**Arrays 101**

Introduction

Arrays are a simple data structure for storing lots of similar items. They exist in all programming languages, and are used as the basis for most other data structures. On their own, Arrays can be used to solve many interesting problems. Arrays come up very often in interview problems, and so being a guru with them is a must!

In this Explore Card, we'll introduce Arrays and solve some cool problems with them. This Card is **beginner friendly**, and we've provided lots of code snippets in **Java** to help you understand. Each topic begins with informative articles, followed by real interview problems for you to practice on.

In addition, the problems have *hints* which will help you in building up ideas on how to solve them. These hints will be subtle enough to set you on a particular path for reaching the optimized solution, without giving away the answer too easily.

After completing this Explore Card on Arrays, you will understand:

* What an Array is.
* Basic properties of Arrays.
* Implementing basic Array operations.
* Simple programming techniques with Arrays.

Before you start, bear in mind that should you have any questions or comments, you can always post them in the [Discussion](https://leetcode.com/discuss/explore/fun-with-arrays) forum that is located at the end of the card. We'll do our best to respond to you as soon as we can.

## Introduction

In this chapter, we'll begin by looking at what an Array is, and what it's used for. We'll start by comparing Arrays to a real-world problem: storing lots of DVDs in an organized way. After that, we'll look at how to create and work with Arrays in **Java**.

**Array - A DVD box?**

Suppose you had a bunch of DVDs at home that you wanted to arrange neatly. What would be the ideal choice for storing such a thing? You could find a cardboard box (or some other box) big enough to arrange all of the DVDs neatly, right? It's as simple as that. However, you might want to add a new DVD to the box, or you might want to get rid of the old ones that you've watched a million times over in the past. An important consideration for this box would be that you would only place DVDs in it and nothing else; you wouldn't place your clothes in it, for example. The box would contain multiple items, but all of them would be of the same type. In this case, that type is DVD. Items of the same type share properties. For DVDs, those properties include:

* All the DVDs would be inside a plastic cover.
* The cover would have the name of the movie, the cast, and all sorts of other details.
* All the covers would be of exactly the same size and would contain just one, and only one, DVD.

You might not actually name the DVD box, but when you want your sister to fetch a DVD, you'd tell her that the DVD is inside your "DVD box", and she would instantly know where to find the box. This is a very simple yet realistic scenario that is easy to understand and relate to. So, now let us move over to the world of computers and port this example to programming.

Text

Description automatically generated

Suppose you were told that you needed to build some software to keep track of all the DVDs in an inventory. This is the exact same scenario that we just described above, but on a much larger scale. So let's imagine the DVD box as a virtual DVD library. For each DVD, you would have certain properties that would be specific attributes of the movies themselves.

In addition to the properties of a DVD, you're also told the maximum number of DVDs that can be stored in the inventory. Obviously, you wouldn't want to store ancient movies from the 1900s unless they were popular ones, right? Say you were told that the requirement is to maintain a maximum inventory of just 100 DVDs. This is an important piece of information because, without this, you wouldn't be able to find the *perfectly sized* box to fit all the DVDs easily. How could we find a box of a particular size that would be able to fit a maximum of 100 DVDs? Well, lucky for us, we don't need to physically find a cardboard box or anything—there's a programming construct for this purpose. That programming construct is known as an Array.

**What Is an Array?**

An Array is a collection of items. The items could be integers, strings, DVDs, games, books—anything really. The items are stored in neighboring (contiguous) memory locations. Because they're stored together, checking through the entire collection of items is straightforward.

So, how can we relate this back to the physical DVDs? Well, do you keep your DVDs all around the house in multiple locations? Hopefully not! Most people keep all of their DVDs right next to one another inside one gigantic box, or perhaps on a bookshelf. We do this so that if we need to find a particular DVD, we can quickly search through all of them without running from room to room.

### **Creating an Array**

On a computer, Arrays can hold up to N items. The value of N is decided by you, the programmer, at the time you create the Array. This is the same as when we found a big enough cardboard box for the DVDs. Additionally, you also need to specify the type of item that will be going into the Array.

In **Java**, we use the following code to create an Array to hold up to 15 DVDs. Note that we've also included a simple definition of a DVD for clarity.

|  |
| --- |
| // The actual code for creating an Array to hold DVD's.  DVD[] dvdCollection = new DVD[15];  // A simple definition for a DVD.  public class DVD {  public String name;  public int releaseYear;  public String director;  public DVD(String name, String releaseYear, String director) {  this.name = name;  this.releaseYear = releaseYear;  this.director = director;  }  public String toString() {  System.out.println(  this.name + ", directed by " + this.director + ", released in " + this.releaseYear));  }  } |

After running the above code, we now have an Array called dvdCollection, with 15 places in it. Each place can hold *one* DVD. At the start, there are *no* DVD's in the Array; we'll have to actually put them in.

