Previous Next

80. Remove Duplicates from Sorted Array II

Oct. 31, 2019 | 7.6K views

Average Rating: 4.86 (14 votes)

Given a sorted array *nums*, remove the duplicates **in-place** such that duplicates appeared at most *twice* and return the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array in-place** with O(1) extra memory.

Example 1:

```
Given nums = [1,1,1,2,2,3],

Your function should return length = 5, with the first five elements of nums
 being 1, 1, 2, 2 and 3 respectively.

It doesn't matter what you leave beyond the returned length.
```

Example 2:

```
Given nums = [0,0,1,1,1,1,2,3,3],

Your function should return length = 7, with the first seven elements of nums
 being modified to 0, 0, 1, 1, 2, 3 and 3 respectively.

It doesn't matter what values are set beyond the returned length.
```

Clarification:

Confused why the returned value is an integer but your answer is an array?

Note that the input array is passed in by **reference**, which means modification to the input array will be known to the caller as well.

Internally you can think of this:

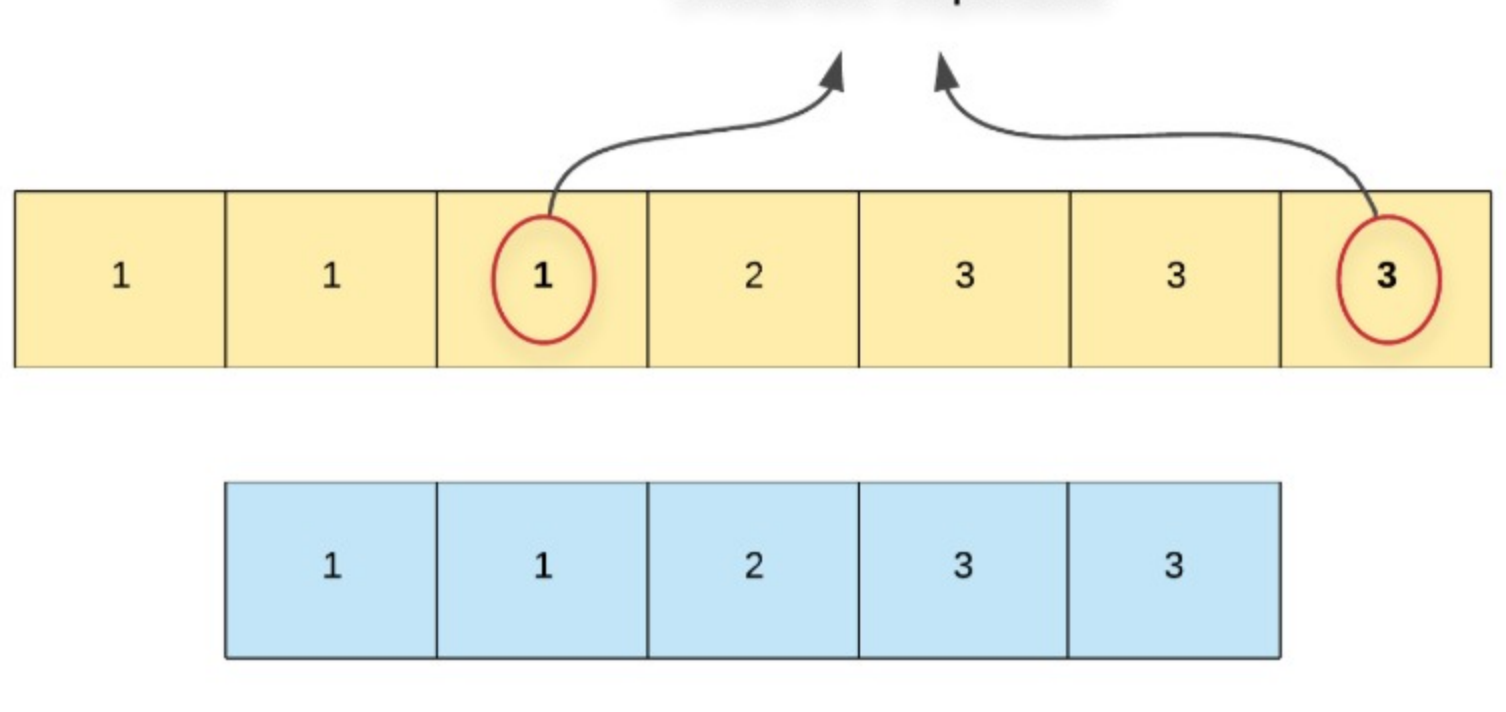
```
// nums is passed in by reference. (i.e., without making a copy)
int len = removeDuplicates(nums);

