# Question

Given an array *nums* and a value val, remove all instances of that value [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) and return the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

The order of elements can be changed. It doesn't matter what you leave beyond the new 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 a 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 = removeElement(nums, val);

// 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]);

}

**Example 1:**

**Input:** nums = [3,2,2,3], val = 3

**Output:** 2, nums = [2,2]

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

It doesn't matter what you leave beyond the returned length. For example if you return 2 with nums = [2,2,3,3] or nums = [2,2,0,0], your answer will be accepted.

**Example 2:**

**Input:** nums = [0,1,2,2,3,0,4,2], val = 2

**Output:** 5, nums = [0,1,4,0,3]

**Explanation:** Your function should return length = **5**, with the first five elements of *nums* containing **0**, **1**, **3**, **0**, and **4**. Note that the order of those five elements can be arbitrary. It doesn't matter what values are set beyond the returned length.

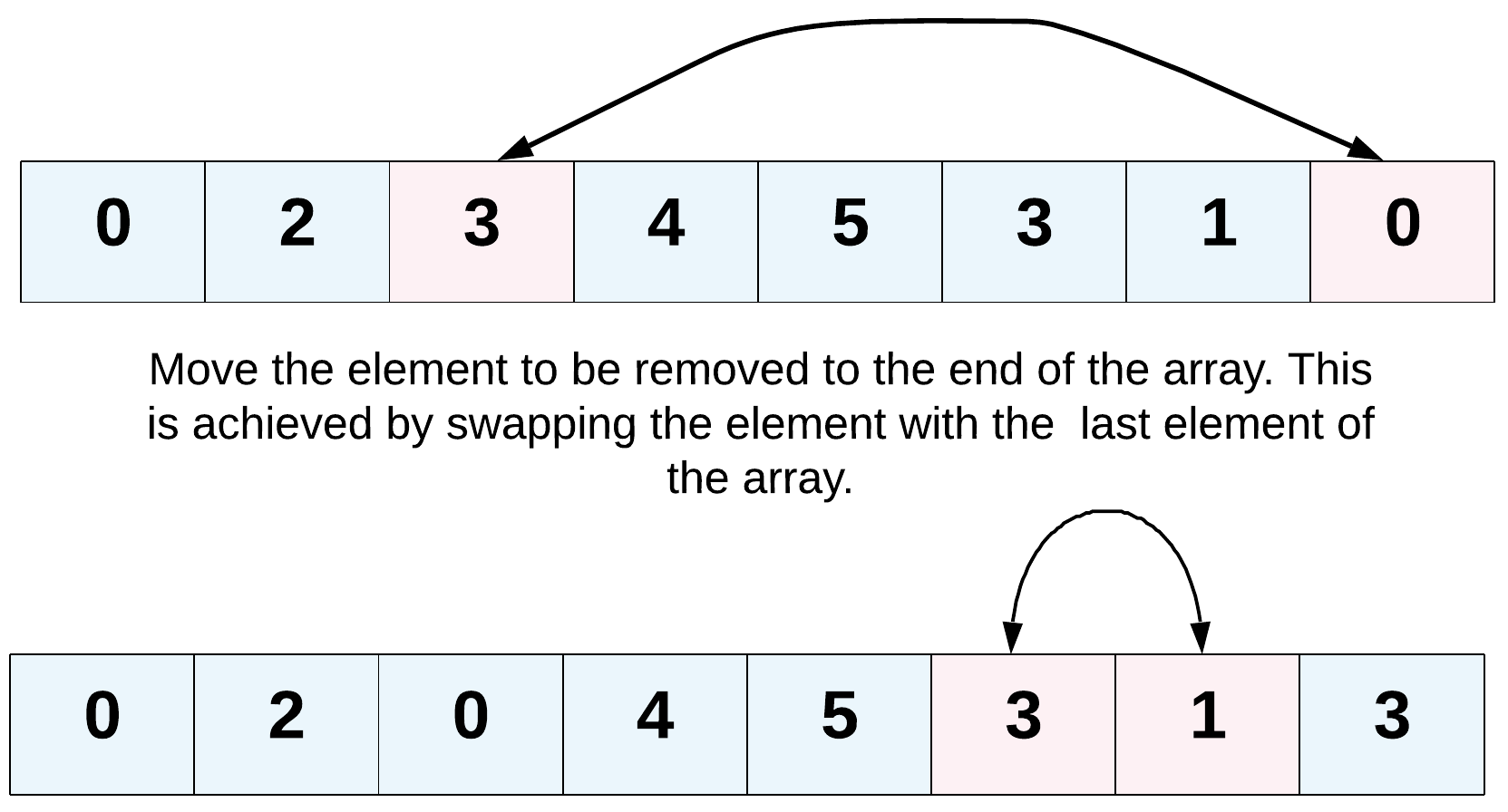
**Constraints:**

* 0 <= nums.length <= 100
* 0 <= nums[i] <= 50
* 0 <= val <= 100

Hide Hint #1

The problem statement clearly asks us to modify the array in-place and it also says that the element beyond the new length of the array can be anything. Given an element, we need to remove all the occurrences of it from the array. We don't technically need to **remove** that element per-say, right?

Hide Hint #2

We can move all the occurrences of this element to the end of the array. Use two pointers!  


Hide Hint #3

Yet another direction of thought is to consider the elements to be removed as non-existent. In a single pass, if we keep copying the visible elements in-place, that should also solve this problem for us.

# Solution

This is a pretty easy problem, but one may get confused by the term "in-place" and think 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 fact that the order of elements can be changed?
3. What happens when the elements to remove are rare?

## **Solution**

#### **Approach 1: Two Pointers**

**Intuition**

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

**Algorithm**

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

This solution is very similar to the solution to [Remove Duplicates from Sorted Array](https://leetcode.com/articles/remove-duplicates-from-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*). Assume the array has a total of n*n* elements, both i*i* and j*j* traverse at most 2n2*n* steps.
* Space complexity : O(1)*O*(1).

#### **Approach 2: Two Pointers - when elements to remove are rare**

**Intuition**

Now consider cases where the array contains few elements to remove. For example, nums = [1,2,3,5,4], val = 4*nums*=[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 = 4*nums*=[4,1,2,3,5],*val*=4. It seems unnecessary to move elements [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] = val*nums*[*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*). Both i*i* and n*n* traverse at most n*n* steps. In this approach, the number of assignment operations 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).