15. 3Sum:

I/p: nums = [-1, 0, 1, 2, -1, -4]

O/p: [[-1, -1, 2], [-1, 0, 1]]

Explanation:

nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0

nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0

nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0

The distinct triplets are [-1, 0, 1] and [-1, -1, 2]

Notice that the order of output and order of triplets does not matter.

Brute force:

Check every possible triplet using three loops:

* pick nums[i]
* pick nums[j] after i
* pick nums[k] after j
* Check if their sum is 0
* Store triples (but make sure to avoid duplicates).

→ This is simplest way but it is slow.

Optimal approach (two pointers + sorting)

1. Sort the array.

Eg: [-1, 0, 1, 2, -1, -4] → [-4, -1, -1, 0, 1, 2]

sorting helps:

use two pointer technique

Avoid duplicate triplets easily.

2. Fix one number → then find 2 numbers using two pointers.

For each i (0 to n-3):

* skip duplicate values of nums[i]
* set:

  left = i + 1

  right = n - 1
* check

  sum = nums[i] + nums[left] + nums[right].

3. Based on sum:

If sum == 0 store triplet, skip duplicates, move both pointers.

If sum < 0 move left++ (to get bigger sum)

If sum > 0 move right-- (to get smaller sum).

```
import java.util.*;
class Solution {
public List<List<Integer>> threeSum(int[] nums){

    Arrays.sort(nums);
    List<List<Integer>> result = new ArrayList<>();
for(int i = 0; i< nums.length-2; i++){
    if (i> 0 && nums[i] == nums[i-1] continue;
int left = i+1;
int right = nums.length-1;
while (left < right){
int sum = nums[i] + nums[left] + nums[right].
```

```
if (sum == 0) {
    result.add(Arrays.asList(nums[i], 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 (sum < 0){
    left++;
  } else {
    right--;
  }
 }
}
return result;
}
}
```

Time → $O(n^3)$ space → $O(1)$.