# Question

Given a fixed length array arr of integers, duplicate each occurrence of zero, shifting the remaining elements to the right.

Note that elements beyond the length of the original array are not written.

Do the above modifications to the input array **in place**, do not return anything from your function.

**Example 1:**

**Input:** [1,0,2,3,0,4,5,0]

**Output:** null

**Explanation:** After calling your function, the **input** array is modified to: [1,0,0,2,3,0,0,4]

**Example 2:**

**Input:** [1,2,3]

**Output:** null

**Explanation:** After calling your function, the **input** array is modified to: [1,2,3]

**Note:**

1. 1 <= arr.length <= 10000
2. 0 <= arr[i] <= 9

Hide Hint #1

This is a great introductory problem for understanding and working with the concept of in-place operations. The problem statement clearly states that we are to modify the array in-place. That does not mean we cannot use another array. We just don't have to return anything.

   Hide Hint #2

A better way to solve this would be without using additional space. The only reason the problem statement allows you to make modifications in place is that it hints at avoiding any additional memory.

Hide Hint #3

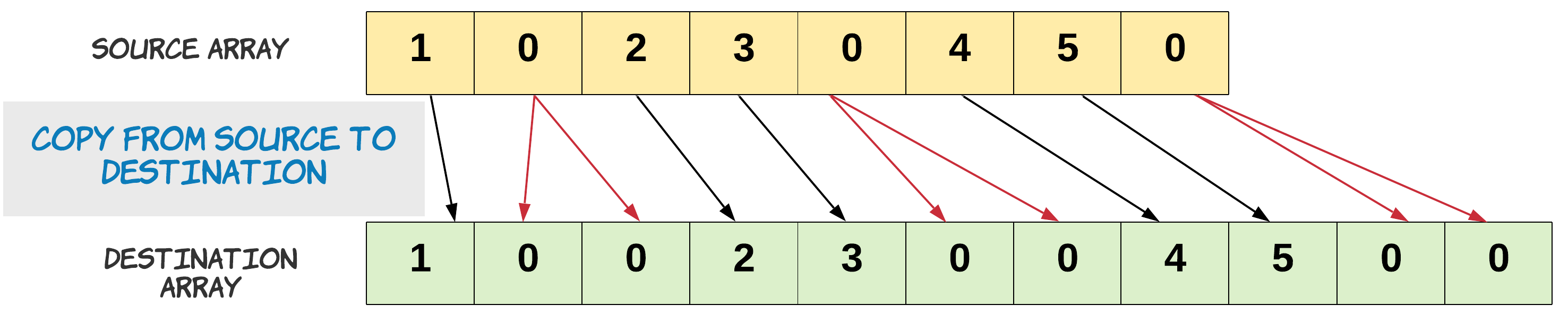
The main problem with not using additional memory is that we might override elements due to the zero duplication requirement of the problem statement. How do we get around that?

Hide Hint #4

If we had enough space available, we would be able to accommodate all the elements properly. The new length would be the original length of the array plus the number of zeros. Can we use this information somehow to solve the problem?

# Solution

The problem demands the array to be modified in-place. If in-place was not a constraint we might have just copied the elements from a source array to a destination array.



Notice, how we copied zero twice.

s = 0

d = 0

# Copy is performed until the destination array is full.

for s in range(N):

if source[s] == 0:

# Copy zero twice.

destination[d] = 0

d += 1

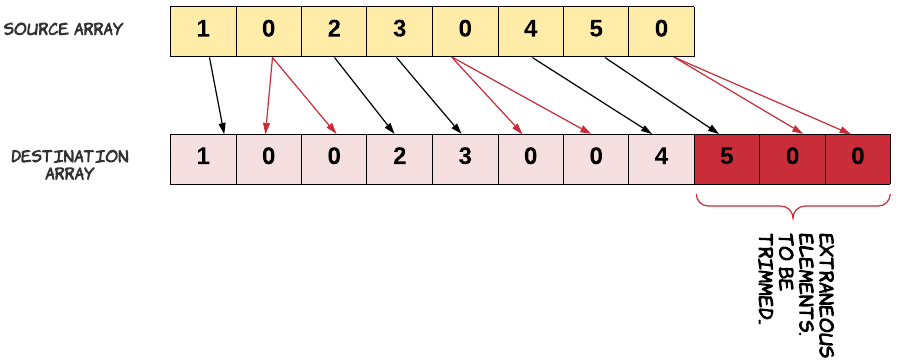
destination[d] = 0

else:

