88. Merge Sorted Arrays

Feb. 7, 2019 | 96.5K views

Given two sorted integer arrays *nums1* and *nums2*, merge *nums2* into *nums1* as one sorted array.

Note:

- The number of elements initialized in *nums1* and *nums2* are *m* and *n* respectively.
- You may assume that *nums1* has enough space (size that is **equal** to *m* + *n*) to hold additional elements from *nums2*.

Example:

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

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

Constraints:

- `-10^9 <= nums1[i], nums2[i] <= 10^9`
- `nums1.length == m + n`
- `nums2.length == n`

Solution

Approach 1 : Merge and sort

Intuition

The naive approach would be to merge both lists into one and then to sort. It's a one line solution (2 lines in Java) with a pretty bad time complexity $\mathcal{O}((n + m)\log(n + m))$ because here one doesn't profit from the fact that both arrays are already sorted.

Implementation

Java | Python | Copy

```python
class Solution(object):
    def merge(self, nums1, m, nums2, n):
        """
        :type nums1: List[int]
        :type m: int
        :type nums2: List[int]
        :type n: int
        :rtype: void Do not return anything, modify nums1 in-place instead.
        """
        nums1[:] = sorted(nums1[:m] + nums2)
```

- Time complexity : $\mathcal{O}((n + m)\log(n + m))$.
- Space complexity : $\mathcal{O}(1)$.

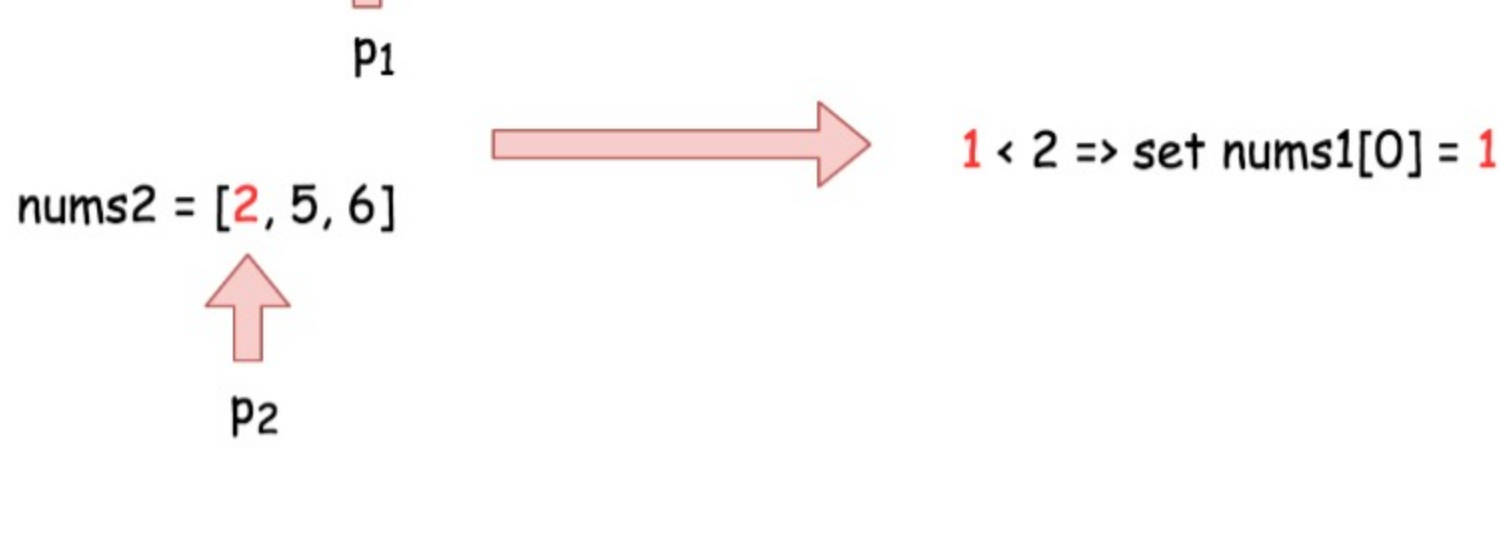
Approach 2 : Two pointers / Start from the beginning

Intuition

Typically, one could achieve $\mathcal{O}(n + m)$ time complexity in a sorted array(s) with the help of *two pointers approach*.

The straightforward implementation would be to set get pointer `p1` in the beginning of `nums1`, `p2` in the beginning of `nums2`, and push the smallest value in the output array at each step.