The Array can only hold up to 15 DVDs. If we get a 16th DVD, we'll need to make a new Array. We'll look at how we deal with running out of space, in the next chapter.

Before we move onto actually putting some DVDs into the Array, though, one thing you might be wondering is why we'd make an Array with *only* 15 places. Why not just make it hold 1000000 DVDs so that we *know for sure* we'll always have enough space?

Well, the reason is the same as it is for the physical box of DVDs. Do you really want to find a box that could hold 1000000 DVDs when you currently only have 15 DVDs and, in fact, never expect to own more than 100 of them? Is it even worth getting a box that could hold 100 DVDs right now, when you only expect to get a few new ones each year? It will take up a lot more space in your home in the meantime.

It's exactly the same with the Array, where the space in your home is analogous to memory on the computer. If you make an Array with 1000000 spaces, the computer will reserve memory to hold 1000000 DVDs, even if you only put 15 DVDs into it. That memory can't be used for anything else in the meantime—just like the space in your house that has been taken over by that *huge* cardboard box!

**Accessing Elements in Arrays**

The two most primitive Array operations are writing elements into them, and reading elements from them. All other Array operations are built on top of these two primitive operations.

### **Writing Items into an Array**

To put a DVD into the Array, we need to decide which of the 15 places we'd like it to go in. Each of the places is identified using a number in the range of 0 to N - 1. The 1st place is 0, the 2nd place is 1, the 3rd place is 2... all the way up to the 15th place, which is 14. We call these numbers that identify each place **indexes**.

Let's put the DVD for The Avengers into the eighth place of the Array we created above.

|  |
| --- |
| // Firstly, we need to actually create a DVD object for The Avengers.  DVD avengersDVD = new DVD("The Avengers", 2012, "Joss Whedon");  // Next, we'll put it into the 8th place of the Array. Remember, because we  // started numbering from 0, the index we want is 7.  dvdCollection[7] = avengersDVD; |

And that's it. We've put the DVD for *The Avengers* into our Array! Let's put a few more DVD's in.

|  |
| --- |
| DVD incrediblesDVD = new DVD("The Incredibles", 2004, "Brad Bird");  DVD findingDoryDVD = new DVD("Finding Dory", 2016, "Andrew Stanton");  DVD lionKingDVD = new DVD("The Lion King", 2019, "Jon Favreau");  // Put "The Incredibles" into the 4th place: index 3.  dvdCollection[3] = incrediblesDVD;  // Put "Finding Dory" into the 10th place: index 9.  dvdCollection[9] = findingDoryDVD;  // Put "The Lion King" into the 3rd place: index 2.  dvdCollection[2] = lionKingDVD; |

Notice that we put *The Incredibles* into the Array at index 3. What happens if we now run this next piece of code?

|  |
| --- |
| DVD starWarsDVD = new DVD("Star Wars", 1977, "George Lucas");  dvdCollection[3] = starWarsDVD; |

Because we just put Star Wars into the Array at index 3, The Incredibles is no longer in the Array. It has been overwritten! If we still have the incrediblesDVD variable in scope, then the DVD still exists in the computer's memory. If not though, it's totally gone!

### **Reading Items from an Array**

We can check what's at a particular Array index.

|  |
| --- |
| // Print out what's in indexes 7, 10, and 3.  System.out.println(dvdCollection[7]);  System.out.println(dvdCollection[10]);  System.out.println(dvdCollection[3]);  // Will print:  // The Avengers, directed by Joss Whedon, released in 2012  // null  // Star Wars, directed by George Lucas, released in 1977 |

Notice that because we haven't yet put anything at index 10, the value it contains is null. In other languages, such as **C**, the Array slot could contain completely random data. Java always initializes empty Array slots to null if the Array contains *objects*, or to default values if it contains *primitive types*. For example, the array int [] would contain the default value of 0 for each element, float[] would contain default values of 0.0, and bool[] would contain default values of false.

### **Writing Items into an Array with a Loop**

We commonly use a loop to put lots of values into an Array. To illustrate this, let's go to another example. This time, we're going to create an Array of ints and put the first 10 square numbers into it.

|  |
| --- |
| int[] squareNumbers = new int[10];  // Go through each of the Array indexes, from 0 to 9.  for (int i = 0; i < 10; i++) {  // We need to be careful with the 0-indexing. The next square number  // is given by (i + 1) \* (i + 1).  // Calculate it and insert it into the Array at index i.  int square = (i + 1) \* (i + 1);  squareNumbers[i] = square;  } |

### **Reading Items from an Array with a Loop**

We can also use a loop to print out everything that's in the Array.

