16. 3Sum Closest

Medium Array Two Pointers Sorting Leetcode Link

Problem Description

The LeetCode problem asks us to find three numbers within an array `nums` such that their sum is closest to a given `target` value. The array `nums` has a length `n`, and it's guaranteed that there is exactly one solution for each input.

To solve this problem, we must search for triplets in the array whose sum has the smallest absolute difference from the target. The final result is not the triplet itself, but rather the sum of its components.

Intuition

The intuition behind the provided solution leverages a sorted array and a two-pointer technique for efficient searching. Here's a step-by-step breakdown of the approach:

1. **Sorting**: We first sort the `nums` array. Sorting is crucial because it allows us to efficiently sweep through the array using two pointers and easily adjust the sum of the current triplet to get closer to the `target`.

2. **Iterating with Three Pointers**: We use a for-loop to iterate over the array with an index `i` representing the first number of the triplet. The other two pointers, `j` and `k`, are initialized to the next element after `i` and the last element of the array, respectively. The three numbers represented by `i`, `j`, and `k` are our current candidates for the closest sum.

3. **Evaluating Sums and Moving Pointers**: In the while-loop, we calculate the sum of the triplet (`t`) and compare it to the `target`. If the sum exactly matches the `target`, we immediately return it.

   If the sum doesn't match, we compare the absolute difference of `t` and `target` with the current closest sum (`ans`), and if it's smaller, we update `ans`.

   To search through different sums, we move pointers `j` and `k` depending on whether the current sum is greater than or less than `target`. If it's greater, we decrement `k` to reduce the sum. If it's less, we increment `j` to increase the sum. This is possible without missing the potential answers because the array is sorted.

By analyzing the difference between `t` and `target` and adjusting `j` and `k` accordingly, we can efficiently find the triplet with the sum that has the smallest absolute difference to the `target`.

4. **Returning the Answer**: Once we've finished iterating through all possible triplets, we return the closest sum recorded as `ans`.

Solution Approach

The algorithm implementation utilizes several important concepts and patterns:

1. **Sorting**: The array is initially sorted to simplify the searching process. By having the array in ascending order, we can make use of the two-pointer technique effectively.

2. **Two-Pointer Technique**: After fixing one number of the potential triplet using the first for-loop, the two other numbers are controlled by two pointers. One starts right after the fixed element (`j`), while the other starts from the end of the array (`k`). These pointers move closer to each other as they iterate.

3. **Avoiding Redundancy**: Since each number in `nums` is used as a starting point for a triplet, the solution avoids re-examining numbers that are identical to the previous starting point (`i` position) by skipping duplicate combinations. *(This is implicit and can be added to optimization in the code if necessary by checking for the same numbers when incrementing `i`).*

4. **Closest Sum Calculation**: As the pointers `j` and `k` move, the solution calculates the sum of the three numbers, compares it with the `target`, and keeps track of the closest sum encountered by comparing the absolute differences with the current best answer (`ans`).

5. **Conditional Pointer Movement**: Based on whether the current sum is greater than or less than `target`, `k` is decremented or `j` is incremented respectively. This allows the solution to narrow down the closest sum without checking all possible triplet combinations, which would result in higher computational complexity.

6. **Early Termination**: If at any point the sum equals the target, the loop breaks and returns the sum immediately since it cannot get any closer than an exact match.

7. **Return Statement**: After going through all possible iterations, the algorithm returns the sum stored in `ans`, which is the closest sum to `target` that the solution could find during the iteration process.

The code uses a simple `inf` value to initialize `ans` so that any other sum found will be closer compared to infinity. Utilizing this approach, the data structure (a single array) is kept simple, and the algorithm achieves a time complexity of O(n^2), since it only involves iterating over the array of length `n` and adjusting the pointers without nested full iterations.

Example Walkthrough

Let's consider a small example to illustrate the solution approach using the strategy detailed in the content provided.