// any modification to nums in your function would be known by the caller.
// using the length returned by your function, it prints the first len elements.
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

Solution

Approach 1: Popping Unwanted Duplicates

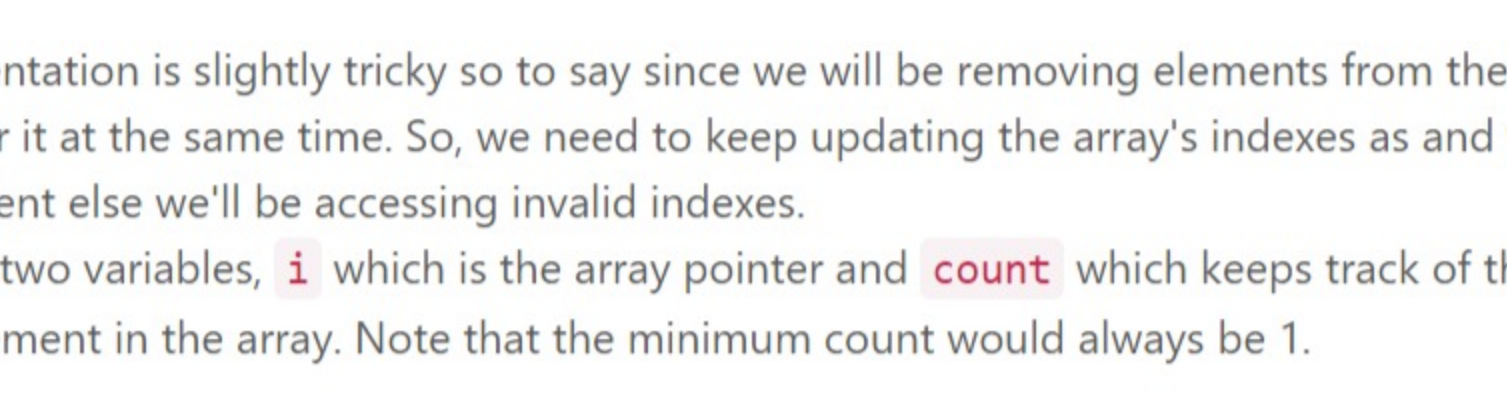
Intuition

The input array is already sorted and hence, all the duplicates appear next to each other. The problem statement mentions that we are not allowed to use any additional space and we have to modify the array in-place. The easiest approach for in-place modifications would be to get rid of all the unwanted duplicates. For every number in the array, if we detect `> 2` duplicates, we simply remove them from the list of elements and we do this for all the elements in the array.

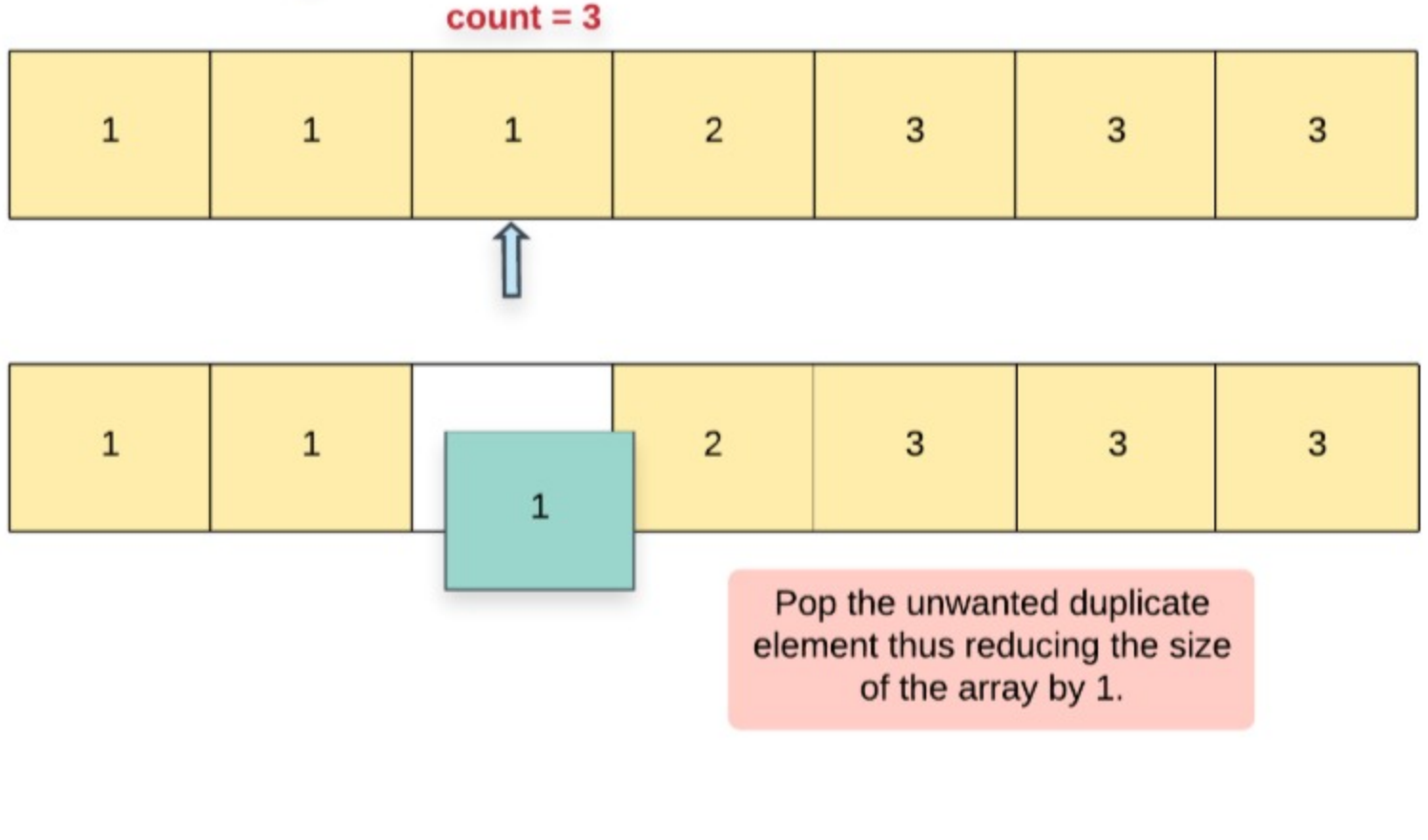


Algorithm

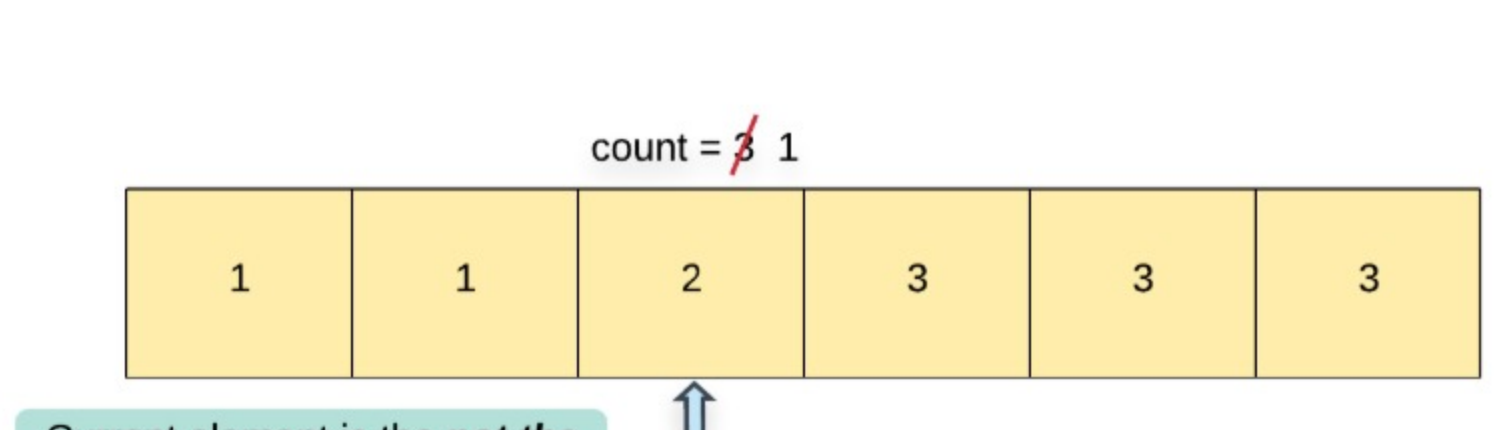
1. The implementation is slightly tricky so to say since we will be removing elements from the array and iterating over it at the same time. So, we need to keep updating the array's indexes as and when we pop an element else we'll be accessing invalid indexes.
2. Say we have two variables, `i` which is the array pointer and `count` which keeps track of the count of a particular element in the array. Note that the minimum count would always be 1.



3. We start with index `1` and process one element at a time in the array.
4. If we find that the current element is the *same* as the previous element i.e. `nums[i] == nums[i - 1]`, then we increment the `count`. If the value of `count > 2`, then we have encountered an unwanted duplicate element and we can remove it from the array. Since we know the index of this element, we can use the `del` or `pop` or `remove` operation (or whatever corresponding operation is supported in your language of choice) to delete the element at index `i` from the array. Since we popped an element, we decrement the index by 1 as well.



5. If we encounter that the current element is *not* the same as the previous element i.e. `nums[i] != nums[i - 1]`, then it means we have a new element at hand and so accordingly, we update `count = 1`.



6. Since we are removing all the unwanted duplicates from the original array, the final array that remains after process all the elements will only contain the valid elements and hence we simply return the length of this array.

Java Python Copy

