Adesiji Blessing Follow

Nov 21, 2020 · 5 min read · Listen

Arrays — Move Element to End

This article is the first part of a series of articles that will explain data structures and algorithms challenges. The purpose of this is to help solidify your knowledge as you practice for coding interviews.

Question

You are given an array of integers and an integer. Write a function that moves all instances of that integer in the array to the end of the array. Return the array. The function should mutate the input array, i.e. the operation should be performed in place, and doesn't need to maintain the order of the integers.

Sample input

```
array = [1, 2, 1, 1, 3, 4, 1]
toMove = 1
```

Sample Output

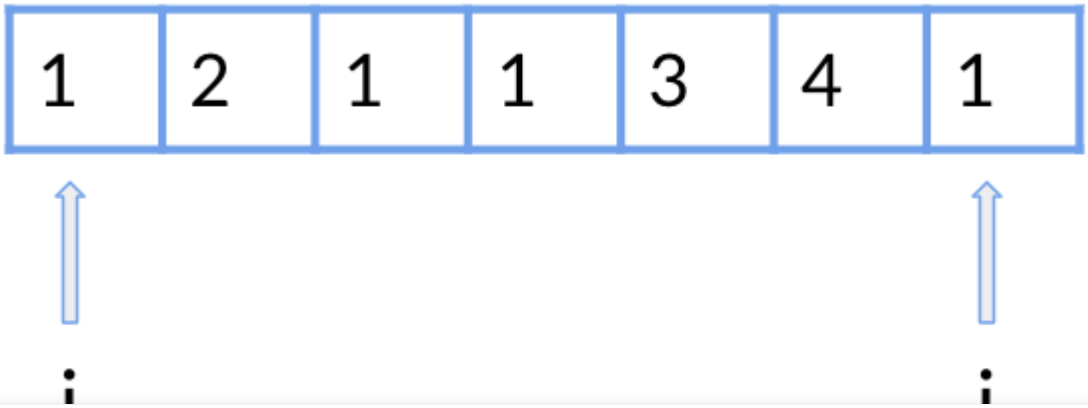
```
[4,2, 3, 1, 1, 1, 1]
```



How would you approach this qu 2 this tutorial for a detailed breakdown on how to solve this question.

Figuring out the Algorithm

You are provided with an integer to move all of the instances to the end of the array. The array can be potentially sorted to arrive at the solution, but it is not a viable method to follow. A better approach will be to have pointers (i, j) at each end element of the array. The pointer i is the pointer at the start of the array while pointer j is the pointer at the end of the array.

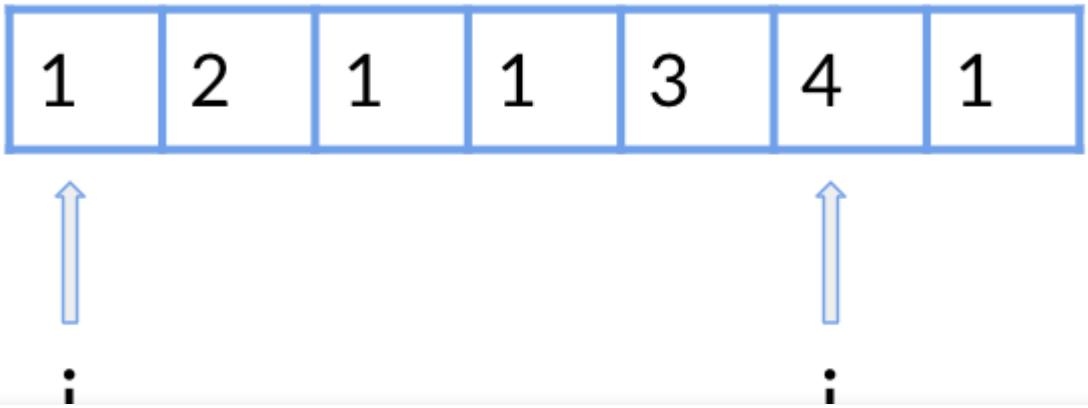


Pointers' Movement

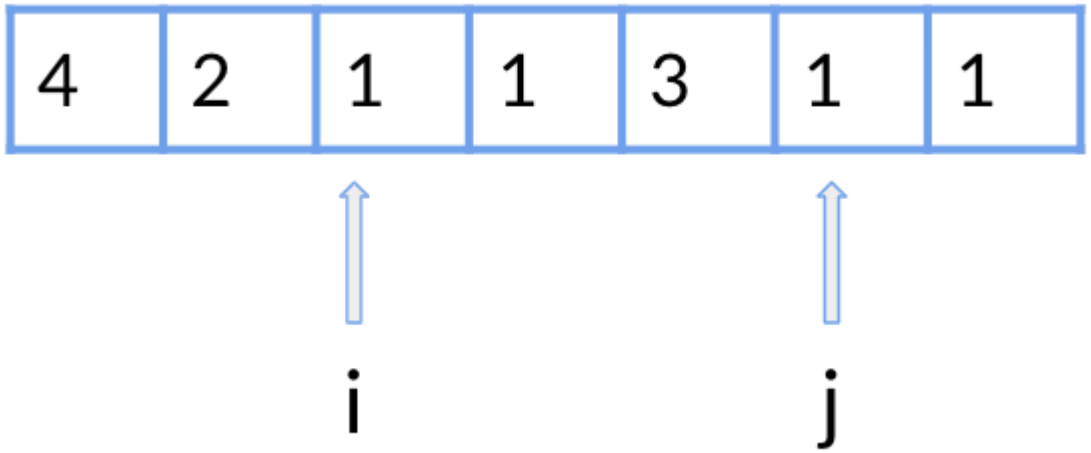
As seen above, the position of the pointers is at the extreme ends of the algorithm. To understand the movement of the pointers, think of it as an indicator to swap the elements to the left and right side of the array.

