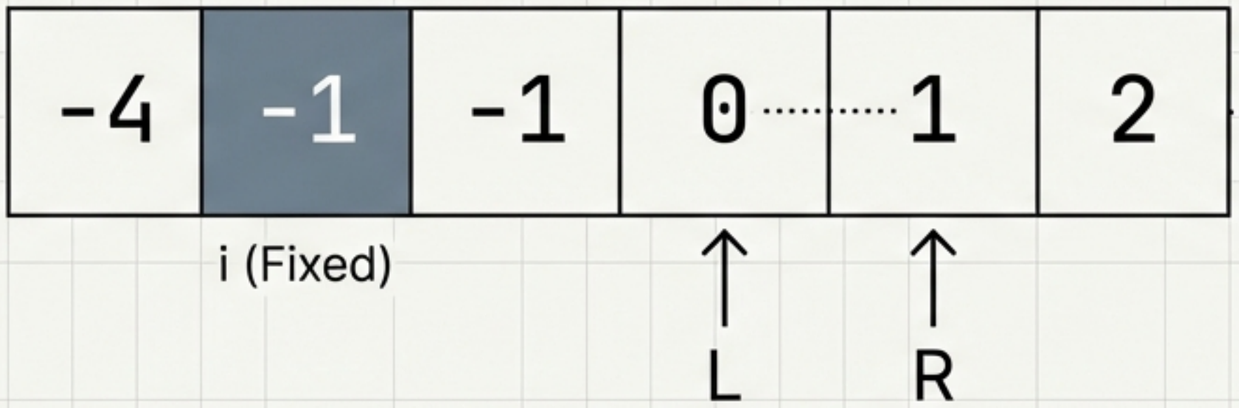


(-1) + (0) + (2) = 1

Target: 0

Result: Too High

Scenario C: Target Hit!

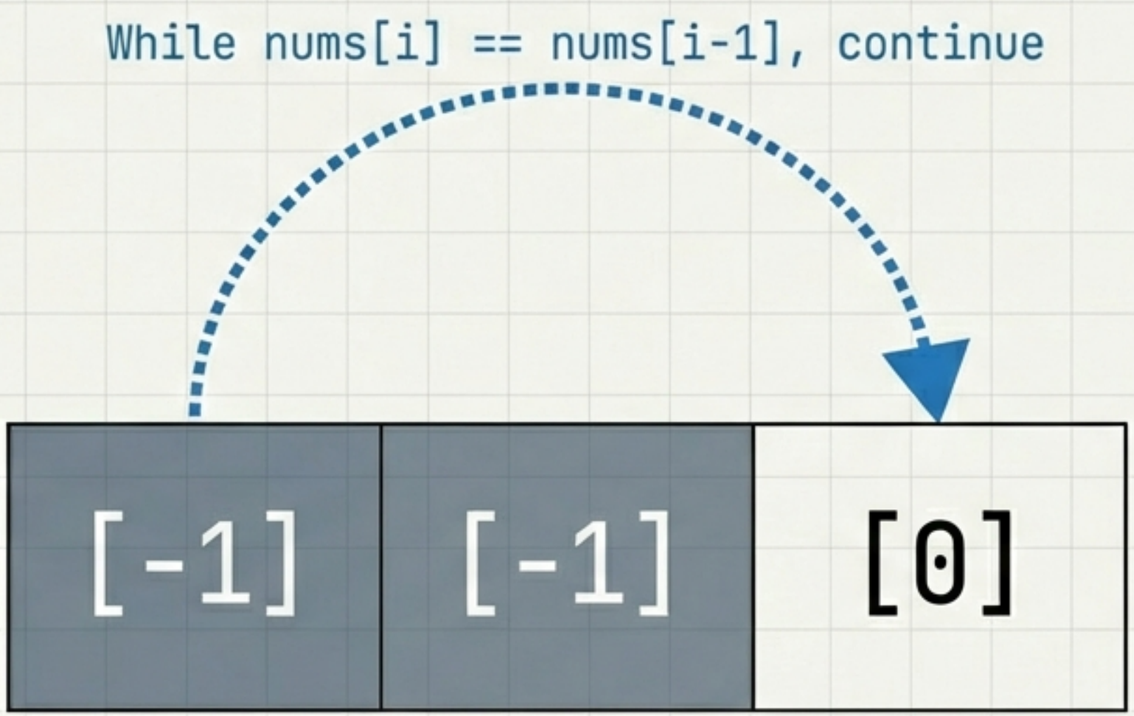


(-1) + (0) + (1) = 0

Found Triplet: [-1, 0, 1]

Record result, then pinch BOTH pointers inward to find other pairs.

The Critical Edge Case: Skipping Duplicates



The Requirement:
Return only unique triplets.

Outer Loop Logic:
Skip the current index if it is identical to the previous one. We have already solved for this value.

Inner Loop Logic:
Apply same logic to 'Left' pointer after finding a match.

The Implementation

Sort is Prerequisite

Sorting the array is the foundational step for the two-pointer approach.

Skip Outer Duplicates

This condition prevents duplicate triplets from the same starting number.

Skip Inner Duplicates

After finding a valid triplet, this loop advances the left pointer past any duplicates.

```python
def threeSum(self, nums):
    res = []
    nums.sort()  # 1

    for i, a in enumerate(nums):
        if i > 0 and a == nums[i-1]:
            continue  # 2

        l, r = i + 1, len(nums) - 1
        while l < r:
            threeSum = a + nums[l] + nums[r]
            if threeSum > 0:
                r -= 1
            elif threeSum < 0:
                l += 1
            else:
                res.append([a, nums[l], nums[r]])
                l += 1
                while nums[l] == nums[l-1] and l < r:
                    l += 1 # 3
    return res
```

Complexity & Key Takeaways

Time Complexity

$O(N^2)$

Sorting: N log N. Loops: N^2. Overall dominated by N^2.

Space Complexity

$O(1)$ or $O(N)$

Depends on sorting implementation (timsort uses O(N)).

The Recipe

1. Sort.
2. Loop 'i'.
3. Two Pointers (Left/Right).

The Pitfall

Always skip duplicates for both 'i' and 'Left' to ensure unique triplets.