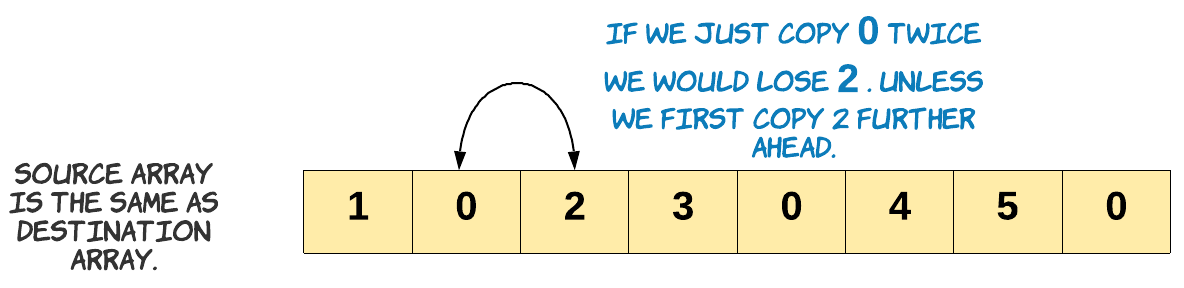
destination[d] = source[s]

d += 1

The problem statement also mentions that we do not grow the new array, rather we just trim it to its original array length. This means we have to discard some elements from the end of the array. These are the elements whose new indices are beyond the length of the original array.



Let's remind ourselves about the problem constraint that we are given. Since we can't use extra space, our source and destination array is essentially the same. We just can't go about copying the source into destination array the same way. If we do that we would lose some elements. Since, we would be overwriting the array.



Keeping this in mind, in the approach below we start copying to the end of the array.

