LeetCode 18. 4Sum

1. Problem Title & Link

- **18. 4Sum**
- https://leetcode.com/problems/4sum/

2. Problem Statement (Short Summary)

Given an integer array nums and an integer target,
return all **unique quadruplets** [nums[a], nums[b], nums[c], nums[d]] such that:
a, b, c, and d are distinct
nums[a] + nums[b] + nums[c] + nums[d] == target
Return results in **non-descending order** with **no duplicates**.

3. Examples (Input → Output)

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

Input: nums = [2,2,2,2,2], target = 8
Output: [[2,2,2,2]]

4. Constraints

- 1 <= nums.length <= 200
- -10^9 <= nums[i] <= 10^9
- -10^9 <= target <= 10^9

5. Thought Process (Step by Step)

The **4Sum** problem is a direct extension of **3Sum**, but we must handle an extra layer carefully.

Step 1: Sort the Array

Sorting simplifies two things:

- Enables efficient duplicate handling.
- Allows ordered pointer movement (two-pointer logic inside nested loops).

Step 2: Fix Two Elements → Use Two Pointers for the Rest

1. Outer loop: fix the first number i.
2. Inner loop: fix the second number j.
3. Use two-pointer approach for the remaining two (left, right):
    - Compute sum = nums[i] + nums[j] + nums[left] + nums[right].
    - Move pointers based on comparison with target.

This structure is **$O(n^3)$** — acceptable for constraints.

Step 3: Handle Duplicates

Skip:

- Duplicate i
- Duplicate j
- Duplicate left and right after adding a valid quadruplet.

Step 4: Use Pruning (Optional Optimization)
If smallest possible sum > target → break early.
If largest possible sum < target → continue outer loop.

6. Pseudocode

```
sort(nums)
result = []

for i in range(0, n-3):
   if i > 0 and nums[i] == nums[i-1]: continue

   for j in range(i+1, n-2):
      if j > i+1 and nums[j] == nums[j-1]: continue

      left = j + 1
      right = n - 1

      while left < right:
         total = nums[i] + nums[j] + nums[left] + nums[right]

         if total == target:
            result.append([nums[i], nums[j], nums[left], nums[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 total < target:
            left += 1
         else:
            right -= 1
```

7. Code Implementation

✅ **Python**

```
class Solution:
   def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
      nums.sort()
      res = []
      n = len(nums)

      for i in range(n - 3):
         if i > 0 and nums[i] == nums[i - 1]:
            continue

         for j in range(i + 1, n - 2):
            if j > i + 1 and nums[j] == nums[j - 1]:
```

```
            continue

        left, right = j + 1, n - 1

        while left < right:
            total = nums[i] + nums[j] + nums[left] + nums[right]

            if total == target:
                res.append([nums[i], nums[j], nums[left], nums[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 total < target:
                left += 1
            else:
                right -= 1
    return res
```

✅ **Java**

```
class Solution {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        Arrays.sort(nums);
        List<List<Integer>> res = new ArrayList<>();
        int n = nums.length;

        for (int i = 0; i < n - 3; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue;

            for (int j = i + 1; j < n - 2; j++) {
                if (j > i + 1 && nums[j] == nums[j - 1]) continue;

                int left = j + 1, right = n - 1;

                while (left < right) {
                    long total = (long) nums[i] + nums[j] + nums[left] + nums[right];

                    if (total == target) {
                        res.add(Arrays.asList(nums[i], nums[j], nums[left], nums[right]));
                        while (left < right && nums[left] == nums[left + 1]) left++;
                        while (left < right && nums[right] == nums[right - 1]) right--;
                        left++;
                        right--;
                    } else if (total < target) left++;
                    else right--;
                }
```

```
        }
      }
      return res;
    }
}
```

8. Time & Space Complexity

- **Time:** $O(n^3)$
- **Space:** O(1) (ignoring output list)

9. Dry Run (Step-by-Step Execution)

👉 Input:

nums = [1,0,-1,0,-2,2], target = 0
After sorting → [-2, -1, 0, 0, 1, 2]

i	j	left	right	total	Action	Result
0 (-2)	1 (-1)	2	5	-1	total < 0 → left++	-
0 (-2)	1 (-1)	3	5	1	total > 0 → right--	-
0 (-2)	1 (-1)	3	4	0	✅ add [-2,-1,1,2]	[[-2,-1,1,2]]
0 (-2)	2 (0)	3	5	0	✅ add [-2,0,0,2]	[[-2,-1,1,2],[-2,0,0,2]]
1 (-1)	2 (0)	3	5	1	total > 0 → right--	-
1 (-1)	2 (0)	3	4	0	✅ add [-1,0,0,1]	[[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

✅ Final Output: [[-2,-1,1,2], [-2,0,0,2], [-1,0,0,1]]

10. Concept Insight Table

Core Concept	Common Use Cases	Common Traps	Builds / Next Steps
Two-Pointer Nested Search (k-Sum) — fixing (k−2) elements and using two pointers for rest.	- k-sum pattern - Subset sum variants - Combinatorial pair searches	- Missing duplicate skips - Overflow in large sums (use long in Java) - Forgetting sorted order assumption	🔷 Builds to **Generalized K-Sum (recursive)** 🔷 Connects to **two-pointer framework** 🔷 Prepares for **subset generation & pruning**

11. Common Mistakes / Edge Cases

- Not skipping duplicates → repeated quadruplets.
- Not casting to long (overflow with big integers).
- Forgetting sorted precondition (breaks logic).
- Using nested hash-based approaches → TLE.

12. Variations / Follow-Ups

- **K-Sum (General Case)** → use recursion with base case = 2Sum.
- **3Sum Closest** or **4Sum Closest** → minimize |sum − target|.
- Use this logic for **subset-sum-style filtering** in interviews.