|  |
| --- |
| // Go through each of the Array indexes, from 0 to 9.  for (int i = 0; i < 10; i++) {  // Access and print what's at the i'th index.  System.out.println(squareNumbers[i]);  }  // Will print:  // 1  // 4  // 9  // 16  // 25  // 36  // 49  // 64  // 81  // 100 |

One last thing worth knowing now is that there's a more elegant way of printing out the values of an Array—a variant of the for loop, commonly referred to as a "for each" loop.

|  |
| --- |
| // For each VALUE in the Array.  for (int square : squareNumbers) {  // Print the current value of square.  System.out.println(square);  }  // Prints exactly the same as the previous example. |

You'll probably agree that this code is a lot simpler to read. We can use it whenever we don't need the index values. For actually *writing* the squares into the Array, it wouldn't have worked because we needed to work with the actual index numbers. You don't *have* to use a "for each" loop when you're starting out, but we recommend you become comfortable with it before interviews. Simple, elegant code is good code!

**Array Capacity VS Length**

If somebody asks you how long the DVD Array is, what what would your answer be?

There are two different answers you might have given.

1. The number of DVDs the box could hold, if it was full, or
2. The number of DVDs currently in the box.

Both answers are correct, and both have very different meanings! It's important to understand the difference between them, and use them correctly. We call the first one the **capacity** of the Array, and the second one the **length** of the Array.

### **Array Capacity**

Let's say we've created a new Array like this.

DVD[] array = new DVD[6]

Is it a valid operation to insert an element at array[6]? What about at array[10]?

Nope, neither of these are valid. When we created the Array, we specified that it can hold up to 6 DVD's. This is the Array's **capacity**.

Remembering that indexing starts at 0, we can only insert items at array[0], array[1], array[2], array[3], array[4], and array[5]. Trying to put an element anywhere else, such as array[-3], array[6], or array[100] will cause your code to crash with an ArrayIndexOutOfBoundsException!

The Array's capacity must be decided when the Array is created. The capacity cannot be changed later. Going back to our DVD's-in-a-cardboard-box-analogy, changing the capacity of an Array would be akin to trying to make a cardboard box bigger. Trying to make a fixed-size cardboard box bigger is impractical, and it's the same as an Array on a computer!

So, what do we do if we get a 7th DVD and we'd like all our DVD's in the same Array? Well, unfortunately it's the same as it is with our cardboard box. We'll need to go get a larger one, and then move all the existing DVD's into it, along with the new one.

The **capacity** of an Array in Java can be checked by looking at the value of its length attribute. This is done using the code arr.length, where arr is the name of the Array. Different programming languages have different ways of checking the length of an Array.

int capacity = array.length;

System.out.println("The Array has a capacity of " + capacity);

Running this code will give the following output:

The Array has a capacity of 6

Yup, it's a bit confusing that you need to access the capacity of an Array by using .length. Unfortunately, this is just something you'll need to get used to.

### **Array Length**

The other definition of **length** is the number of DVDs, or other items, currently in the Array. This is something you'll need to keep track of yourself, and you won't get any errors if you overwrite an existing DVD, or if you leave a gap in the Array.

You might have noticed that we've been using a length variable in our previous examples, to keep track of the next empty index.

|  |
| --- |
| // Create a new array with a capacity of 6.  int[] array = new int[6];  // Current length is 0, because it has 0 elements.  int length = 0;  // Add 3 items into it.  for (int i = 0; i < 3; i++) {  array[i] = i \* i;  // Each time we add an element, the length goes up by one.  length++;  }  System.out.println("The Array has a capacity of " + array.length);  System.out.println("The Array has a length of " + length); |

Running this code will give the following output:

The Array has a capacity of 6

The Array has a length of 3

**Max Consecutive Ones**

Given a binary array, find the maximum number of consecutive 1s in this array.

**Example 1:**

**Input:** [1,1,0,1,1,1]

**Output:** 3

**Explanation:** The first two digits or the last three digits are consecutive 1s.

The maximum number of consecutive 1s is 3.

**Note:**

* The input array will only contain 0 and 1.
* The length of input array is a positive integer and will not exceed 10,000

   Hide Hint #1

You need to think about two things as far as any window is concerned. One is the starting point for the window. How do you detect that a new window of 1s has started? The next part is detecting the ending point for this window. How do you detect the ending point for an existing window? If you figure these two things out, you will be able to detect the windows of consecutive ones. All that remains afterward is to find the longest such window and return the size.

**Find Numbers with Even Number of Digits**

Given an array nums of integers, return how many of them contain an **even number** of digits.

**Example 1:**

**Input:** nums = [12,345,2,6,7896]

**Output:** 2

**Explanation:**

12 contains 2 digits (even number of digits).