#### **Approach 1: Two pass, O(1) space**

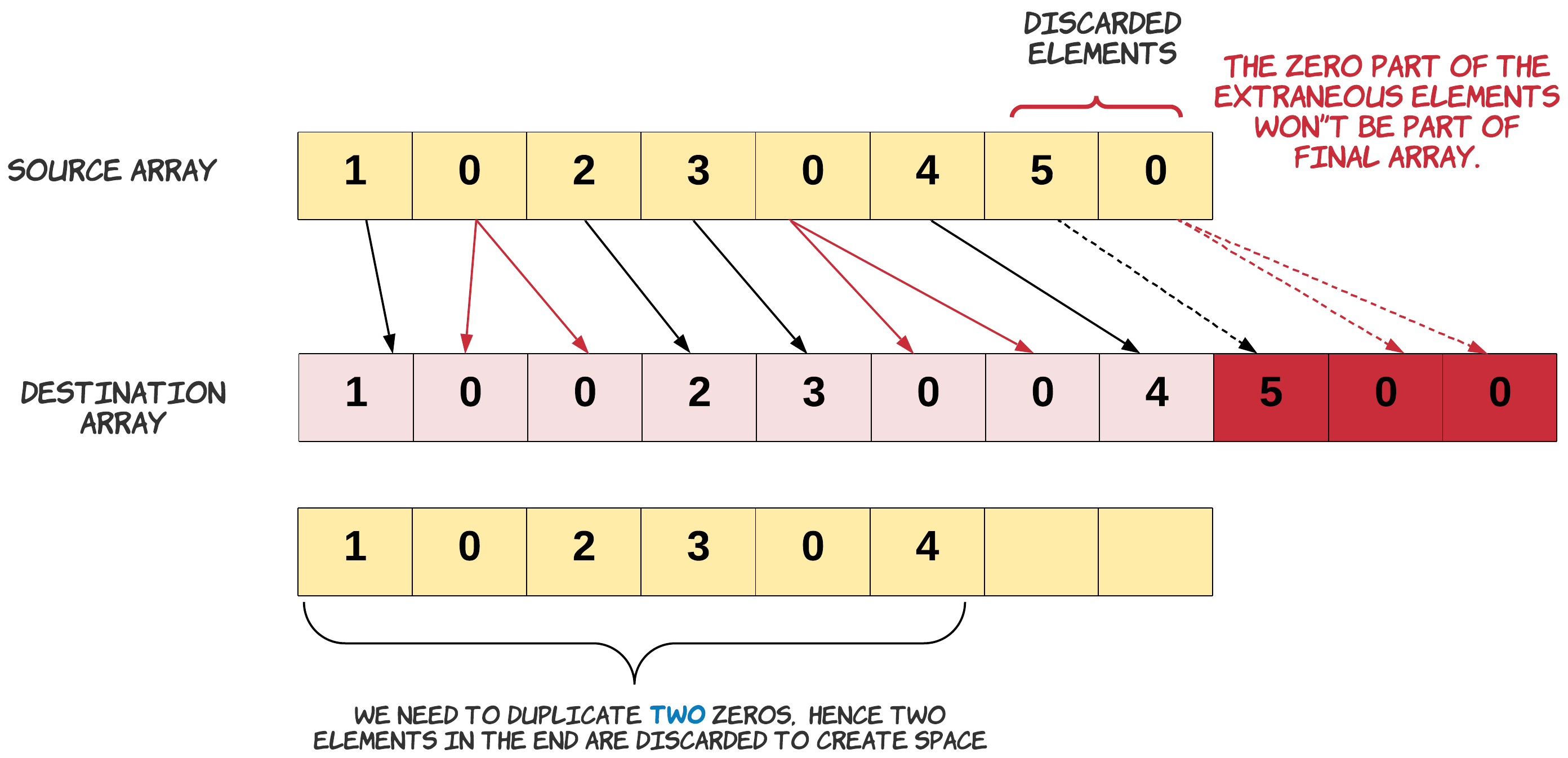
**Intuition**

If we know the number of elements which would be discarded from the end of the array, we can copy the rest. How do we find out how many elements would be discarded in the end? The number would be equal to the number of extra zeros which would be added to the array. The extra zero would create space for itself by pushing out an element from the end of the array.

Once we know how many elements from the original array would be part of the final array, we can just start copying from the end. Copying from the end ensures we don't lose any element since, the last few extraneous elements can be overwritten.

**Algorithm**

1. Find the number of zeros which would be duplicated. Let's call it possible\_dups. We do need to make sure we are not counting the zeros which would be trimmed off. Since, the discarded zeros won't be part of the final array. The count of possible\_dups would give us the number of elements to be trimmed off the original array. Hence at any point, length\_ - possible\_dups is the number of elements which would be included in the final array.



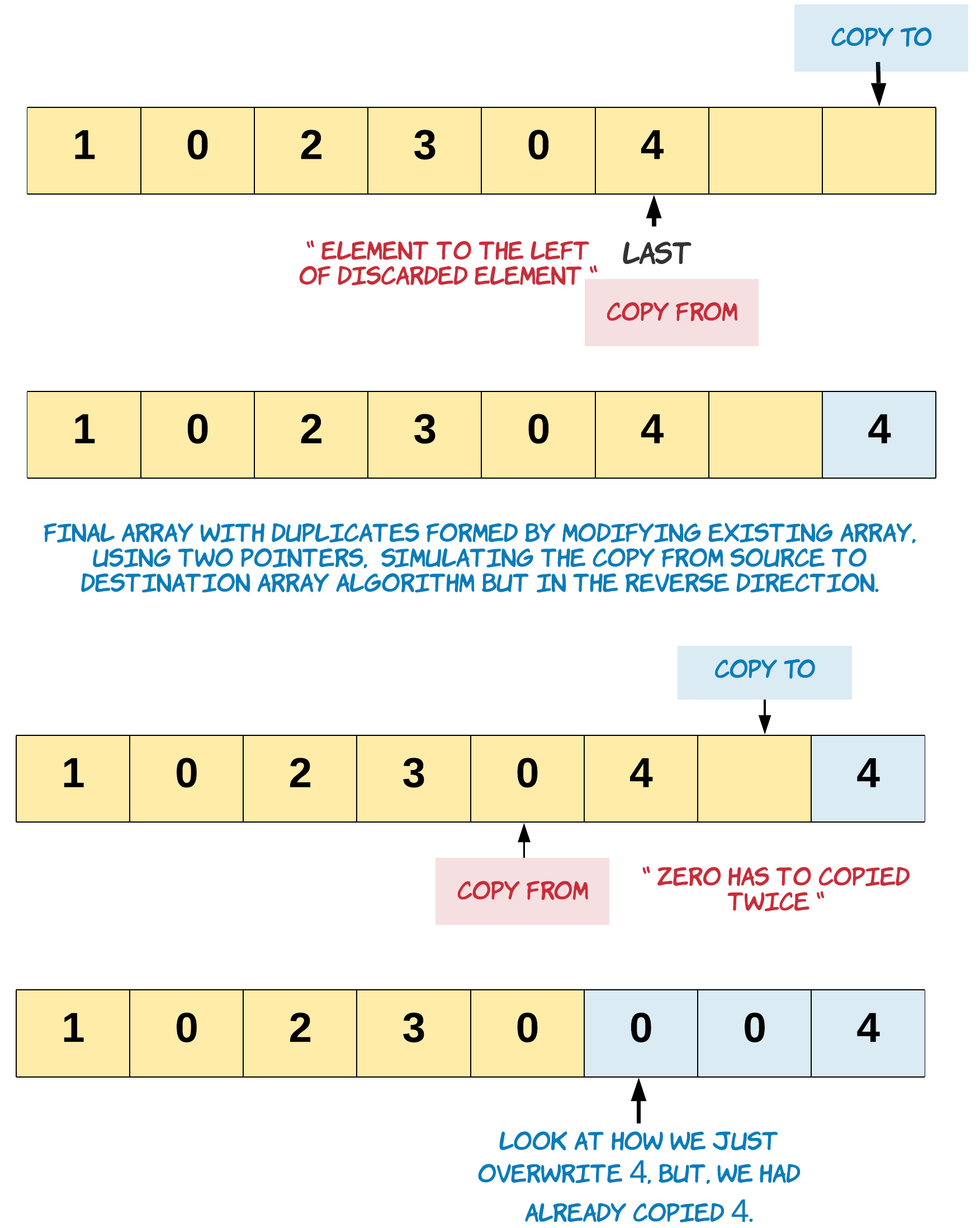
Note: In the diagram above we just show source and destination array for understanding purpose. We will be doing these operations only on one array.

