88. Merge Sorted Array

Easy Array Two Pointers Sorting   Leetcode Link

Problem Description

You have two sorted arrays `nums1` and `nums2` with lengths `m` and `n`, respectively. These arrays are sorted in non-decreasing order, which means each element is equal to or greater than the previous element. Initially, the array `nums1` has `m` elements in it, and an additional `n` spaces filled with `0` to allow space for merging. The array `nums2` has exactly `n` elements.

The goal is to merge these two arrays so that the `nums1` array becomes a single sorted array in non-decreasing order. The merge should happen in-place in the `nums1` array, using the additional space provided so that it can fit all `m + n` elements by the end.

Intuition

The challenge is to merge the two sorted arrays in-place, without using extra space for another array. Our intuition might suggest starting from the beginning of both arrays and comparing the elements one by one. However, this approach would require shifting elements in `nums1` to make space for elements from `nums2`, which is not efficient.

To efficiently merge these arrays, we leverage the fact that we have empty space at the end of `nums1` where `n` zeros are placed. This allows us to fill `nums1` from the end (right to left), placing the largest elements first and avoiding the need for shifting.

We use a two-pointer approach. The first pointer `i` starts at the last actual element in `nums1`, and the second pointer `j` starts at the last element in `nums2`. We also have a third pointer `k` that starts at the very end of `nums1`, and it indicates where the next element should be placed.

Each step involves comparing the elements pointed by `i` and `j`. The larger of the two elements is placed at the position indicated by `k`, and then we move the respective pointer (`i` or `j`) and `k` one step back. We repeat this process until all elements from `nums2` are placed into `nums1`. If `nums1` still has elements left when `nums2` runs out, they are already in place as the array is sorted. If `nums2` had the greatest element, it would be placed last, ensuring the non-decreasing order is maintained throughout the process.

Solution Approach

The implementation of the solution starts with the recognition that by merging the arrays from right to left, you can avoid additional space complexities and time-consuming operations like shifting elements.

The pointers `i`, `j`, and `k` are initialized as follows: `i` starts at `m - 1`, `j` at `n - 1`, and `k` at `m + n - 1`.

The algorithm is inherently a comparison between the elements pointed to by `i` and `j`. Let's walk through the steps:

- Compare the elements at pointer `i` in `nums1` and at pointer `j` in `nums2`.
- If `nums1[i]` is greater, place `nums1[i]` in `nums1[k]`, and decrement both `i` and `k`.
- If `nums2[j]` is greater or `i` is out of bounds (which means all elements of `nums1` have been placed), place `nums2[j]` in `nums1[k]`, and decrement both `j` and `k`.

This logic is concisely written in a loop that continues until all elements of `nums2` are processed, which is when `j < 0`. The condition `i >= 0 and nums1[i] > nums2[j]` ensures that you are still within bounds of `nums1` and that the current element in `nums1` is indeed larger than `nums2[j]`.

```python
while j >= 0:
    if i >= 0 and nums1[i] > nums2[j]:
        nums1[k] = nums1[i]
        i -= 1
    else:
        nums1[k] = nums2[j]
        j -= 1
    k -= 1
```

This approach does not require any extra data structures. It utilizes the space already allocated within `nums1` and takes advantage of the sorted order, allowing an efficient in-place merge. The pattern used here is commonly known as the two-pointer technique, which is often applied to problems that involve sorted arrays.

Example Walkthrough

Let's use a small example to illustrate the solution approach.

Assume our `nums1` array is `[1, 2, 3, 0, 0, 0]` with `m = 3`, and our `nums2` array is `[2, 5, 6]` with `n = 3`. The goal is to merge `nums2` into `nums1` so that it becomes a single sorted array.

Following the steps described in the solution approach:

1. Set `i` to `m - 1`, which is `2`, pointing at the last actual element in `nums1`.
2. Set `j` to `n - 1`, which is `2`, pointing at the last element in `nums2`.
3. Set `k` to `m + n - 1`, which is `5`, pointing to the last space in `nums1`.

Now we begin comparing and placing elements:

- First iteration:
  - Compare `nums1[i]` (which is `3`) and `nums2[j]` (which is `6`).
  - Since `nums2[j]` is greater, we place `6` in `nums1[k]`.
  - Decrement `j` to `1` and `k` to `4`.
  - Now, `nums1` looks like `[1, 2, 3, 0, 0, 6]`.
- Second iteration:
  - Compare `nums1[i]` (which is `3`) and `nums2[j]` (which is `5`).
  - Since `nums2[j]` is greater, we place `5` in `nums1[k]`.
  - Decrement `j` to `0` and `k` to `3`.
  - Now, `nums1` looks like `[1, 2, 3, 0, 5, 6]`.
- Third iteration:
  - Compare `nums1[i]` with `nums2[j]` (which is `2`).
  - `nums1[i]` is `3` which is greater than `nums2[j]`.
  - Place `nums1[i]` in `nums1[k]`, then decrement `i` to `1` and `k` to `2`.
  - Now, `nums1` looks like `[1, 2, 3, 3, 5, 6]`.
- Fourth iteration:
  - Compare `nums1[i]` (which is `2`) and `nums2[j]` (which is `2`).
  - They are equal, but it doesn't matter which one we choose. Here's we choose `nums2[j]`.
  - Place `nums2[j]` in `nums1[k]`, then decrement `j` to `-1` and `k` to `1`.
  - Now, `nums1` looks like `[1, 2, 2, 3, 5, 6]`.

Since `j` is now `-1`, we have placed all elements from `nums2` into `nums1`. The elements from `nums1` that have not yet been moved are already correctly placed because `nums1` was sorted to begin with. Our merged array is `[1, 2, 2, 3, 5, 6]`, which is sorted in non-decreasing order.

