Summary

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) extra space. How to solve it? We can keep two pointers *i* and *j*, where *i* is the slow-runner while *j* is the fast-runner.

**Algorithm**

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

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

**Complexity analysis**

* Time complexity : *O*(*n*). Assume the array has a total of *n* elements, both i*i* and *j* traverse at most 2*n* steps.
* Space complexity :  *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. The previous algorithm will do unnecessary copy operation of the first four elements. Another example is nums = [4,1,2,3,5], val = 4. It seems unnecessary to move elements [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, 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*). Both i*i* and *n* traverse at most *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).