1. Handle the edge case for a zero present on the boundary of the leftover elements.

Let's talk about the edge case of this problem. We need to be extra careful when we are duplicating the zeros in the leftover array. This care should be taken for the zero which is lying on the boundary. Since, this zero might be counted as with possible duplicates, or may be just got included in the left over when there was no space left to accommodate its duplicate. If it is part of the possible\_dups we would want to duplicate it otherwise we don't.

An example of the edge case is - [8,4,5,0,0,0,0,7]. In this array there is space to accommodate the duplicates of first and second occurrences of zero. But we don't have enough space for the duplicate of the third occurrence of zero. Hence when we are copying we need to make sure for the third occurrence we don't copy twice. Result = [8,4,5,0,0,0,0,0]

1. Iterate the array from the end and copy a non-zero element once and zero element twice. When we say we discard the extraneous elements, it simply means we start from the left of the extraneous elements and start overwriting them with new values, eventually right shifting the left over elements and creating space for all the duplicated elements in the array.



|  |
| --- |
| class Solution {  public void duplicateZeros(int[] arr) {  int possibleDups = 0;  int length\_ = arr.length - 1;  // Find the number of zeros to be duplicated  // Stopping when left points beyond the last element in the original array  // which would be part of the modified array  for (int left = 0; left <= length\_ - possibleDups; left++) {  // Count the zeros  if (arr[left] == 0) {  // Edge case: This zero can't be duplicated. We have no more space,  // as left is pointing to the last element which could be included  if (left == length\_ - possibleDups) {  // For this zero we just copy it without duplication.  arr[length\_] = 0;  length\_ -= 1;  break;  }  possibleDups++;  }  }  // Start backwards from the last element which would be part of new array.  int last = length\_ - possibleDups;  // Copy zero twice, and non zero once.  for (int i = last; i >= 0; i--) {  if (arr[i] == 0) {  arr[i + possibleDups] = 0;  possibleDups--;  arr[i + possibleDups] = 0;  } else {  arr[i + possibleDups] = arr[i];  }  }  }  } |

|  |
| --- |
| class Solution:  def duplicateZeros(self, arr: List[int]) -> None:  """  Do not return anything, modify arr in-place instead.  """  possible\_dups = 0  length\_ = len(arr) - 1  # Find the number of zeros to be duplicated  for left in range(length\_ + 1):  # Stop when left points beyond the last element in the original list  # which would be part of the modified list  if left > length\_ - possible\_dups:  break  # Count the zeros  if arr[left] == 0:  # Edge case: This zero can't be duplicated. We have no more space,  # as left is pointing to the last element which could be included  if left == length\_ - possible\_dups:  arr[length\_] = 0 # For this zero we just copy it without duplication.  length\_ -= 1  break  possible\_dups += 1  # Start backwards from the last element which would be part of new list.  last = length\_ - possible\_dups  # Copy zero twice, and non zero once.  for i in range(last, -1, -1):  if arr[i] == 0:  arr[i + possible\_dups] = 0  possible\_dups -= 1  arr[i + possible\_dups] = 0  else:  arr[i + possible\_dups] = arr[i] |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the number of elements in the array. We do two passes through the array, one to find the number of possible\_dups and the other to copy the elements. In the worst case we might be iterating the entire array, when there are less or no zeros in the array.
* Space Complexity: O(1)*O*(1). We do not use any extra space.