# Question

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.

Hide Hint #1

**In-place** means we should not be allocating any space for extra array. But we are allowed to modify the existing array. However, as a first step, try coming up with a solution that makes use of additional space. For this problem as well, first apply the idea discussed using an additional array and the in-place solution will pop up eventually.

Hide Hint #2

A **two-pointer** approach could be helpful here. The idea would be to have one pointer for iterating the array and another pointer that just works on the non-zero elements of the array.

# 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)*O*(*n*). Since we are creating the "ans" array to store results.

Time Complexity: O(n)*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)*O*(1). Only constant space is used.

Time Complexity: O(n)*O*(*n*). However, the total number of operations are still sub-optimal. The total operations (array writes) that code does is n*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*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)*O*(1). Only constant space is used.

Time Complexity: O(n)*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

<https://www.youtube.com/watch?v=0rPuILjoVsg&feature=emb_logo&ab_channel=TerribleWhiteboard>

|  |
| --- |
| // Shift non-zero values as far forward as possible  // Fill remaining space with zeros  public void moveZeroes(int[] nums) {  if (nums == null || nums.length == 0) return;  int insertPos = 0;  for (int num: nums) {  if (num != 0) nums[insertPos++] = num;  }  while (insertPos < nums.length) {  nums[insertPos++] = 0;  }  } |