189. Rotate Array

815 492

Description (/problems/rotate-array/description/) Hints (/problems/rotate-array/hints/) Submissions (/problems/rotate-array/submissions/)

Quick Navigation View in Article (/articles/rotate-array/)

Notes

Summary

We have to rotate the elements of the given array k times to the right.

Solution

Approach #1 Brute Force [Time Limit Exceeded]

The simplest approach is to rotate all the elements of the array in k steps by rotating the elements by 1 unit in each step.

Java

```java
public class Solution {
    public void rotate(int[] nums, int k) {
        int temp, previous;
        for (int i = 0; i < k; i++) {
            previous = nums[nums.length - 1];
            for (int j = 0; j < nums.length; j++) {
                temp = nums[j];
                nums[j] = previous;
                previous = temp;
            }
        }
    }
}
```

Complexity Analysis

- Time complexity : $O(n*k)$. All the numbers are shifted by one step($O(n)$) k times($O(k)$).
- Space complexity : $O(1)$. No extra space is used.

Approach #2 Using Extra Array [Accepted]

Algorithm

We use an extra array in which we place every element of the array at its correct position i.e. the number at index i in the original array is placed at the index $(i+k)$. Then, we copy the new array to the original one.

Java

```
public class Solution {
    public void rotate(int[] nums, int k) {
        int[] a = new int[nums.length];
        for (int i = 0; i < nums.length; i++) {
            a[(i + k) % nums.length] = nums[i];
        }
        for (int i = 0; i < nums.length; i++) {
            nums[i] = a[i];
        }
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. One pass is used to put the numbers in the new array. And another pass to copy the new array to the original one.
- Space complexity : $O(n)$. Another array of the same size is used.

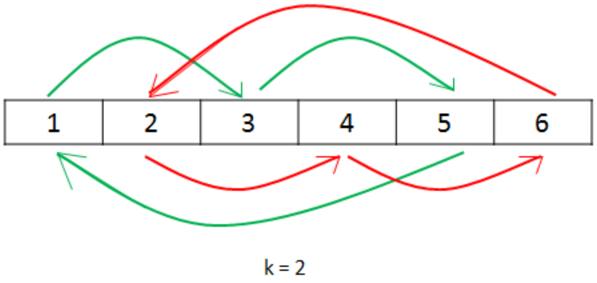
Approach #3 Using Cyclic Replacements [Accepted]

Algorithm

We can directly place every number of the array at its required correct position. But if we do that, we will destroy the original element. Thus, we need to store the number being replaced in a $temp$ variable. Then, we can place the replaced number($temp$) at its correct position and so on, n times, where n is the length of array. We have chosen n to be the number of replacements since we have to shift all the elements of the array(which is n). But, there could be a problem with this method, if n where $k = k$(since a value of k larger than n eventually leads to a k equivalent to k). In this case, while picking up numbers to be placed at the correct position, we will eventually reach the number from which we originally started. Thus, in such a case, when we hit the original number's index again, we start the same process with the number following it.

Now let's look at the proof of how the above method works. Suppose, we have n as the number of elements in the array and k is the number of shifts required. Further, assume n. Now, when we start placing the elements at their correct position, in the first cycle all the numbers with their index i satisfying i get placed at their required position. This happens because when we jump k steps every time, we will only hit the numbers k steps apart. We start with index $i = 0$, having i. Thus, we hit all the numbers satisfying the above condition in the first cycle. When we reach back the original index, we have placed $\frac{n}{k}$ elements at their correct position, since we hit only that many elements in the first cycle. Now, we increment the index for replacing the numbers. This time, we place other $\frac{n}{k}$ elements at their correct position, different from the ones placed correctly in the first cycle, because this time we hit all the numbers satisfy the condition i. When we hit the starting number again, we increment the index and repeat the same process from $i = 1$ for all the indices satisfying i. This happens till we reach the number with the index i again, which occurs for $i = k$. We will reach such a number after a total of k cycles. Now, the total count of numbers exclusive numbers placed at their correct position will be $k \times \frac{n}{k} = n$. Thus, all the numbers will be placed at their correct position.

Look at the following example to clarify the process: `nums: [1, 2, 3, 4, 5, 6] k: 2`



java

```
public class Solution {
    public void rotate(int[] nums, int k) {
        k = k % nums.length;
        int count = 0;
        for (int start = 0; count < nums.length; start++) {
            int current = start;
            int prev = nums[start];
            do {
                int next = (current + k) % nums.length;
                int temp = nums[next];
                nums[next] = prev;
                prev = temp;
                current = next;
                count++;
            } while (start != current);
        }
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. Only one pass is used.
- Space complexity : $O(1)$. Constant extra space is used.

Approach #4 Using Reverse [Accepted]

Algorithm

This approach is based on the fact that when we rotate the array k times, k elements from the back end of the array come to the front and the rest of the elements from the front shift backwards.

In this approach, we firstly reverse all the elements of the array. Then, reversing the first k elements followed by reversing the rest $n - k$ elements gives us the required result.

Let $n = 7$ and $k = 3$.

```
Original List                   : 1 2 3 4 5 6 7
After reversing all numbers     : 7 6 5 4 3 2 1
After reversing first k numbers : 5 6 7 4 3 2 1
After revering last n-k numbers : 5 6 7 1 2 3 4 --> Result
```

java

```
public class Solution {
    public void rotate(int[] nums, int k) {
        k %= nums.length;
        reverse(nums, 0, nums.length - 1);
        reverse(nums, 0, k - 1);
        reverse(nums, k, nums.length - 1);
    }
    public void reverse(int[] nums, int start, int end) {
        while (start < end) {
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            start++;
            end--;
        }
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. n elements are reversed a total of three times.
- Space complexity : $O(1)$. No extra space is used.

Analysis written by: @vinod23 (https://leetcode.com/vinod23)

Comments: 38

Sort By ▾

Type comment here... (Markdown is supported)

Preview Post

korte (/korte) ★ 1 September 16, 2018 10:42 PM

(/korte)

I can't understand the approach#3 vrey much.

1 Share Reply

SHOW 1 REPLY

novice87 (/novice87) ★ 34 September 12, 2018 12:37 AM

(/novice87)

For approach #4, shouldn't it loop until `start<=end` ?

0 Share Reply

SHOW 1 REPLY

x__CAS__x (/x__cas__x) ★ -1 August 30, 2018 6:02 PM

(/x__cas__x)

Where are checks for nums.length > 0 for all accepted answers?
All of them will fail with division by zero.

-1 Share Reply

SHOW 1 REPLY

olivercamel (/olivercamel) ★ 2 August 30, 2018 8:31 AM

(/olivercamel)

I think approach #3 should be faster in execution time than approach #4. Because for #4, almost all elements in the array are moved 2 times. (1st reverse = 1st time, 2nd + 3rd reverse = 2nd time).
But in #3 all elements only moved 1 time and they are in the correct location.
In embedded world, writing data to RAM is a costly action in execution. So #4 would be slower.
However, I see in submission statistics that #4 based answers are faster (like 4ms in C) than #3 based

Read More

0 Share Reply