```python
class Solution(object):
    def removeDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """

        # Initialize the counter and the array index.
        i, count = 1, 1

        # Start from the second element of the array and process
        # elements one by one.
        while i < len(nums):

            # If the current element is a duplicate,
            # increment the count.
            if nums[i] == nums[i - 1]:
                count += 1

                # If the count is more than 2, this is an
                # unwanted duplicate element and hence we
                # remove it from the array.
                if count > 2:
                    nums.pop(i)

                    # Note that we have to decrement the
                    # array index value to keep it consistent
```

Complexity Analysis

- Time Complexity: Let's see what the costly operations in our array are:
  - We have to iterate over all the elements in the array. Suppose that the original array contains `N` elements, the time taken here would be $O(N)$.
  - Next, for every unwanted duplicate element, we will have to perform a delete operation and deletions in arrays are also $O(N)$.
  - The worst case would be when all the elements in the array are the same. In that case, we would be performing $N - 2$ deletions thus giving us $O(N^2)$ complexity for deletions
  - Overall complexity = $O(N) + O(N^2) \equiv O(N^2)$.
- Space Complexity: $O(1)$ since we are modifying the array in-place.

Approach 2: Overwriting unwanted duplicates

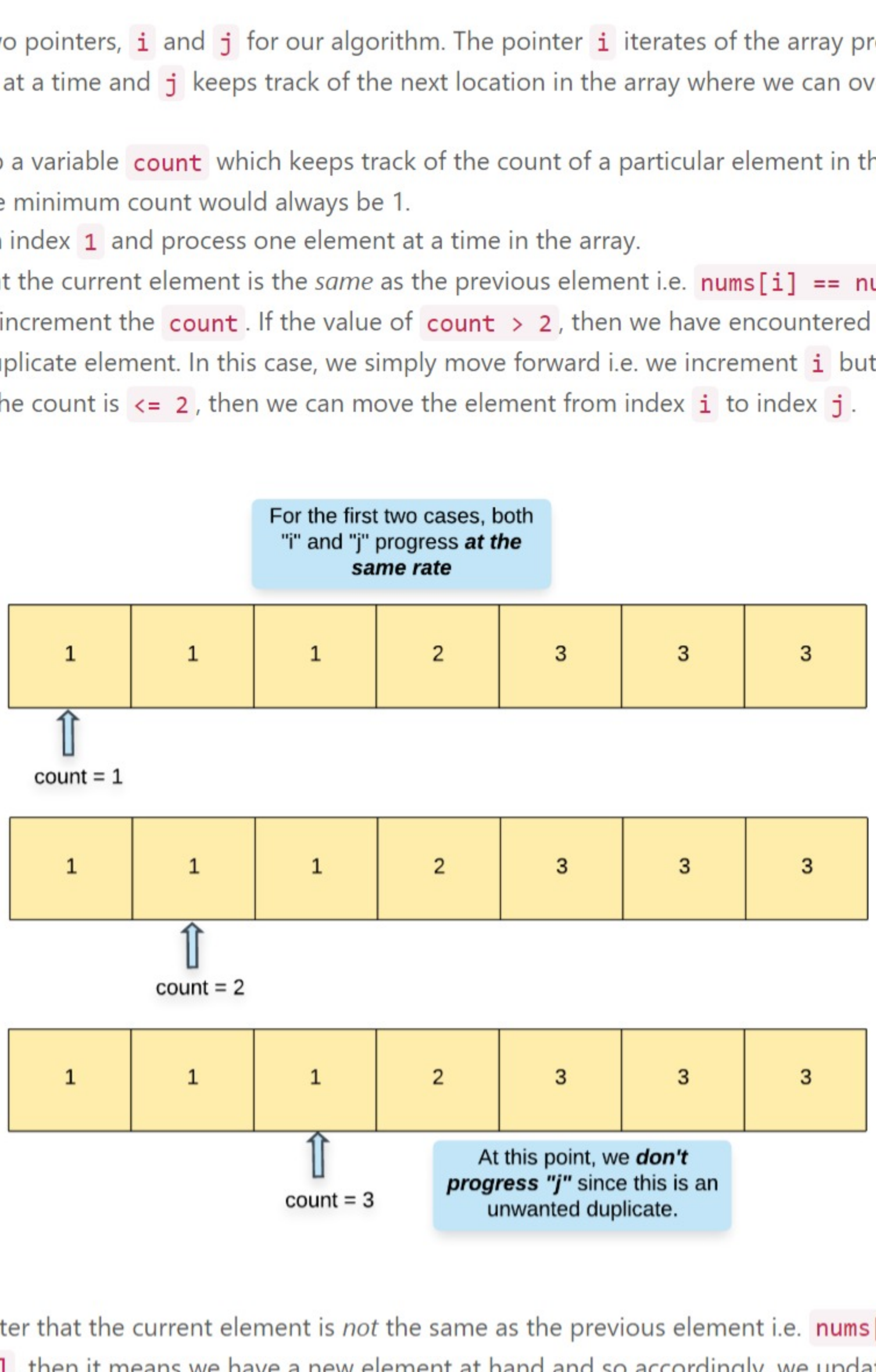
Intuition

The second approach is really inspired by the fact that the problem statement asks us to return the *new length of the array* from the function. If all we had to do was *remove elements*, the function would not really ask us to return the updated length. However, in our scenario, this is really an indication that we don't need to actually remove elements from the array. Instead, we can do something better and simply overwrite the duplicate elements that are unwanted.

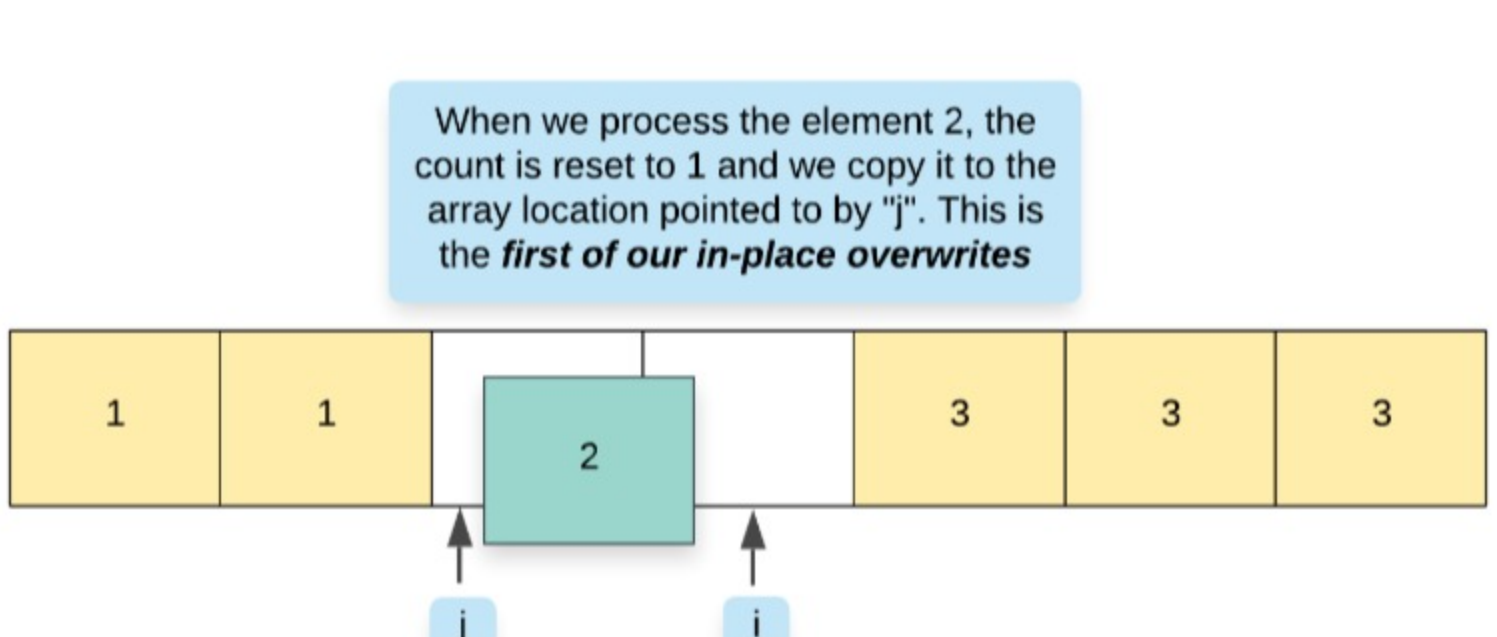
> We won't be able to achieve this using a single pointer. We will be using a two-pointer approach where one pointer iterates over the original set of elements and another one that keeps track of the next "empty" location in the array or the next location that can be overwritten in the array.

Algorithm

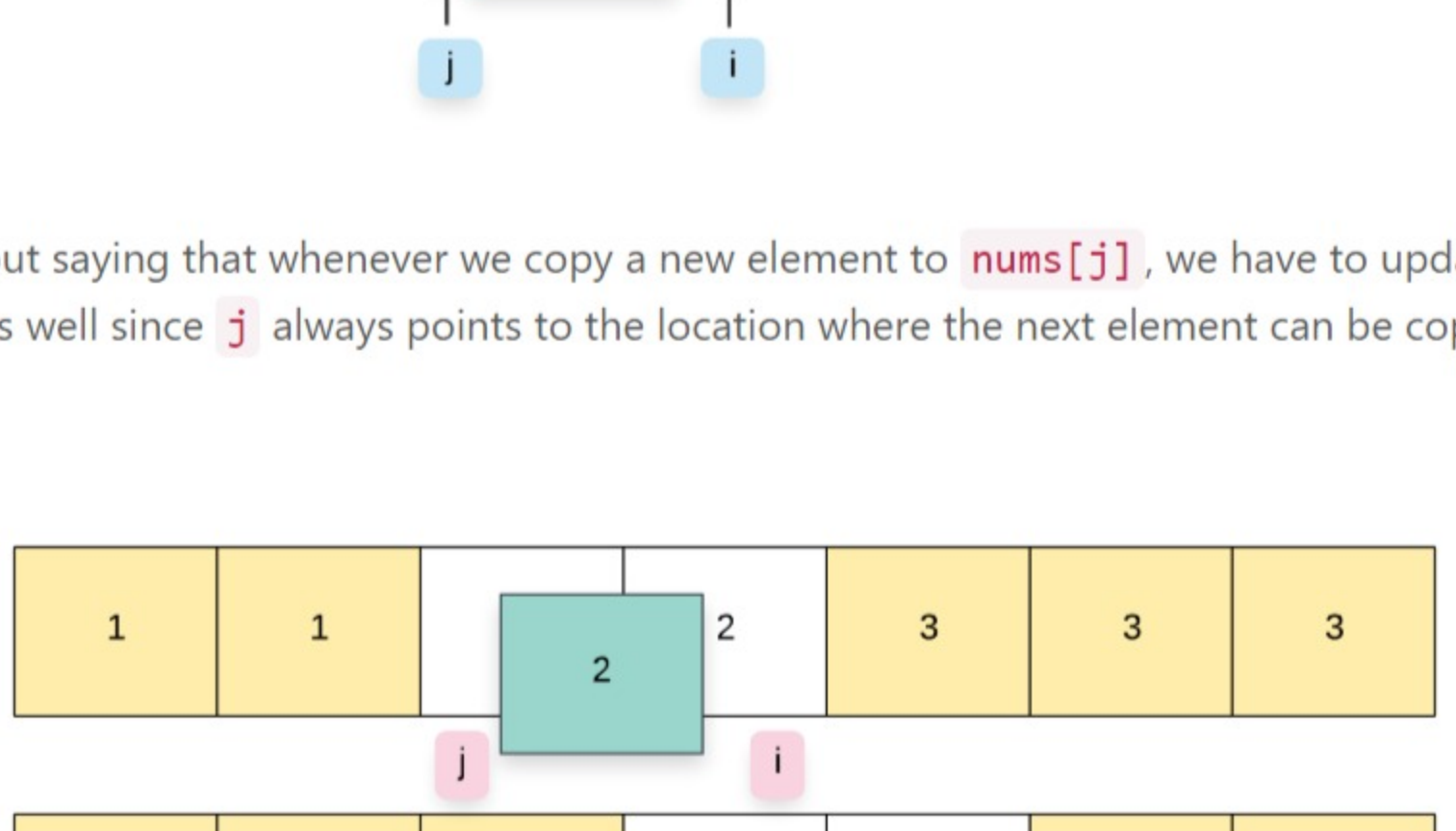
1. We define two pointers, `i` and `j` for our algorithm. The pointer `i` iterates of the array processing one element at a time and `j` keeps track of the next location in the array where we can overwrite an element.
2. We also keep a variable `count` which keeps track of the count of a particular element in the array. Note that the minimum count would always be 1.
3. We start with index `1` and process one element at a time in the array.
4. If we find that the current element is the *same* as the previous element i.e. `nums[i] == nums[i - 1]`, then we increment the `count`. If the value of `count > 2`, then we have encountered an unwanted duplicate element. In this case, we simply move forward i.e. we increment `i` but not `j`.
5. However, if the count is `<= 2`, then we can move the element from index `i` to index `j`.



6. If we encounter that the current element is *not* the same as the previous element i.e. `nums[i] != nums[i - 1]`, then it means we have a new element at hand and so accordingly, we update `count = 1` and also move this element to index `j`.



7. It goes without saying that whenever we copy a new element to `nums[j]`, we have to update the value of `j` as well since `j` always points to the location where the next element can be copied to in the array.



Java Python Copy

