remove_duplicated_from_sorted_array

May 5, 2025

1 Remove Duplicated From Sorted Array

1.1 Problem Definition

Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return the number of unique elements in nums.

Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:

Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums. Return k.

1.1.1 Example Cases

Example 1:

Input: nums = [1, 1, 2] Output: 2, nums = [1,2,_] Explanation: Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively. It does not matter what you leave beyond the returned k (hence they are underscores).

Example 2:

Input: nums = [0,0,1,1,1,2,2,3,3,4] Output: 5, nums = [0,1,2,3,4,,,,,_] Explanation: Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively. It does not matter what you leave beyond the returned k (hence they are underscores).

1.1.2 Constraints

Constraints:

- 1 <= nums.length <= 3 * 10^4
- -100 <= nums[i] <= 100
- nums is sorted in non-decreasing order.

1.2 Example test cases

[122]:
```
def check(nums, expectedNums, removeDuplicates):
    k, _ = removeDuplicates(nums)
    assert k == len(expectedNums), f"Expected length {len(expectedNums)}, got␣
↪{k}"
    for i in range(k):
        assert nums[i] == expectedNums[i], f"At index {i}, expected␣
↪{expectedNums[i]}, got {nums[i]}"
    return True
```

[131]:
```
def test_cases_inplace_check():
    assert check([1,1,2], [1,2], removeDuplicates)
    assert check([0,0,1,1,1,2,2,3,3,4], [0,1,2,3,4], removeDuplicates)
    print("All test cases passed!")
```

[132]:
```
def test_cases():
    count, nums = removeDuplicates([1,1,2])
    assert (count, nums) == (2, [1,2,''])
    count, nums = removeDuplicates([0,0,1,1,1,2,2,3,3,4])
    assert (count, nums) == (5, [0,1,2,3,4,'','','','',''])
    print("All test cases passed!")
```

1.3 Solutions

Time complexity - $O(n^2)$, State complexity - $O(n)$

The first solution that came to my mind is to create a new list and to count the number of non-unique elements throughout iteration. If there is a unique element, add to the new list. Return the length of the new list and the combination of the new list (unique elements) and of a list containing number of elements corresponding to the number of non-unique elements.

Time complexity is $O(n^2)$ because checking if the number is in the list or not takes another n iterations.

[133]:
```
def removeDuplicates(nums):
    new_nums = []
    count = 0
    for num in nums:
        if num in new_nums:
            count += 1
        else:
            new_nums.append(num)
    return len(new_nums), new_nums + count*['']
```

[134]:
```
removeDuplicates([1,2,2])
```

[134]:
```
(2, [1, 2, ''])
```

[135]:
```
test_cases()
```

```
All test cases passed!
```

Improvement Time complexity - $O(n)$, State complexity - $O(n)$

Replacing the list with a set would make the look up O(1). Thus, the overall time complexity will become O(n). However, for smaller sets it may be slower because of the conversion from set to lists. Another alternative is to have another a separate list and set instead.

[136]:
```
# With conversion
def removeDuplicates(nums):
    new_nums = set()
    count = 0
    for num in nums:
        if num in new_nums:
            count += 1
        else:
            new_nums.add(num)
    return len(new_nums), list(new_nums) + count*['']
```

[137]:
```
# With a separate set
def removeDuplicates(nums):
    seen = set()
    new_nums = []
    count = 0
    for num in nums:
        if num in seen:
            count += 1
        else:
            seen.add(num)
            new_nums.append(num)

    # For more readability and extend works faster than summing because it␣
 ↪doens't create a new list but extends the existing one
    new_nums.extend([''] * count)
    return len(new_nums) - count, new_nums
```

[138]:
```
removeDuplicates([1,2,2])
```

[138]:
```
(2, [1, 2, ''])
```

[139]:
```
test_cases()
```

```
All test cases passed!
```

Time complexity - $O(n)$, State complexity - $O(1)$

The state complexity can also be reduced to $O(1)$ if the list elements are modified during the for loop. An important notice regarding this is that it's safe only if we are overwriting elements.

Changing the structure or the length of the list while doing a for loop is a problem.

[140]:
```
# With a separate set
def removeDuplicates(nums):
    k = 0
    for num in nums:

        # For the first iteration the next the second statement of the or␣
 ↪statement will not be called since the first is True
        if k==0 or num != nums[k-1]:
            nums[k] = num
            k+=1
    return k, nums
```

[141]:
```
removeDuplicates([1,1,2])
```

[141]:
```
(2, [1, 2, 2])
```

[142]:
```
test_cases_inplace_check()
```

```
All test cases passed!
```