Previous Next

283. Move Zeroes

June 27, 2016 | 444.4K views

★★★★★
Average Rating: 4.57 (167 votes)

Given an array `nums`, write a function to move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

Example:

```
Input: [0,1,0,3,12]
Output: [1,3,12,0,0]
```

Note:

1. You must do this **in-place** without making a copy of the array.
2. Minimize the total number of operations.

Solution

This question comes under a broad category of "Array Transformation". This category is the meat of tech interviews. Mostly because arrays are such a simple and easy to use data structure. Traversal or representation doesn't require any boilerplate code and most of your code will look like the Pseudocode itself.

The 2 requirements of the question are:

1. Move all the 0's to the end of array.

2. All the non-zero elements must retain their original order.

It's good to realize here that both the requirements are mutually exclusive, i.e., you can solve the individual sub-problems and then combine them for the final solution.

Approach #1 (Space Sub-Optimal) [Accepted]

C++

```
void moveZeroes(vector<int>& nums) {
    int n = nums.size();

    // Count the zeroes
    int numZeroes = 0;
    for (int i = 0; i < n; i++) {
        numZeroes += (nums[i] == 0);
    }

    // Make all the non-zero elements retain their original order.
    vector<int> ans;
    for (int i = 0; i < n; i++) {
        if (nums[i] != 0) {
            ans.push_back(nums[i]);
        }
    }

    // Move all zeroes to the end
    while (numZeroes--) {
        ans.push_back(0);
    }

    // Combine the result
    for (int i = 0; i < n; i++) {
        nums[i] = ans[i];
    }
}
```

Complexity Analysis

Space Complexity : $O(n)$. Since we are creating the "ans" array to store results.

Time Complexity: $O(n)$. However, the total number of operations are sub-optimal. We can achieve the same result in less number of operations.

If asked in an interview, the above solution would be a good start. You can explain the interviewer(not code) the above and build your base for the next Optimal Solution.

Approach #2 (Space Optimal, Operation Sub-Optimal) [Accepted]

This approach works the same way as above, i.e. , first fulfills one requirement and then another. The catch? It does it in a clever way. The above problem can also be stated in alternate way, " Bring all the non 0 elements to the front of array keeping their relative order same".

This is a 2 pointer approach. The fast pointer which is denoted by variable "cur" does the job of processing new elements. If the newly found element is not a 0, we record it just after the last found non-0 element. The position of last found non-0 element is denoted by the slow pointer "lastNonZeroFoundAt" variable. As we keep finding new non-0 elements, we just overwrite them at the "lastNonZeroFoundAt + 1" 'th index. This overwrite will not result in any loss of data because we already processed what was there(if it were non-0,it already is now written at it's corresponding index,or if it were 0 it will be handled later in time).

After the "cur" index reaches the end of array, we now know that all the non-0 elements have been moved to beginning of array in their original order. Now comes the time to fulfil other requirement, "Move all 0's to the end". We now simply need to fill all the indexes after the "lastNonZeroFoundAt" index with 0.

C++

```
void moveZeroes(vector<int>& nums) {
    int lastNonZeroFoundAt = 0;
    // If the current element is not 0, then we need to
    // append it just in front of last non 0 element we found.
    for (int i = 0; i < nums.size(); i++) {
        if (nums[i] != 0) {
            nums[lastNonZeroFoundAt++] = nums[i];
        }
    }
    // After we have finished processing new elements,
    // all the non-zero elements are already at beginning of array.
    // We just need to fill remaining array with 0's.
    for (int i = lastNonZeroFoundAt; i < nums.size(); i++) {
        nums[i] = 0;
    }
}
```

Complexity Analysis

Space Complexity : $O(1)$. Only constant space is used.

Time Complexity: $O(n)$. However, the total number of operations are still sub-optimal. The total operations (array writes) that code does is n (Total number of elements).

Approach #3 (Optimal) [Accepted]

The total number of operations of the previous approach is sub-optimal. For example, the array which has all (except last) leading zeroes: [0, 0, 0, ..., 0, 1].How many write operations to the array? For the previous approach, it writes 0's $n-1$ times, which is not necessary. We could have instead written just once. How? ..... By only fixing the non-0 element,i.e., 1.

The optimal approach is again a subtle extension of above solution. A simple realization is if the current element is non-0, its' correct position can at best be it's current position or a position earlier. If it's the latter one, the current position will be eventually occupied by a non-0 ,or a 0, which lies at a index greater than 'cur' index. We fill the current position by 0 right away,so that unlike the previous solution, we don't need to come back here in next iteration.

