259. 3Sum Smaller

Two Pointers Binary Search Sorting

## **Problem Description**

Medium Array

triplets (i, j, k), where i, j, and k are the indices in the array such that 0 <= i < j < k < n, and the sum of the elements at these indices is less than the given target. More formally, we want to find the count of combinations where nums[i] + nums[j] + nums[k] < target.

The problem provides us with an array of integers nums and another integer target. Our task is to count the number of unique

To solve this problem, the idea is to first sort the given array. Sorting the array helps because it allows us to use the two-pointer technique effectively. Once the array is sorted, we use a for-loop to iterate over the array, fixing one element at a time. For each

### Intuition

element at index i, we then set two pointers: one at j = i + 1 (the next element) and the other at k = n - 1 (the last element). With this setup, we can check the sum s = nums[i] + nums[j] + nums[k]. If the sum s is greater than or equal to target, we move the k pointer to the left as we need a smaller sum. If the sum s is less than target, we have found a triplet that satisfies

the condition. Moreover, due to sorting, we know that all elements between j and k will also form valid triplets with i and j. This is because nums[k] is the largest possible value and nums[j] + nums[i] will only be smaller with any j' such that j < j' < k. Therefore, we can add k - j to our answer and then move the j pointer to the right, looking for the next valid triplet.

We repeat this process until j and k meet, and continue to the next i until we have considered all elements as the first element in the triplet. The ans variable accumulates the count of valid triplets throughout the whole process, and it's returned as the final

**Solution Approach** The given solution utilizes a sorted array and the two-pointer technique to find the number of index triplets that satisfy the given

## Here is a step-by-step breakdown of the solution implementation:

count once we've exhausted all possibilities.

condition nums[i] + nums[j] + nums[k] < target.</pre>

predictably move pointers based on the sum compared to the target.

nums [k] with the target. The process for this comparison is:

the condition is met, which counts all valid j to k pairings with i.

more triplet combinations with a new j and the same i.

Sorting the Array: Before the algorithm begins with its primary logic, it sorts the array with the sort() method, which enables the use of the two-pointer technique effectively. Sorting is essential because it orders the elements, allowing us to

element with i. **Initializing Pointers:** For every position i in the array, the algorithm initializes two pointers j and k. The j pointer starts just

Iterating Through the Array: The solution involves a for-loop that goes over each element of the array. It indexes each

- after i (i.e., i + 1) and k starts at the end of the array (i.e., n 1 where n is the length of the array). Using the Two-Pointer Technique: The main logic resides in a while-loop that compares the sum of nums[i], nums[j], and
- Else if the sum is less than the target (s < target), it means that all combinations of i, j, and any index between j and k will also have a sum less than the target, since nums[k] is the maximum possible value and replacing k with any index less than k would only make the sum smaller. Thus, we can directly add the count of these valid combinations (k - j) to our overall answer ans and increment j to find

• If the sum is greater than or equal to the target (s >= target), we decrement k because the array is sorted and we need a smaller sum.

Count Accumulation: The ans variable is updated every time valid triplets are found. This is done by adding k - j each time

**Returning the Result:** After all elements have been considered for i and all valid j and k pairs have been explored, the ans holds the final count of valid triplets. The function then returns ans.

This solution is efficient because it avoids the need to check every possible triplet combination individually, which would have a

time complexity of O(n^3). Instead, by sorting the array and using the two-pointer technique, the solution brings down the

- complexity to O(n^2), making it much more efficient for large arrays. **Example Walkthrough** 
  - **Sorting the Array:** First, we sort the array to get [0, 1, 2, 3]. Iterating Through the Array: Start a for-loop with index i. Initially, i = 0, and nums [i] is 0.

Using the Two-Pointer Technique: The sum of the current elements is s = nums[i] + nums[j] + nums[k] = 0 + 1 + 3 = 4.

• Since s < target, we can include not just nums[j] but any number between nums[j] and nums[k] with nums[i] to form a valid triplet. So,

Let's say we have an array nums with elements [3, 1, 0, 2] and our target is 5. Following the provided solution approach:

all valid triplets.

and efficient solution.

nums.sort()

count = 0

class Solution:

we add k - j = 3 - 1 = 2 to our answer ans. Our answer now holds 2, and we move j to the right. Now j points to 2, and we repeat the check:

Initializing Pointers: Set j = i + 1, which is 1, and k = n - 1, which is 3. Now j points to 1 and k to 3.