Suppose we have the following `nums` array and `target`:

```
nums = [-1, 2, 1, -4]
target = 1
```

Firstly, according to our approach, we need to sort the `nums`:

```
sorted_nums = [-4, -1, 1, 2]
```

Now we iterate through `sorted_nums` with our three pointers. For simplicity, I'll walk through the first complete iteration:

1. Set `i = 0`, which is the value `-4`. This is our first number of the potential triplet.
2. Initialize two other pointers, `j = i + 1` ⇒ `j = 1` (value `-1`) and `k = n - 1` ⇒ `k = 3` (value `2`).
3. We then enter a while loop with `j < k` and calculate the sum `t` using `sorted_nums[i]`, `sorted_nums[j]`, and `sorted_nums[k]`. So, our first sum is `t = (-4) + (-1) + (2) = -3`.
4. Since our goal is to get the sum close to `target` (1), we check the absolute difference between `t` and `target`. Abs(-3 - 1) = Abs(-4) = 4.
5. We initialize our answer `ans` with infinity. Our first comparison will set `ans = -3` as it's the closest sum we've encountered.
6. `t` is less than `target`, we increment `j` to increase the sum. Now, `j = 2` (value `1`).
7. Calculate a new sum: `t = (-4) + (1) + (2) = -1`.
8. Compare the new sum's absolute difference with `target`. Abs(-1 - 1) = 2, which is closer to the target than our previous best of 4. Update `ans` to `-1`.
9. Since `-1` is still less than the `target`, we increment `j` once again. Now, `j = 3` which is equal to `k`, so we break out of the while-loop.
10. The loop for the index `i` continues, but for the sake of brevity, let's assume the remaining iterations do not find a sum closer to the target than the sum when `ans` was `-1`.

With these steps completed, we assert that the closest sum to the `target` we have found using this method is `-1`. As per our algorithm's implementation, `-1` will be the final answer returned.

Note that for a real implementation, the process would involve iterating over all valid `i`, `j`, and `k` combinations until the end of the array is reached, continuously updating `ans` if closer sums are found, and potentially breaking early if an exact match is found. However, this simple example serves to convey the essentials of the solution approach.

Python Solution

```python
class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        # Sort the list to apply the two-pointer approach
        nums.sort()
        n = len(nums)

        # Initialize the answer with infinity to ensure any sum will be closer to target
        closest_sum = float('inf')

        # Iterate through each number in the sorted list
        for i in range(n - 2):  # last two elements will be covered by the two pointers
            left_pointer = i + 1
            right_pointer = n - 1

            # Use two pointers to find the closest sum for the current element nums[i]
            while left_pointer < right_pointer:
                current_sum = nums[i] + nums[left_pointer] + nums[right_pointer]

                # If the sum is exactly the target, return the sum immediately
                if current_sum == target:
                    return current_sum

                # Update the closest sum if the current one is closer to target
                if abs(current_sum - target) < abs(closest_sum - target):
                    closest_sum = current_sum

                # Move the pointers accordingly to get closer to target
                if current_sum > target:
                    right_pointer -= 1  # Decrease sum by moving right pointer left
                else:
                    left_pointer += 1  # Increase sum by moving left pointer right

        # Return the closest sum after checking all triples
        return closest_sum
```

Please note that in the comments as well as the code, `List` and `inf` are assumed to be imported from the appropriate modules, which should be included when running the code in a complete script:

```python
from typing import List
from math import inf