In other words, the code will maintain the following invariant:

> 1. All elements before the slow pointer (lastNonZeroFoundAt) are non-zeroes.
> 2. All elements between the current and slow pointer are zeroes.

Therefore, when we encounter a non-zero element, we need to swap elements pointed by current and slow pointer, then advance both pointers. If it's zero element, we just advance current pointer.

With this invariant in-place, it's easy to see that the algorithm will work.

C++

```
void moveZeroes(vector<int>& nums) {
    for (int lastNonZeroFoundAt = 0, cur = 0; cur < nums.size(); cur++) {
        if (nums[cur] != 0) {
            swap(nums[lastNonZeroFoundAt++], nums[cur]);
        }
    }
}
```

Complexity Analysis

Space Complexity : $O(1)$. Only constant space is used.

Time Complexity: $O(n)$. However, the total number of operations are optimal. The total operations (array writes) that code does is Number of non-0 elements.This gives us a much better best-case (when most of the elements are 0) complexity than last solution. However, the worst-case (when all elements are non-0) complexity for both the algorithms is same.

Analysis written by: @spandan.pathak

Rate this article: ★★★★★

Previous Next

Comments: 221

Sort By

Type comment here... (Markdown is supported)

Preview | Post

kevin217 ★ 79 April 25, 2018 1:34 PM

Solution 2 is always better than Solution 3. On average it's less ops (unless you believe swap a single op...), and the claim Solution 3 is better when most are 0 doesn't hold water either. If most are zero, the last step of Solution 2 can be optimized with a simple memset.

79 | Share | Reply

SHOW 4 REPLIES

wguo32 ★ 55 May 2, 2018 11:31 PM

public void moveZeroes(int[] nums) {
int pos = 0;
for(int i = 0; i < nums.length; i++){
if(nums[i] != 0){
nums[pos]= nums[i];

Read More

45 | Share | Reply

SHOW 1 REPLY

williamfu4leetcode ★ 34 April 20, 2018 3:49 AM

I don't think the solution 3 is better than solution 2. It depends on the cases. Since swap is actually writing at least 2 times (if writing to temp variables don't count..there are other techniques like bitwise operations to swap, but will take more operations), a case of kicking a zero from the beginning to the end would be horrible. ie. [0,1,1,1,1,1]. That's almost 2*n writes, while solution 2 will take only n writes.

30 | Share | Reply

SHOW 1 REPLY

miketung2013 ★ 54 December 29, 2018 9:27 PM

Here's my python solution in O(n) I believe

```
class Solution:
    def moveZeroes(self, nums):
        """
```

Read More

45 | Share | Reply

SHOW 8 REPLIES

terrible_whiteboard ★ 633 May 19, 2020 6:18 PM

I made a video if anyone is having trouble understanding the solution (clickable link)
https://youtu.be/0rPulLjoVsg

Read More

22 | Share | Reply

watermoon008 ★ 18 April 26, 2019 6:51 AM

Approach #3
probably we need to check weather lastNonZeroFoundAt equals to cur before we take swap operation.
just imaging the testcase [1, 2, 3, 4]
The solution do four swap operations. In fact we need zero.

17 | Share | Reply

wei44 ★ 14 November 18, 2018 9:09 PM

This isn't optimal because if let's say nums[0] isn't zero you would have swapped num[0] with itself. You will need to check nums[lastNonZeroFoundAt] == 0 before swapping to avoid that.

14 | Share | Reply

browe004 ★ 21 October 13, 2018 9:34 AM

Python Solution:

```
        while 0 in nums:
            for i, num in enumerate(nums):
                if nums[i] == 0:
```

Read More

14 | Share | Reply

SHOW 6 REPLIES

Dr_Sean ★ 542 April 24, 2019 3:13 AM

Python solution:
Starts from the first element and goes toward the end, if the element is zero, it pops the element and appends it to the end.

```
class Solution(object):
```

Read More

12 | Share | Reply

SHOW 5 REPLIES

bowensun1224 ★ 19 December 9, 2018 8:41 AM

Python3 Solution:Not fast but easy understand:

```
class Solution(object):
    def moveZeroes(self, nums):
        append_times=nums.count(0)
```

Read More

10 | Share | Reply

‹ 1 2 3 4 5 6 ... 22 23 ›