Since s >= target, no valid triplet can be formed with the current j and k. Hence, we move the k pointer to the left.

def threeSumSmaller(self, nums: List[int], target: int) -> int:

# Calculate the sum of the current triplet

# to the left to reduce the sum

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

# Return the total count of triplets with the sum smaller than target

# Sort the input array to use the two-pointer approach effectively

# Initialize the count of triplets with the sum smaller than target

# Use two pointers to find the pair whose sum with nums[i] is smaller than target

# If the sum is smaller than target, all elements between left and right

# If the sum is equal to or greater than target, move the right pointer

# form valid triplets with nums[i] because the array is sorted

 $\circ$  s = nums[i] + nums[j] + nums[k] = 0 + 2 + 3 = 5.

Count Accumulation: When i = 1, j = 2, and k = 3, we continue in the same manner. s = nums[i] + nums[j] + nums[k]= 1 + 2 + 3 = 6. This is not less than target, so we decrement k. Eventually, j and k will meet, and since no valid triplets

Now k points to 2 and j equals k, so we stop this iteration and move on to the next value of  $\mathbf{i}$ .

are found for this i, ans remains 2.

Returning the Result: Continue this process until all elements have been considered for i. Finally, ans contains the count of

In conclusion, for our example [3, 1, 0, 2] with the target of 5, after iterating through the sorted array [0, 1, 2, 3], the final

number of valid triplets less than 5 is 2. This demonstrates how using a sorted array and the two-pointer approach yields a quick

- Solution Implementation **Python** from typing import List
- # Get the length of nums n = len(nums)# Iterate through the array. Since we are looking for triplets, we stop at n-2for i in range(n - 2): # Initialize two pointers, one after the current element and one at the end

### if triplet sum < target:</pre> # Add the number of valid triplets to the count count += right - left # Move the left pointer to the right to look for new triplets

else:

left, right = i + 1, n - 1

left += 1

right -= 1

while left < right:</pre>

```
return count
Java
class Solution {
    public int threeSumSmaller(int[] numbers, int target) {
        // Sort the input array to make it easier to navigate.
        Arrays.sort(numbers);
        // Initialize the count of triplets with sum smaller than the target.
        int count = 0;
        // Iterate over the array. The outer loop considers each element as the first element of the triplet.
        for (int firstIndex = 0; firstIndex < numbers.length; ++firstIndex) {</pre>
            // Initialize two pointers,
            // 'secondIndex' just after the current element of the first loop ('firstIndex + 1'),
            // 'thirdIndex' at the end of the array.
            int secondIndex = firstIndex + 1;
            int thirdIndex = numbers.length - 1;
            // Use a while loop to find pairs with 'secondIndex' and 'thirdIndex' such that their sum with 'numbers[firstIndex]'
            // is less than the target.
            while (secondIndex < thirdIndex) {</pre>
                int sum = numbers[firstIndex] + numbers[secondIndex] + numbers[thirdIndex];
                // If the sum is greater than or equal to the target, move the 'thirdIndex' pointer
                // to the left to reduce sum.
                if (sum >= target) {
                    --thirdIndex;
                } else {
                    // If the sum is less than the target, count all possible third elements by adding
                    // the distance between 'thirdIndex' and 'secondIndex' to the 'count'
                    // because all elements to the left of 'thirdIndex' would form a valid triplet.
                    count += thirdIndex - secondIndex;
                    // Move the 'secondIndex' pointer to the right to find new pairs.
                    ++secondIndex;
        // Return the total count of triplets.
        return count;
C++
```

```
};
TypeScript
/**
```

#include <vector>

class Solution {

public:

#include <algorithm> // Required for the std::sort function

std::sort(numbers.begin(), numbers.end());

for (int i = 0; i < numbers.size(); ++i) {</pre>

int right = numbers.size() - 1;

// Sort the input array

int left = i + 1;

while (left < right) {</pre>

if (sum < target) {</pre>

++left;

--right;

} else {

return count;

\* smaller than the `target`.

int threeSumSmaller(std::vector<int>& numbers, int target) {

int count = 0; // Initialize the count of triplets

// Initiate the second and third pointers

count += right - left;

