Project Write-Up: Merge Sorted Array

Author(s): Shalom

Date: 12/18/2024

# Table of Contents

1. Executive Summary  
2. Introduction  
3. System Design and Architecture  
4. Implementation Details  
5. Testing and Validation  
6. Usage Instructions  
7. Challenges and Solutions  
8. Future Improvements  
9. Conclusion  
10. Appendix (if needed)

# 3. Executive Summary

This project addresses the problem of merging two sorted arrays, `nums1` and `nums2`, into a single sorted array stored in `nums1`. The solution efficiently merges the arrays in-place, with a time complexity of O(m + n), where `m` and `n` are the sizes of `nums1` and `nums2`, respectively.

# 4. Introduction

Objective: The goal is to merge two sorted integer arrays, `nums1` and `nums2`, into one sorted array stored in `nums1`. The function modifies `nums1` in-place, leveraging the extra space at the end of `nums1` (set to 0) to accommodate the merged elements.

Background: Merging sorted arrays is a fundamental operation in computer science, often used as a step in merge sort and other algorithms.

Scope: The solution focuses on solving the merge problem efficiently, with a time complexity of O(m + n), where `m` and `n` are the lengths of `nums1` and `nums2`.

# 5. System Design and Architecture

High-Level Design: The solution uses a two-pointer approach to merge the arrays in-place, starting from the end of both arrays to avoid overwriting elements in `nums1`.

Component Overview:  
- `nums1` (length `m + n`): The array that stores the final merged result.  
- `nums2` (length `n`): The array that needs to be merged with `nums1`.

Design Decisions:  
- The function begins the merge process from the back of `nums1` and `nums2` to ensure that no elements in `nums1` are overwritten.  
- The two-pointer technique ensures that the merging process is completed in linear time (O(m + n)).

# 6. Implementation Details

Code Structure:   
The method `merge(int[] nums1, int m, int[] nums2, int n)` merges `nums2` into `nums1` in-place.

Algorithm:   
- Step 1: Initialize two pointers, `p1` for `nums1` and `p2` for `nums2`, starting from the end of the valid elements.  
- Step 2: Iterate backward from the end of `nums1` and compare elements from `nums1` and `nums2`. Place the larger element in the correct position in `nums1`.  
- Step 3: Continue this process until all elements from `nums2` are placed into `nums1`.  
- Step 4: If elements remain in `nums2`, copy them directly into `nums1`.

Code Snippet:

class Solution {  
 public void merge(int[] nums1, int m, int[] nums2, int n) {  
 int p1 = m - 1; // Pointer for nums1  
 int p2 = n - 1; // Pointer for nums2  
  
 // Start merging from the end of nums1  
 for (int i = nums1.length - 1; i >= 0; i--) {  
 if (p2 < 0) {  
 break; // No more elements in nums2 to merge  
 } else if (p1 < 0) {  
 nums1[i] = nums2[p2]; // Copy element from nums2  
 p2--;  
 } else if (nums2[p2] > nums1[p1]) {  
 nums1[i] = nums2[p2]; // Copy larger element from nums2  
 p2--;  
 } else {  
 nums1[i] = nums1[p1]; // Copy element from nums1  
 p1--;  
 }  
 }  
 }  
}

# 7. Testing and Validation

Testing Approach:  
Various test cases are designed to test different scenarios, including:  
- Both `nums1` and `nums2` contain elements.  
- One array is empty while the other contains elements.  
- Edge cases such as `nums1` or `nums2` being empty, or arrays with zero elements.

Results: The function passed all test cases, correctly merging arrays and maintaining the sorted order.

Known Issues: There are no known issues, as the solution handles all edge cases properly.

# 8. Usage Instructions

Installation: This function can be added to any Java project. No installation is required.

Running the Project: To use the merge function, call it with two sorted arrays and their sizes:

int[] nums1 = {1, 2, 3, 0, 0, 0};  
int m = 3;  
int[] nums2 = {2, 5, 6};  
int n = 3;  
Solution solution = new Solution();  
solution.merge(nums1, m, nums2, n);  
System.out.println(Arrays.toString(nums1)); // Output: [1, 2, 2, 3, 5, 6]

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

# 9. Challenges and Solutions

Challenges:  
Merging arrays in-place without using extra space was a key challenge. The approach of starting from the back of `nums1` allowed for a clean in-place merge without overwriting elements.

Solutions:  
The use of the two-pointer technique efficiently merged both arrays in linear time, avoiding unnecessary space usage.

# 10. Future Improvements

Optimizations: The current solution already runs in O(m + n) time, which is optimal for this problem. Future improvements could focus on handling larger datasets or incorporating additional features like removing duplicates, but for the given problem, the approach is efficient.

Known Issues: No further issues have been identified in this solution.

# 11. Conclusion

The solution effectively merges two sorted arrays in-place with O(m + n) time complexity, meeting the problem's constraints and objectives. This approach ensures that the final merged array is correctly sorted and stored within `nums1`.

# 12. Appendix

References: LeetCode Problem 88: Merge Sorted Array (https://leetcode.com/problems/merge-sorted-array/)