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

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

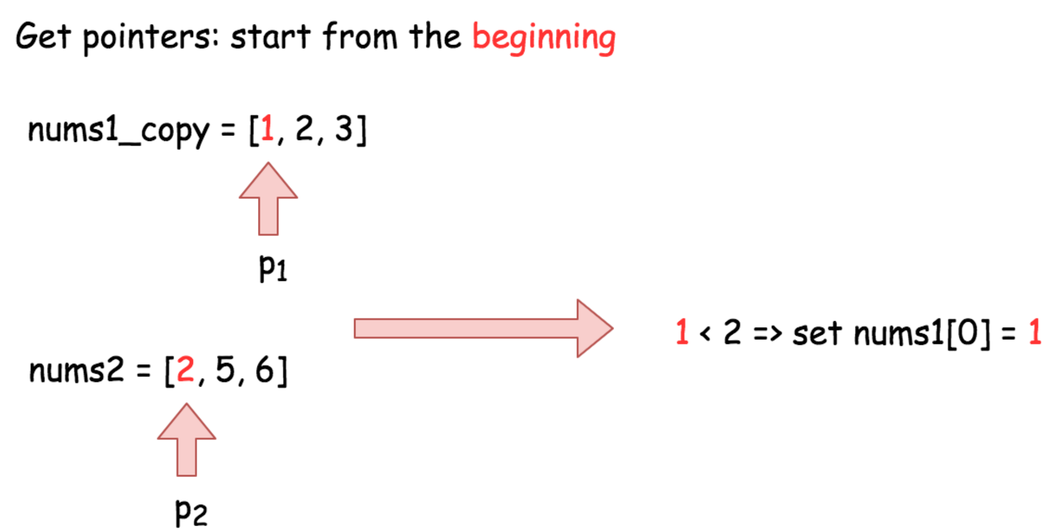
Approach 2 : Two pointers / Start from the beginning

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

Complexity Analysis

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

- Space complexity : $O(m)$.

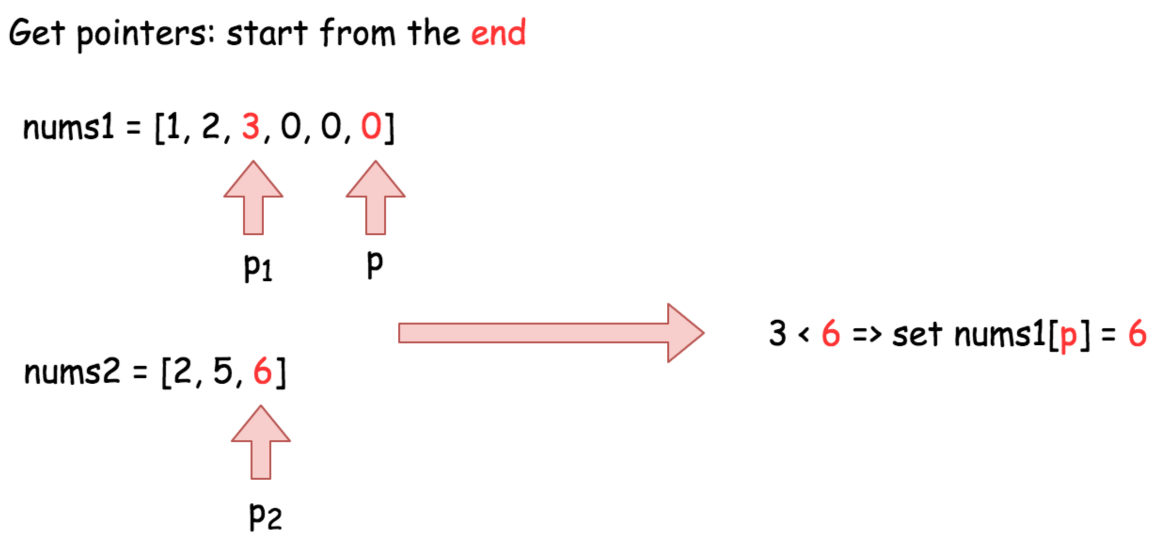
Approach 3 : Two pointers / Start from the end

Intuition

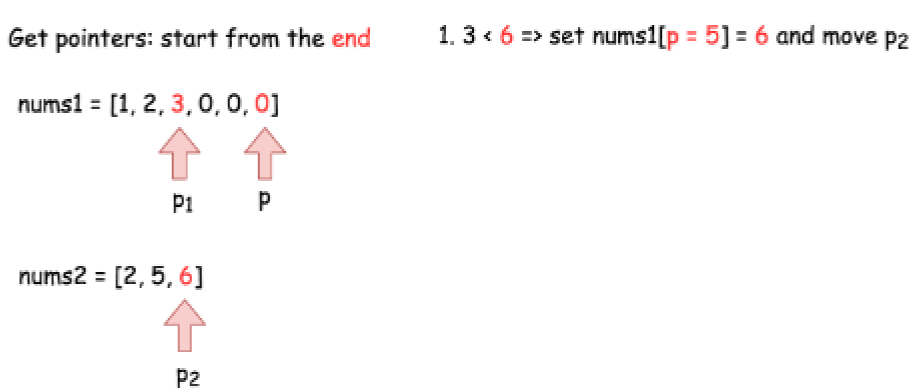
Approach 2 already demonstrates the best possible time complexity $O(n + m)$ but still uses an additional space. This is because one has to keep somewhere the elements of array `nums1` while overwriting it starting from the beginning.

> 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



Complexity Analysis

- Time complexity : $O(n + m)$.
- Space complexity : $O(1)$.