```

Java Solution

```java
// Class name should be descriptive and use PascalCase
class Solution {
    // Method names in camelCase, which is already followed here
    public int threeSumClosest(int[] nums, int target) {
        // Sort the array to have numbers in ascending order
        Arrays.sort(nums);

        // Initialize the answer with a large value for comparison purposes
        int closestSum = Integer.MAX_VALUE;

        // The length of the numbers array
        int length = nums.length;

        // Iterate through each number in the array
        for (int i = 0; i < length; ++i) {
            // Initialize two pointers, one just after the current number and one at the end of the array
            int left = i + 1;
            int right = length - 1;

            // Continue as long as the left pointer is less than the right pointer
            while (left < right) {
                // Calculate the current sum of the three numbers
                int currentSum = nums[i] + nums[left] + nums[right];

                // If the current sum is equal to the target, return it immediately as the closest sum
                if (currentSum == target) {
                    return currentSum;
                }

                // If the current sum is closer to the target than the previous sum,
                // update closestSum with the current sum
                if (Math.abs(currentSum - target) < Math.abs(closestSum - target)) {
                    closestSum = currentSum;
                }

                // Move pointers based on how currentSum compares to the target
                if (currentSum > target) {
                    // If currentSum is greater than the target,
                    // move the right pointer to the left to reduce the sum
                    --right;
                } else {
                    // If currentSum is less than the target,
                    // move the left pointer to the right to increase the sum
                    ++left;
                }
            }
        }

        // Return the closest sum found
        return closestSum;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>  // for sort function
#include <cstdlib>    // for abs function

class Solution {
public:
    int threeSumClosest(std::vector<int>& nums, int target) {
        // First, sort the input vector in non-decreasing order
        std::sort(nums.begin(), nums.end());

        // Initialize the closest sum to a very large value
        int closestSum = INT_MAX;
        // Calculate the size of the input vector
        int numSize = nums.size();

        // Iterate through each element of the vector
        for (int i = 0; i < numSize; ++i) {
            // Set pointers for the current element, next element, and the last element
            int leftPointer = i + 1, rightPointer = numSize - 1;

            // Use a while loop to check the sums with the current element
            while (leftPointer < rightPointer) {
                // Calculate the sum of the three elements
                int sum = nums[i] + nums[leftPointer] + nums[rightPointer];

                // If the exact sum is found, return it
                if (sum == target) {
                    return sum;
                }

                // Update closestSum if the current sum is closer to the target
                if (std::abs(sum - target) < std::abs(closestSum - target)) {
                    closestSum = sum;
                }

                // Move the right pointer left if the sum is greater than the target
                if (sum > target) {
                    --rightPointer;
                } else {
                    ++leftPointer;  // Otherwise, move the left pointer right
                }
            }
        }

        // Return the closest sum found
        return closestSum;
    }
};

```

Typescript Solution

```typescript
function threeSumClosest(nums: number[], target: number): number {
    // Sort the array in non-decreasing order
    nums.sort((a, b) => a - b);

    // Initialize the answer with the maximum safe integer value
    let closestSum: number = Number.MAX_SAFE_INTEGER;

    // Get the length of the nums array
    const length = nums.length;

    // Iterate over the array to find the three numbers
    for (let i = 0; i < length; ++i) {
        // Set pointers for the current element, next element, and the last element
        let left = i + 1;
        let right = length - 1;

        // While the left pointer is less than the right pointer
        while (left < right) {
            // Calculate the current sum of the triplets
            const currentSum: number = nums[i] + nums[left] + nums[right];

            // If the current sum exactly equals the target, return the current sum
            if (currentSum === target) {
                return currentSum;
            }

            // If the current sum is closer to the target than the previous closest, update the closestSum
            if (Math.abs(currentSum - target) < Math.abs(closestSum - target)) {
                closestSum = currentSum;
            }

            // If the current sum is greater than the target, move the right pointer to find a smaller sum
            if (currentSum > target) {
                --right;
            } else { // If the current sum is less, move the left pointer to find a larger sum
                ++left;
            }
        }
    }

    // Return the closest sum found
    return closestSum;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given function depends on a few distinct steps:

1. Sorting the array: The `sort()` method in Python uses the Timsort algorithm, which has a time complexity of `O(n log n)`, where `n` is the length of the list being sorted.

2. The for-loop: The loop runs for each element in the sorted array, resulting in a complexity of `O(n)` iterations.

3. The while-loop inside the for-loop: In the worst case, for each iteration of the for-loop, the while-loop can perform nearly `O(n)` operations since it might iterate from `i+1` to the `n-1` index.

Combining these complexities, the first step is dominant if `n` is large. However, since the nested loop inside the for-loop could potentially run `n` times for each `n` iterations of the for-loop, the resulting time complexity is `O(n^2)`, since `n(n-1)/2` simplifies to `O(n^2)`. This nested loop is the most influential factor for large `n`.

So, the overall time complexity of the algorithm is `O(n log n) + O(n^2)`, which simplifies to `O(n^2)` since the `O(n^2)` term is dominant for large `n`.

Space Complexity

The space complexity is `O(1)` if we ignore the space used for input and output since the sorting is done in-place and only a fixed number of variables are used, which does not scale with the size of the input array.