// Return the total count of triplets found

\* Counts the number of triplets in the array `nums` that sum to a value

// Calculate the sum of the current triplet

// Iterate through each element in the array, treating it as the first element of the triplet

// If the sum is smaller, all elements between the current second pointer

// and third pointer will form valid triplets, add them to the count

// Move the second pointer to the right to explore other possibilities

// Iterate while the second pointer is to the left of the third pointer

int sum = numbers[i] + numbers[left] + numbers[right];

// Check if the summed value is less than the target value

// If the sum is greater than or equal to the target,

// move the third pointer to the left to reduce the sum

```
* @param {number[]} nums - The array of numbers to check for triplets.
* @param {number} target - The target sum that triplets should be less than.
* @return {number} - The count of triplets with a sum less than `target`.
function threeSumSmaller(nums: number[], target: number): number {
   // First, sort the array in non-decreasing order.
   nums.sort((a, b) => a - b);
   // Initialize the answer to 0.
    let answer: number = 0;
   // Iterate through each number in the array, using it as the first number in a potential triplet.
    for (let i: number = 0, n: number = nums.length; i < n; ++i) {</pre>
        // Initialize two pointers, one starting just after i, and one at the end of the array.
        let i: number = i + 1:
        let k: number = n - 1;
       // As long as i is less than k, try to find valid triplets.
       while (j < k) {
            // Calculate the sum of the current triplet.
            let sum: number = nums[i] + nums[j] + nums[k];
           // If the sum is greater than or equal to the target, we need to reduce it, so decrement k_{ullet}
            if (sum >= target) {
                --k;
            } else {
                // Otherwise, all triplets between j and k are valid, so add them to the answer.
                answer += k - j;
                // Increment j to check for the next potential triplet.
                ++j;
   // Return the total count of valid triplets.
   return answer;
from typing import List
class Solution:
   def threeSumSmaller(self, nums: List[int], target: int) -> int:
       # Sort the input array to use the two-pointer approach effectively
       nums.sort()
       # Initialize the count of triplets with the sum smaller than target
        count = 0
       # Get the length of nums
        n = len(nums)
       # Iterate through the array. Since we are looking for triplets, we stop at n - 2
        for i in range(n - 2):
            # Initialize two pointers, one after the current element and one at the end
            left, right = i + 1, n - 1
           # Use two pointers to find the pair whose sum with nums[i] is smaller than target
```

# Time and Space Complexity

which simplifies to O(n^2).

return count

else:

while left < right:</pre>

if triplet sum < target:</pre>

left += 1

right -= 1

count += right - left

# Calculate the sum of the current triplet

# to the left to reduce the sum

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

# If the sum is smaller than target, all elements between left and right

# Move the left pointer to the right to look for new triplets

worst-case time complexity of  $O(n \log n)$ , where n is the length of the input list nums.

# If the sum is equal to or greater than target, move the right pointer

# form valid triplets with nums[i] because the array is sorted

# Add the number of valid triplets to the count

# Return the total count of triplets with the sum smaller than target

## The given Python code sorts the input list and then uses a three-pointer approach to find triplets of numbers that sum up to a

**Time Complexity** 

value smaller than the target. Sorting the Array: The sort() method used on the array is based on Timsort algorithm for Python's list sorting, which has a

Three-Pointer Approach: The algorithm uses a for loop combined with a while loop to find all possible triplets that meet the condition. For each element in the list (handled by the for loop), the while loop can iterate up to n - i - 1 times in the worst

case. As i ranges from 0 to n - 1, the total number of iterations across all elements is less than or equal to n/2 \* (n - 1),

**Space Complexity** 

Combining both complexities, since  $0(n \log n)$  is overshadowed by  $0(n^2)$ , the overall time complexity of the code is  $0(n^2)$ .

constant factors. Therefore, it has a space complexity of 0(1). Variable Storage: The algorithm uses a constant amount of additional space for variables ans, n, i, j, k, and s. This does

Extra Space for Sorting: The sort() method sorts the list in place and thus does not use extra space except for some

Hence, the total space complexity of the code is 0(1), as it only requires a constant amount of space besides the input list.

not depend on the size of the input and therefore also contributes 0(1) to the space complexity.