```python
class Solution(object):
    def removeDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """

        # Initialize the counter and the second pointer.
        j, count = 1, 1

        # Start from the second element of the array and process
        # elements one by one.
        for i in range(1, len(nums)):

            # If the current element is a duplicate,
            # increment the count.
            if nums[i] == nums[i - 1]:
                count += 1
            else:
                # Reset the count since we encountered a different element
                # than the previous one
                count = 1

            # For a count <= 2, we copy the element over thus
            # overwriting the element at index "j" in the array
            if count <= 2:
                nums[j] = nums[i]
```

Complexity Analysis

- Time Complexity: $O(N)$ since we process each element exactly once.
- Space Complexity: $O(1)$.

Rate this article:

Previous Next

Comments: 9

Sort By

Type comment here... (Markdown is supported)

Preview Post

soumyajitchatterjee73 15 November 1, 2019 9:10 AM

using two-pointer: O(n) solution

```
```private static int getRemoveDuplicates(int [] a) {
    int m = 0, i=1, count=1, lastVisit = a[0];
    a[m] = a[0];
```

Read More

2 Share Reply

ngoc_lam 42 March 23, 2020 4:59 PM

Clean code :(.

```
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        length = 0
```

Read More

1 Share Reply

SHOW 2 REPLIES

shawn_li 9 November 7, 2019 12:55 PM

```
def removeDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: int
```

Read More

1 Share Reply

SHOW 1 REPLY

stevenxtw 0 May 21, 2020 1:53 PM

Why cannot use two pointer like this?

class Solution:
def removeDuplicates(self, nums: List[int]) -> int:
if len(nums) <= 2:

Read More

0 Share Reply

takenpilot 2 May 10, 2020 3:00 AM

Super fast JavaScript solution:

```
var removeDuplicates = function(nums) {
  if (nums.length <= 2) {
    return nums.length;
```

Read More

0 Share Reply

ysr55 3 April 5, 2020 12:12 PM

@sachinmalhotra1993 Superb Second solution , clean and simple .

0 Share Reply

hxuanhung 158 March 25, 2020 1:57 PM

```
class Solution:
    def removeDuplicates(self, A: List[int]) -> int:
        n = len(A)
        if n < 3:
```

Read More

0 Share Reply

migeater 126 December 2, 2019 8:16 AM

implement python `itertools.groupby`

```
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        # only sorted
```

Read More

0 Share Reply

rahulkun 446 November 11, 2019 10:41 AM

desired complexity is O(logn). Both solutions are inefficient since you could have just iterated over same elements in sorted array to see which one has count = 1.

-1 Share Reply

SHOW 2 REPLIES
```