

259. 3Sum Smaller

Medium

Array

Two Pointers

Binary Search

Sorting

Leetcode Link

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 $0 \leq i < j < k < n$, 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.

Python Solution

```
1 from typing import List
2
3 class Solution:
4     def threeSumSmaller(self, nums: List[int], target: int) -> int:
5         # Sort the input array to use the two-pointer approach effectively
6         nums.sort()
7         # Initialize the count of triplets with the sum smaller than target
8         count = 0
9         # Get the length of nums
10        n = len(nums)
11
12        # Iterate through the array. Since we are looking for triplets, we stop at n - 2
13        for i in range(n - 2):
14            # Initialize two pointers, one after the current element and one at the end
15            left, right = i + 1, n - 1
16            # Use two pointers to find the pair whose sum with nums[i] is smaller than target
17            while left < right:
18                # Calculate the sum of the current triplet
19                triplet_sum = nums[i] + nums[left] + nums[right]
20                # If the sum is smaller than target, all elements between left and right
21                # form valid triplets with nums[i] because the array is sorted
22                if triplet_sum < target:
23                    # Add the number of valid triplets to the count
24                    count += right - left
25                    # Move the left pointer to the right to look for new triplets
26                    left += 1
27                else:
28                    # If the sum is equal to or greater than target, move the right pointer
29                    # to the left to reduce the sum
30                    right -= 1
31
32        # Return the total count of triplets with the sum smaller than target
33        return count
34
```

Java Solution

```
1 class Solution {
2     public int threeSumSmaller(int[] numbers, int target) {
3         // Sort the input array to make it easier to navigate.
4         Arrays.sort(numbers);
5
6         // Initialize the count of triplets with sum smaller than the target.
7         int count = 0;
8
9         // Iterate over the array. The outer loop considers each element as the first element of the triplet.
10        for (int firstIndex = 0; firstIndex < numbers.length; ++firstIndex) {
11            // Initialize two pointers,
12            // 'secondIndex' just after the current element of the first loop ('firstIndex + 1'),
13            // 'thirdIndex' at the end of the array.
14            int secondIndex = firstIndex + 1;
15            int thirdIndex = numbers.length - 1;
16
17            // Use a while loop to find pairs with 'secondIndex' and 'thirdIndex' such that their sum with 'numbers[firstIndex]'
18            // is less than the target.
19            while (secondIndex < thirdIndex) {
20                int sum = numbers[firstIndex] + numbers[secondIndex] + numbers[thirdIndex];
21
22                // If the sum is greater than or equal to the target, move the 'thirdIndex' pointer
23                // to the left to reduce sum.
24                if (sum >= target) {
25                    --thirdIndex;
26                } else {
27                    // If the sum is less than the target, count all possible third elements by adding
28                    // the distance between 'thirdIndex' and 'secondIndex' to the 'count'
29                    // because all elements to the left of 'thirdIndex' would form a valid triplet.
30                    count += thirdIndex - secondIndex;
31
32                    // Move the 'secondIndex' pointer to the right to find new pairs.
33                    ++secondIndex;
34                }
35            }
36        }
37
38        // Return the total count of triplets.
39        return count;
40    }
41 }
42
```

C++ Solution

```
1 #include <vector>
2 #include <algorithm> // Required for the std::sort function
3
4 class Solution {
5 public:
6     int threeSumSmaller(std::vector<int>& numbers, int target) {
7         // Sort the input array
8         std::sort(numbers.begin(), numbers.end());
9
10        int count = 0; // Initialize the count of triplets
11        // Iterate through each element in the array, treating it as the first element of the triplet
12        for (int i = 0; i < numbers.size(); ++i) {
13            // Initiate the second and third pointers
14            int left = i + 1;
15            int right = numbers.size() - 1;
16
17            // Iterate while the second pointer is to the left of the third pointer
18            while (left < right) {
19                // Calculate the sum of the current triplet
20                int sum = numbers[i] + numbers[left] + numbers[right];
21                // Check if the summed value is less than the target value
22                if (sum < target) {
23                    // If the sum is smaller, all elements between the current second pointer
24                    // and third pointer will form valid triplets, add them to the count
25                    count += right - left;
26
27                    // Move the second pointer to the right to explore other possibilities
28                    ++left;
29                } else {
30                    // If the sum is greater than or equal to the target,
31                    // move the third pointer to the left to reduce the sum
32                    --right;
33                }
34            }
35        }
36
37        // Return the total count of triplets found
38        return count;
39    };
40 };
```

Typescript Solution

```
1 /**
2  * Counts the number of triplets in the array `nums` that sum to a value
3  * smaller than the `target`.
4  *
5  * @param {number[]} nums - The array of numbers to check for triplets.
6  * @param {number} target - The target sum that triplets should be less than.
7  * @return {number} - The count of triplets with a sum less than `target`.
8  */
9 function threeSumSmaller(nums: number[], target: number): number {
10    // First, sort the array in non-decreasing order.
11    nums.sort((a, b) => a - b);
12
13    // Initialize the answer to 0.
14    let answer: number = 0;
15
16    // Iterate through each number in the array, using it as the first number in a potential triplet.
17    for (let i: number = 0; n: number = nums.length; i < n; ++i) {
18        // Initialize two pointers, one starting just after i, and one at the end of the array.
19        let j: number = i + 1;
20        let k: number = n - 1;
21
22        // As long as j is less than k, try to find valid triplets.
23        while (j < k) {
24            // Calculate the sum of the current triplet.
25            let sum: number = nums[i] + nums[j] + nums[k];
26
27            // If the sum is greater than or equal to the target, we need to reduce it, so decrement k.
28            if (sum >= target) {
29                --k;
30            } else {
31                // Otherwise, all triplets between j and k are valid, so add them to the answer.
32                answer += k - j;
33                // Increment j to check for the next potential triplet.
34                ++j;
35            }
36        }
37    }
38
39    // Return the total count of valid triplets.
40    return answer;
41 };
```

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.