# Problem Understanding

You're given an array nums and need to find **all unique triplets** [nums[i], nums[j], nums[k]] such that:

* i ≠ j ≠ k
* nums[i] + nums[j] + nums[k] == 0
* No **duplicate triplets** in the final output.

## Key Observations:

* The order of numbers in the triplet doesn't matter (e.g., [-1, 0, 1] is same as [0, -1, 1]).
* You must **avoid duplicate triplets**.
* Brute force is too slow: we need something better than O(n³).

## Optimized Java Solution (Two Pointers + Sorting)

class Solution {

public List<List<Integer>> threeSum(int[] nums) {

List<List<Integer>> result = new ArrayList<>();

Arrays.sort(nums); // Sort the array first

for (int i = 0; i < nums.length - 2; i++) {

// Skip duplicate values for i

if (i > 0 && nums[i] == nums[i - 1]) continue;

int a = nums[i];

if (a > 0) break; // Optimization: no valid triplet if a > 0

int left = i + 1;

int right = nums.length - 1;

while (left < right) {

int sum = a + nums[left] + nums[right];

if (sum < 0) {

left++;

} else if (sum > 0) {

right--;

} else {

result.add(Arrays.asList(a, nums[left], nums[right]));

// Skip duplicates

while (left < right && nums[left] == nums[left + 1]) left++;

while (left < right && nums[right] == nums[right - 1]) right--;

left++;

right--;

}

}

}

return result;

}

}

# Dry Run Using Table

### Input:

nums = [-1, 0, 1, 2, -1, -4]

### Step 1: Sort

nums = [-4, -1, -1, 0, 1, 2]

### Step 2: Iterate and Find Triplets

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| i | a=nums[i] | left | right | triplet | sum | Action |
| 0 | -4 | 1 | 5 | [-4, -1, 2] | -3 | sum < 0 → left++ |
|  |  | 2 | 5 | [-4, -1, 2] | -3 | sum < 0 → left++ |
|  |  | 3 | 5 | [-4, 0, 2] | -2 | sum < 0 → left++ |
|  |  | 4 | 5 | [-4, 1, 2] | -1 | sum < 0 → left++ |
| 1 | -1 | 2 | 5 | [-1, -1, 2] ✅ | 0 | valid → skip duplicates |
|  |  | 3 | 4 | [-1, 0, 1] ✅ | 0 | valid → skip duplicates |
|  |  | 4 | 3 |  |  | exit inner loop |
| 2 | -1 (dupe) | — | — |  |  | skip |
| 3 | 0 | 4 | 5 | [0, 1, 2] | 3 | sum > 0 → right-- |

## Time/Space Complexity

|  |  |
| --- | --- |
| Metric | Value |
| Time | **O(n²)** |
| Space (Extra) | **O(1)** |
| Space (Result) | **O(k)** for output size |

**Why O(n²)?**

* Sorting = O(n log n)
* For each of n elements, you perform a two-pointer scan → O(n²)

# Alternate Approaches

### 1. ****Brute Force (Triple nested loops)****

* Check all combinations (i, j, k)
* Check sum == 0 and use a set to remove duplicates
* **Time**: O(n³)
* ❌ Too slow for large input

### 2. ****HashSet for Two Sum****

* For each index i, convert the problem to 2Sum with target -nums[i]
* Use a HashSet to check complements
* Easier to implement but harder to deduplicate
* **Time**: O(n²)
* **Space**: O(n)

### 3. ****Two Pointers (Best Approach)****

* Sort the array
* Fix a = nums[i]
* Use two pointers to find pairs such that a + b + c == 0
* Skip duplicates to avoid repeated triplets
* ✅ Clean and efficient