**27. Remove Element**

**Question**

Given an array and a value, remove all instances of that value in place and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

**Example:**  
Given input array *nums* = [3,2,2,3], *val* = 3

Your function should return length = 2, with the first two elements of *nums* being 2.

1. Try two pointers.
2. Did you use the property of "the order of elements can be changed"?
3. What happens when the elements to remove are rare?

**Quick Navigation**

* [Summary](https://leetcode.com/articles/remove-element/#summary)
* [Hints](https://leetcode.com/articles/remove-element/#hints)
* [Solution](https://leetcode.com/articles/remove-element/#solution)
  + [Approach #1 (Two Pointers) [Accepted]](https://leetcode.com/articles/remove-element/#approach-1-two-pointers-accepted)
  + [Approach #2 (Two Pointers - when elements to remove are rare) [Accepted]](https://leetcode.com/articles/remove-element/#approach-2-two-pointers-when-elements-to-remove-are-rare-accepted)

**Summary**

This is a pretty easy problem, but one may get confused by the term "in-place" and thought it is impossible to remove an element from the array without making a copy of the array.

**Hints**

1. Try two pointers.
2. Did you use the property of "the order of elements can be changed"?
3. What happens when the elements to remove are rare?

**Solution**

**Approach #1 (Two Pointers) [Accepted]**

**Intuition**

Since question asked us to remove all elements of the given value in-place, we have to handle it with O(1)O(1)O(1) extra space. How to solve it? We can keep two pointers iii and jjj, where iii is the slow-runner while jjj is the fast-runner.

**Algorithm**

When nums[j]nums[j]nums[j] equals to the given value, skip this element by incrementing jjj. As long as nums[j]≠valnums[j] \neq valnums[j]≠val, we copy nums[j]nums[j]nums[j] to nums[i]nums[i]nums[i] and increment both indexes at the same time. Repeat the process until jjj reaches the end of the array and the new length is iii.

This solution is very similar to the solution to [Remove Duplicates from Sorted Array](https://leetcode.com/articles/remove-duplicates-sorted-array/).

**public** **int** **removeElement(int[]** nums**,** **int** val**)** **{**

**int** i **=** 0**;**

**for** **(int** j **=** 0**;** j **<** nums**.**length**;** j**++)** **{**

**if** **(**nums**[**j**]** **!=** val**)** **{**

nums**[**i**]** **=** nums**[**j**];**

i**++;**

**}**

**}**

**return** i**;**

**}**

**Complexity analysis**

* Time complexity : O(n)O(n)O(n). Assume the array has a total of nnn elements, both iii and jjj traverse at most 2n2n2n steps.
* Space complexity : O(1)O(1)O(1).

**Approach #2 (Two Pointers - when elements to remove are rare) [Accepted]**

**Intuition**

Now consider cases where the array contains few elements to remove. For example, nums=[1,2,3,5,4],val=4nums = [1,2,3,5,4], val = 4nums=[1,2,3,5,4],val=4. The previous algorithm will do unnecessary copy operation of the first four elements. Another example is nums=[4,1,2,3,5],val=4nums = [4,1,2,3,5], val = 4nums=[4,1,2,3,5],val=4. It seems unnecessary to move elements [1,2,3,5][1,2,3,5][1,2,3,5] one step left as the problem description mentions that the order of elements could be changed.

**Algorithm**

When we encounter nums[i]=valnums[i] = valnums[i]=val, we can swap the current element out with the last element and dispose the last one. This essentially reduces the array's size by 1.

Note that the last element that was swapped in could be the value you want to remove itself. But don't worry, in the next iteration we will still check this element.

**public** **int** **removeElement(int[]** nums**,** **int** val**)** **{**

**int** i **=** 0**;**

**int** n **=** nums**.**length**;**

**while** **(**i **<** n**)** **{**

**if** **(**nums**[**i**]** **==** val**)** **{**

nums**[**i**]** **=** nums**[**n **-** 1**];**

*// reduce array size by one*

n**--;**

**}** **else** **{**

i**++;**

**}**

**}**

**return** n**;**

**}**

**Complexity analysis**

* Time complexity : O(n)O(n)O(n). Both iii and nnn traverse at most nnn steps. In this approach, the number of assignment operation is equal to the number of elements to remove. So it is more efficient if elements to remove are rare.
* Space complexity : O(1)O(1)O(1).