Problem Description (2 points)

- I solved the 3Sum Closest problem. The objective was to find three integers in an array whose sum is closest to a given target value.
- Requirements: Given an integer array and a target integer, find a triplet in the array whose sum is closest to the target.Constraints: the array can contain positive and negative integers. The array length could vary, and edge cases like arrays with fewer than three elements must be handled. Necessary Data Structures: The problem required sorting the array to enable an efficient two-pointer approach. No other advanced data structures were necessary.
- Memory Allocation: No explicit dynamic memory allocation was necessary beyond temporary variables for pointers and comparisons.

Solution Description (4 points)

- I approached the problem by first sorting the input array, then iteratively fixing one element at a time and using two pointers (one at the element right after the fixed element, and one at the array's end) to find the closest sum to the target. By adjusting pointers based on whether the current sum was greater or less than the target, I could approach the closest possible sum efficiently.
- Sorting introduced an O(n log n) overhead initially, but this enabled a more efficient O(n^2) two-pointer search rather than a naive O(n^3) brute force search. Additionally, I overlooked one edge case, that is when input arrays contain fewer than three elements.

C Code Submission (2 points)

```
int threeSumClosest(int* nums, int numsSize, int target) {
    if (numsSize < 3) return 0;  // or handle according to interviewer's instruction
    qsort(nums, numsSize, sizeof(int), compare);
    int closestSum = nums[0] + nums[1] + nums[2];
    for (int i = 0; i < numsSize - 2; i++) {
        int left = i + 1;
        int right = numsSize - 1;
        while (left < right) { // Initially forgot to add this condition
            int currentSum = nums[i] + nums[left] + nums[right];
            if (abs(currentSum - target) < abs(closestSum - target)) {
                closestSum = currentSum;
            }
            if (currentSum > target) {
                right--;
            } else if (currentSum < target) {
```

```
            left++;
        } else {
            return currentSum;
        }
    }
  }
  return closestSum;
}

int compare(const void* a, const void* b) {
  return (*(int*)a - *(int*)b);
}
```

Complexity Analysis (2 points)

Time Complexity: O(n^2)

Sorting takes O(n \log n) time initially.

After sorting, two nested loops (fixing one number and two-pointer approach) contribute O(n^2).

Thus, overall complexity is dominated by O(n^2).

Space Complexity: O(1) or O(n) depending on the sorting algorithm implementation (e.g., in-place sorting would be O(1) additional space).

Explanation:

After sorting, each number is fixed once, and two pointers scan the remaining array from both ends towards each other, performing a linear number of operations each iteration. The combination of sorting and two-pointer traversal gives a total time complexity of O(n^2).

Optimality of the Solution (2 points)

The implemented solution (two-pointer after sorting) is considered optimal in terms of complexity. The brute-force approach (checking all triplets) would be O(n^3), which is inefficient. Thus, this O(n^2) solution is significantly more efficient. No known solution to this problem offers better complexity than O(n^2), making my chosen approach optimal.

Reflection on Performance (2 points)

- I successfully described and implemented the general two-pointer strategy, clearly explained my thought process, and quickly identified the correct algorithmic approach to optimize from brute force. I also successfully answered the follow up question, which is to return the elements that make up the closest 3 sum.
- I forgot a critical condition (while left < right) that significantly impacted the accuracy of my solution. Additionally, I did not proactively consider important edge cases, like arrays containing fewer than three elements or duplicates.

Strengths in Coding Interviews (2 points)

My greatest strengths in coding interviews are:

• Clearly explaining my thought process and algorithmic reasoning out loud.

• Quickly identifying appropriate algorithmic techniques (such as sorting and two-pointers).

Areas for Improvement (2 points)

Attention to detail, especially boundary conditions and loop invariants.

Proactively identifying and handling edge cases before coding.

After writing the code, demonstrate the example through my code again.

Improvement Plan (2 points)

My concrete plan to address these improvements is:

• Regularly practicing problems on platforms like LeetCode, particularly focusing on edge cases and boundary checks.

• Adopting a structured checklist approach (clarify inputs, edge cases, constraints, outputs) before writing code in interviews.

• Simulating real-time mock interviews weekly, especially focusing on avoiding careless mistakes by thoroughly thinking through loop conditions and pointer logic.