The simple approach to follow using the pointers is that the array element (1) should move or remain in the extreme right side of the array, while other array elements should move to the left side of the array.

From the figure above, the element to move (1) is at the end of the array already(right side), so the pointer j will move to the left side.

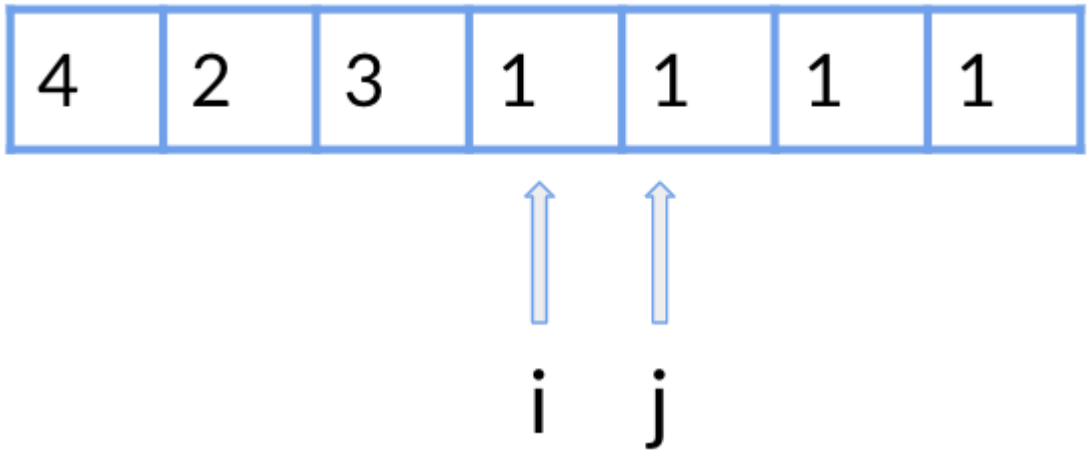


Next, swap element 1 at the left side (pointer i) with element 4 (pointer j). After this, the pointer i moves to the right side since the element 2 is in the expected position (See image below)



Current pointer position after doing the swap

Looking at the image above, the next step happens when pointer j moves to the left side after checking the condition. The new position of pointer j indicates that the current element should be swapped with the element at the position of the pointer i. (See image below)



Pointer position

At this point, we've traversed through the array as the pointers are set to overlap and cross to a different side.

Time & Space Complexity

Time Complexity

The two operations that were performed in the algorithm were accessing the index of each array of elements and swapping. So, the time complexity is linear time *O(N)*. No complex operation was carried out.

The space complexity is constant space as we didn't make use of any other data structure and the operations were carried out in place.

Coding the Algorithm

The first step here is to initialize the pointers as shown below.

```
def moveElementToEnd(array, toMove):

    i = 0

    j = len(array) - 1
```

Using a while loop with a condition to break after traversing through all the arrays. You would have traversed through all the arrays at the point where the pointers overlap.

Another while loop condition to capture is when the pointer j is pointing at the element to move at the right side of the array (1), we have to decrease the index of the array by 1 as we move inwards(left side) till we get to a value that is not



< j" to the inner while loop, and this prevents further decrement of pointer j and also prevent over swapping of the elements to the side as j decreases towards the left side.

```
def moveElementToEnd(array, toMove):

    i = 0

    j = len(array) - 1

    while i < j:

        while i < j and array[j] == toMove:

            j -= 1
```

So when the value at pointer j is not the value to move, we can perform the swap as long as the value at pointer i is equal to the value to move (1). After the swapping we increase the value of index i by moving to the right side. In the end you can return the array.



```
j = len(array) - 1

while i < j:

    while i < j and array[j] == toMove:

        j -= 1

    if array[i] == toMove:

        #Swapping

        array[i], array[j] = array[j], array[i]

    i += 1

return array
```

In conclusion, the problem becomes easy when you break it down into smaller chunks. Before your write any code, take time to think through the problem just like we did in the “Figuring out the algorithm” section.

Watch out for the next part of the tutorial, as we continue to tackle Array questions.