Since `nums1` is an array used for output, one has to keep first `m` elements of `nums1` somewhere aside, that means $\mathcal{O}(m)$ space complexity for this approach.



Implementation

Java | Python | Copy

```python
class Solution(object):
    def merge(self, nums1, m, nums2, n):
        """
        :type nums1: List[int]
        :type m: int
        :type nums2: List[int]
        :type n: int
        :rtype: void Do not return anything, modify nums1 in-place instead.
        """
        # Make a copy of nums1.
        nums1_copy = nums1[:m]
        nums1[:] = []

        # Two get pointers for nums1_copy and nums2.
        p1 = 0
        p2 = 0

        # Compare elements from nums1_copy and nums2
        # and add the smallest one into nums1.
        while p1 < m and p2 < n:
            if nums1_copy[p1] < nums2[p2]:
                nums1.append(nums1_copy[p1])
                p1 += 1
            else:
                nums1.append(nums2[p2])
                p2 += 1

```

Complexity Analysis

- Time complexity : $\mathcal{O}(n + m)$.
- Space complexity : $\mathcal{O}(m)$.

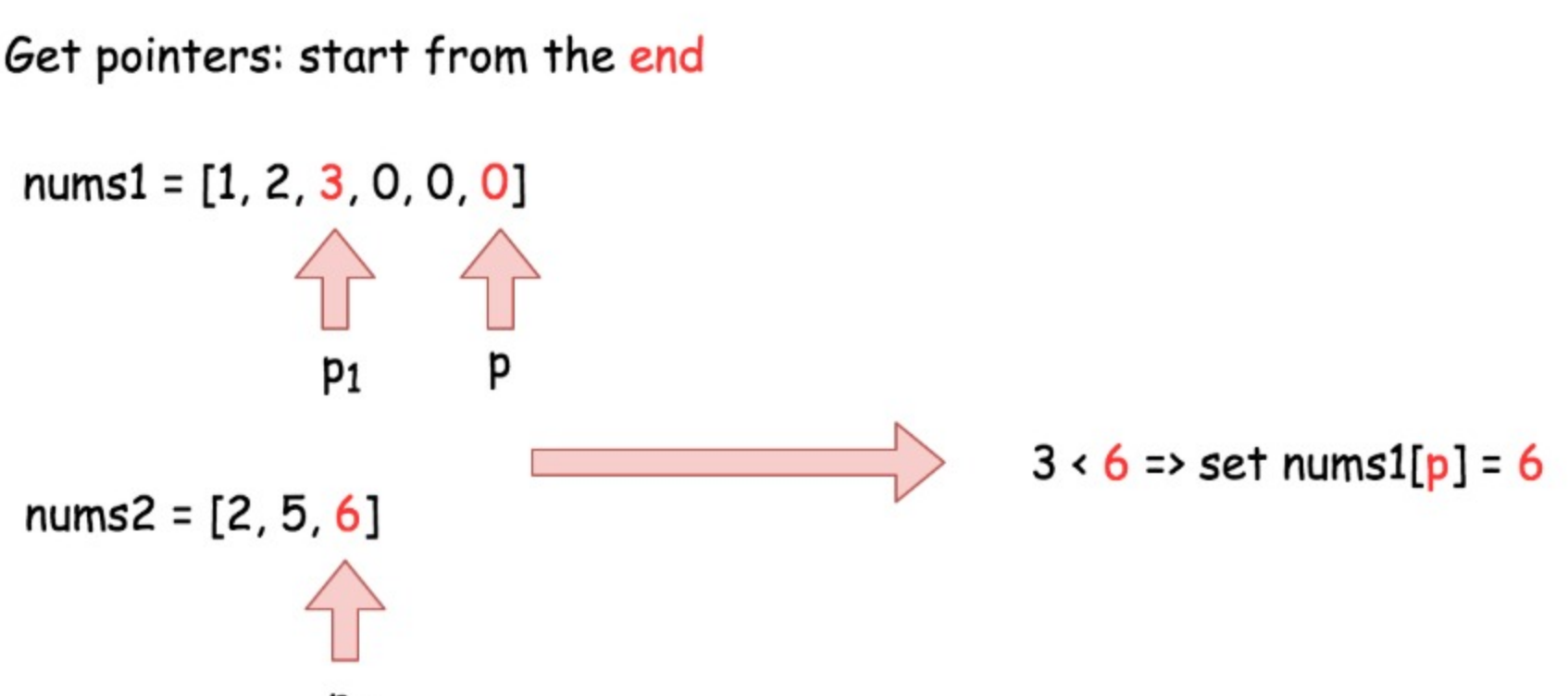
Approach 3 : Two pointers / Start from the end

