LeetCode 88: Merge Sorted Array - Interview Solution

Step 1: Restate the Problem Clearly

Let me make sure I understand the problem correctly. You're asking me to merge two sorted arrays, nums1 and nums2, into a single sorted array in-place within nums1. The first array, nums1, has enough space (i.e., of size m + n) to hold the additional elements from nums2. The first m elements of nums1 are valid, and nums2 contains n elements that should be merged into nums1.

Step 2: Ask Clarifying Questions

- Can the input arrays be empty?
- Are the arrays already sorted in non-decreasing order?
- Can the numbers be negative?
- Do we need to return a new array, or modify nums1 in-place?
- Is it guaranteed that nums1 has enough buffer to accommodate all n elements from nums2?

Step 3: Walk Through Examples

Example:

Input: nums1 = [1,2,3,0,0,0], m = 3

    nums2 = [2,5,6], n = 3

Expected Output: [1,2,2,3,5,6]

Explanation:

Start from the back of both arrays, comparing and placing larger values in-place into nums1.

Step 4: Explain the Approach

We can avoid extra space by merging from the back of nums1, starting at position m + n - 1.

We use two pointers, starting at the ends of the actual elements in nums1 and nums2.

Compare the elements and place the larger one at the insert position in nums1.

Continue until nums2 is fully merged.

Step 5: Write Pseudocode

```
1. pointer1 = m - 1
2. pointer2 = n - 1
3. insertPos = m + n - 1

While pointer1 >= 0 and pointer2 >= 0:
    If nums1[pointer1] > nums2[pointer2]:
        nums1[insertPos] = nums1[pointer1]
        pointer1--
    Else:
        nums1[insertPos] = nums2[pointer2]
        pointer2--
    insertPos--

While pointer2 >= 0:
    nums1[insertPos] = nums2[pointer2]
    pointer2--
    insertPos--
```

Step 6: Write the Actual Code

```
// Java
public class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int pointer1 = m - 1;
        int pointer2 = n - 1;
        int insertPos = m + n - 1;

        while (pointer1 >= 0 && pointer2 >= 0) {
            if (nums1[pointer1] > nums2[pointer2]) {
                nums1[insertPos] = nums1[pointer1];
                pointer1--;
            } else {
                nums1[insertPos] = nums2[pointer2];
                pointer2--;
            }
            insertPos--;
        }

        while (pointer2 >= 0) {
            nums1[insertPos] = nums2[pointer2];
            pointer2--;
            insertPos--;
        }
    }
}
```

Step 7: Test with Multiple Inputs

Test Case 1:

Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3 -> Output: [1,2,2,3,5,6]

Test Case 2:

Input: nums1 = [0], m = 0, nums2 = [1], n = 1 -> Output: [1]

Test Case 3:

Input: nums1 = [1], m = 1, nums2 = [], n = 0 -> Output: [1]

Test Case 4:

Input: nums1 = [4,5,6,0,0,0], m = 3, nums2 = [1,2,3], n = 3 -> Output: [1,2,3,4,5,6]

Step 8: Analyze Time and Space Complexity

Time Complexity: O(m + n) - We process each element once from both arrays.

Space Complexity: O(1) - We merge directly in-place without using extra memory.

Step 9: Suggest Optimizations

This solution is already optimal. It merges in-place, with linear time and constant space.

If nums1 didn't have enough space, we'd need to allocate a new array.

Step 10: Handle Follow-up Questions

Q: What if nums1 didn't have enough space?

A: Then we can't merge in-place. We'd need to create a new array and return it.

Q: What if we wanted to return a new sorted array instead?

A: We'd use two-pointer merging logic and build the result array step by step.