<https://leetcode.ca/all/259.html>

Given an array of n integers nums and a target, find the number of index triplets i, j, k with 0 <= i < j < k < n that satisfy the condition nums[i] + nums[j] + nums[k] < target.

**Example:**

Input: nums = [-2,0,1,3], and target = 2

Output: 2

Explanation: Because there are two triplets which sums are less than 2:

  [-2,0,1]

[-2,0,3]

**Follow up:** Could you solve it in *O*(*n^*2) runtime?

**Attempt 1: 2023-03-03**

**Solution 1: Two Pointers (10 min)**

class Solution {

    public int threeSumSmaller(int[] numbers, int target) {

        Arrays.sort(numbers);

        int count = 0;

        int n = numbers.length;

        for(int i = 0; i < n; i++) {

            int lo = i + 1;

            int hi = n - 1;

            while (lo < hi) {

                int sum = numbers[i] + numbers[lo] + numbers[hi];

                if (sum >= target) {

                    hi--;

                } else {

                    count += hi - lo;

                    lo++;

                }

            }

        }

        return count;

    }

}

Time Complexity: O(N^2)

Space Complexity: O(1)

**Refer to**

<https://algo.monster/liteproblems/259>

**Problem Description**

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

**Intuition**

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 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 count once we've exhausted all possibilities.

**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 condition nums[i] + nums[j] + nums[k] < target.

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

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

Iterating Through the Array: The solution involves a for-loop that goes over each element of the array. It indexes each 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 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 nums[k] with the target. The process for this comparison is:

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.

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

Count Accumulation: The ans variable is updated every time valid triplets are found. This is done by adding k - j each time the condition is met, which counts all valid j to k pairings with i.

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**

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

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.

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.

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, 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:

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

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

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

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 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 all valid triplets.

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 and efficient solution.

**Java Solution**

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) {

            // 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) {

                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;

    }

}

**Time and Space Complexity**

**Time Complexity**

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

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), which simplifies to O(n^2).

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

**Space Complexity**

Extra Space for Sorting: The sort() method sorts the list in place and thus does not use extra space except for some constant factors. Therefore, it has a space complexity of O(1).

Variable Storage: The algorithm uses a constant amount of additional space for variables ans, n, i, j, k, and s. This does not depend on the size of the input and therefore also contributes O(1) to the space complexity.

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