Intuition

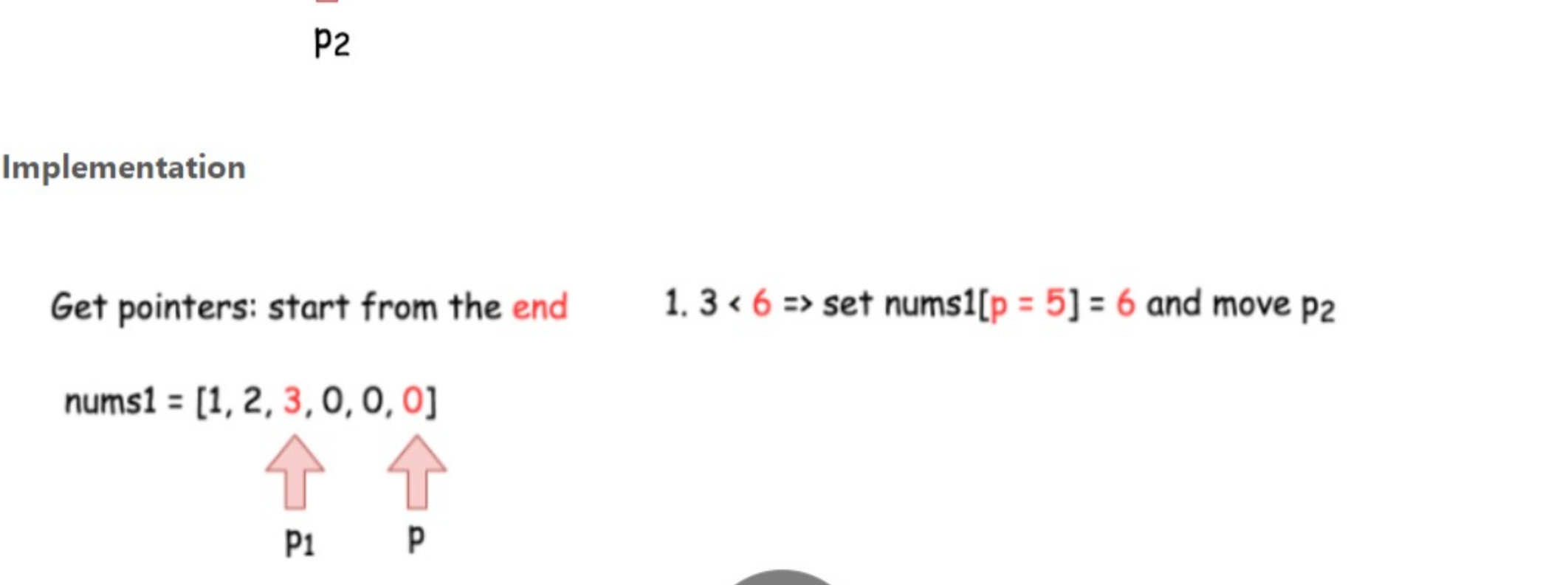
Approach 2 already demonstrates the best possible time complexity $\mathcal{O}(n + m)$ but still uses an additional space. This is because one has to keep somewhere the elements of array `nums1` while overwriting it starting from the beginning.

> What if we start to overwrite `nums1` from the end, where there is no information yet? Then no additional space is needed.

The set pointer `p` here is used to track the position of an added element.



Implementation



Java | Python | Copy

```python
1  class Solution(object):
2      def merge(self, nums1, m, nums2, n):
3          """
4          :type nums1: List[int]
5          :type m: int
6          :type nums2: List[int]
7          :type n: int
8          :rtype: void Do not return anything, modify nums1 in-place instead.
9          """
10         # two get pointers for nums1 and nums2
11         p1 = m - 1
12         p2 = n - 1
13         # set pointer for nums1
14         p = m + n - 1
15 
16         # while there are still elements to compare
17         while p1 >= 0 and p2 >= 0:
18             if nums1[p1] < nums2[p2]:
19                 nums1[p] = nums2[p2]
20                 p2 -= 1
21             else:
22                 nums1[p] =  nums1[p1]
23                 p1 -= 1
24             p -= 1
25 
26         # add missing elements from nums2
27         nums1[:p2 + 1] = nums2[:p2 + 1]
```

Complexity Analysis

- Time complexity : $\mathcal{O}(n + m)$.
- Space complexity : $\mathcal{O}(1)$.