Python Solution

```python
from typing import List

class Solution:
    def merge(self, nums1: List[int], total_nums1: int, nums2: List[int], total_nums2: int) -> None:
        """
        Merges two sorted arrays, nums1 and nums2, into a single sorted array.
        The first array nums1 has a size sufficient to hold the contents of both arrays.
        The merge is done in-place.

        :param nums1: List[int], the first sorted array with extra space for nums2.
        :param total_nums1: int, the number of valid elements in nums1.
        :param nums2: List[int], the second sorted array to be merged into nums1.
        :param total_nums2: int, the number of elements in nums2.
        """
        # Initialize pointers for nums1 and nums2 starting from the end of their valid elements
        index_nums1, index_nums2 = total_nums1 - 1, total_nums2 - 1

        # Start filling nums1 from the end, to avoid overwriting elements of nums1 that are not yet merged
        merge_index = total_nums1 + total_nums2 - 1

        # Merge in reverse order
        while index_nums2 >= 0:
            # If nums1 is not yet exhausted and the current element is larger than nums2's, place it in the current position
            if index_nums1 >= 0 and nums1[index_nums1] > nums2[index_nums2]:
                nums1[merge_index] = nums1[index_nums1]
                index_nums1 -= 1  # Move the nums1 index backwards
            else:
                # Else, place element from nums2
                nums1[merge_index] = nums2[index_nums2]
                index_nums2 -= 1  # Move the nums2 index backwards
            merge_index -= 1  # Move the merge index backwards

```

Java Solution

```java
class Solution {
    // Merges two sorted arrays, nums1 and nums2, into a single sorted array.
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        // Initialize pointers for nums1, nums2 and the merged array.
        int indexNums1 = m - 1; // Pointer for the last element in the nums1's original part
        int indexNums2 = n - 1; // Pointer for the last element in nums2
        int mergedIndex = m + n - 1; // Pointer for the last element of merged array (end of nums1)

        // Iterate over nums2 and nums1 from the end of both arrays
        while (indexNums2 >= 0) {
            // If nums1 is exhausted, or the current element in nums2 is larger
            if (indexNums1 < 0 || nums1[indexNums1] <= nums2[indexNums2]) {
                nums1[mergedIndex] = nums2[indexNums2]; // Place nums2's element in the merged array
                indexNums2--; // Move to the next element in nums2
            } else {
                // The current element in nums1 is larger; place it in the merged array
                nums1[mergedIndex] = nums1[indexNums1];
                indexNums1--; // Move to the next element in nums1
            }
            mergedIndex--; // Move to the next position in the merged array
        }
        // No need to check the remaining elements of nums1,
        // if any left, since they are already in their sorted position.
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        // Initialize two pointers for the end of the two arrays and
        // one pointer for the end of the merged array
        int nums1Index = m - 1; // Pointer for the end of nums1
        int nums2Index = n - 1; // Pointer for the end of nums2
        int mergeIndex = m + n - 1; // Pointer for the end of the merged array (nums1)

        // Iterate through nums1 and nums2 from the end until all elements from nums2 are inserted
        while (nums2Index >= 0) {
            // If there are still elements in nums1 and the current
            // element of nums1 is larger than that of nums2
            if (nums1Index >= 0 && nums1[nums1Index] > nums2[nums2Index]) {
                // Place the element of nums1 in the correct position of the merged array
                nums1[mergeIndex] = nums1[nums1Index];
                nums1Index--; // Move the pointer of nums1 to the left
            } else {
                // Place the element of nums2 in the correct position of the merged array
                nums1[mergeIndex] = nums2[nums2Index];
                nums2Index--; // Move the pointer of nums2 to the left
            }
            mergeIndex--; // Move the pointer of the merged array to the left
        }
        // No need to copy the remaining elements of nums1, if any, since they are already in place.
    }
};

```

Typescript Solution

```typescript
// Merges two sorted arrays nums1 and nums2 into nums1 in sorted order.
// The first m elements of nums1 contain the initial sorted elements
// and nums1 has enough space to hold the additional elements from nums2.
// The arguments n and m represent the number of elements in nums2 and nums1, respectively.
function merge(nums1: number[], countNums1: number, nums2: number[], countNums2: number): void {
    // Initialize pointers:
    // lastNums1Index - Pointer for the last element in nums1's initial sorted part.
    // lastNums2Index - Pointer for the last element in nums2.
    // mergedIndex - Pointer for the last position in nums1 where we will place the merged element.
    let lastNums1Index: number = countNums1 - 1;
    let lastNums2Index: number = countNums2 - 1;
    let mergedIndex: number = countNums1 + countNums2 - 1;

    // Merge in reverse order to avoid overwriting elements in nums1 that have not been checked yet.
    while (lastNums2Index >= 0) {
        // Check and compare elements from nums1 and nums2.
        // Place the larger element in the correct sorted position at the end of nums1.
        nums1[mergedIndex--] = lastNums1Index >= 0 && nums1[lastNums1Index] > nums2[lastNums2Index]
            ? nums1[lastNums1Index--] // Use the element from nums1 and decrement the pointer.
            : nums2[lastNums2Index--]; // Use the element from nums2 and decrement the pointer.
    }
}

```

Time and Space Complexity

The given code has a time complexity of `O(m + n)` since it involves iterating over the elements of `nums1` and `nums2` in reverse order, where `m` and `n` are the lengths of the respective arrays. It's a single pass through the combined size of both arrays, hence the addition of `m` and `n`.

The space complexity of the code is `O(1)`, as it only uses a constant amount of extra space. The merging is done in place and does not require any additional data structures that scale with the input size.