345 contains 3 digits (odd number of digits).

2 contains 1 digit (odd number of digits).

6 contains 1 digit (odd number of digits).

7896 contains 4 digits (even number of digits).

Therefore only 12 and 7896 contain an even number of digits.

**Example 2:**

**Input:** nums = [555,901,482,1771]

**Output:** 1

**Explanation:**

Only 1771 contains an even number of digits.

**Constraints:**

* 1 <= nums.length <= 500
* 1 <= nums[i] <= 10^5

   Hide Hint #1

How to compute the number of digits of a number ?

   Hide Hint #2

Divide the number by 10 again and again to get the number of digits.

**Squares of a Sorted Array**

Given an integer array nums sorted in **non-decreasing** order, return *an array of****the squares of each number****sorted in non-decreasing order*.

**Example 1:**

**Input:** nums = [-4,-1,0,3,10]

**Output:** [0,1,9,16,100]

**Explanation:** After squaring, the array becomes [16,1,0,9,100].

After sorting, it becomes [0,1,9,16,100].

**Example 2:**

**Input:** nums = [-7,-3,2,3,11]

**Output:** [4,9,9,49,121]

**Constraints:**

* 1 <= nums.length <= 104
* -104 <= nums[i] <= 104
* nums is sorted in **non-decreasing** order.

**Follow up:** Squaring each element and sorting the new array is very trivial, could you find an O(n) solution using a different approach?

## Solution

#### **Approach 1: Sort**

**Intuition and Algorithm**

Create an array of the squares of each element, and sort them.

|  |
| --- |
| class Solution {  public int[] sortedSquares(int[] A) {  int N = A.length;  int[] ans = new int[N];  for (int i = 0; i < N; ++i)  ans[i] = A[i] \* A[i];  Arrays.sort(ans);  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*log*N*), where *N* is the length of A.
* Space complexity : O(*N*) or O(log*N*)
  + The space complexity of the sorting algorithm depends on the implementation of each program language.
  + For instance, the list.sort() function in Python is implemented with the [Timsort](https://en.wikipedia.org/wiki/Timsort) algorithm whose space complexity is O(*N*).
  + In Java, the [Arrays.sort()](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html" \l "sort-byte:A-) is implemented as a variant of quicksort algorithm whose space complexity is O(log*N*).

#### **Approach 2: Two Pointer**

**Intuition**

Since the array A is sorted, loosely speaking it has some negative elements with squares in decreasing order, then some non-negative elements with squares in increasing order.

For example, with [-3, -2, -1, 4, 5, 6], we have the negative part [-3, -2, -1] with squares [9, 4, 1], and the positive part [4, 5, 6] with squares [16, 25, 36]. Our strategy is to iterate over the negative part in reverse, and the positive part in the forward direction.

**Algorithm**

We can use two pointers to read the positive and negative parts of the array - one pointer j in the positive direction, and another i in the negative direction.

Now that we are reading two increasing arrays (the squares of the elements), we can merge these arrays together using a two-pointer technique.

|  |
| --- |
| class Solution {  public int[] sortedSquares(int[] nums) {  int n = nums.length;  int[] result = new int[n];  int left = 0;  int right = n - 1;  for (int i = n - 1; i >= 0; i--) {  int square;  if (Math.abs(nums[left]) < Math.abs(nums[right])) {  square = nums[right];  right--;  } else {  square = nums[left];  left++;  }  result[i] = square \* square;  }  return result;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*), where *N* is the length of A.
* Space Complexity: *O*(*N*) if you take output into account and *O*(1) otherwise.

## Inserting Items Into an Array

In the previous chapter, we looked at what Arrays are, and the basic programming constructs of Arrays in Java. We're now going to use these basic constructs to implement three key operations for Arrays:

* Inserting items.
* Removing items.
* Searching for items.

These three operations are the fundamental operations for all data structures.

In this chapter, we'll be starting with **inserting items into an array**. Like before, we'll approach the learning with lots of examples and programming snippets. After that, there are some more interview questions for you to practice on. We hope you have fun!

**Basic Array Operations**

Now that we have a fairly good understanding of what an Array actually is, and how it is stored inside the computer's physical memory, the next important thing to look at is all the operations that Arrays support. An Array is a data structure, which means that it stores data in a specific format and supports certain operations on the data it stores. Consider the DVD inventory management software from the introduction section. Let's look at some operations you might want to perform using this software:

* **Insert** a new DVD into the collection at a specific location.
* **Delete** a DVD from the existing collection if it doesn't make sense to keep it in the inventory anymore.
* **Search** for a particular DVD in the collection. This is one of the most commonly executed operation on our collection, because our inventory management software would be used hundreds of times a day to look for a particular DVD asked for by the user.

In this section, we'll be looking at the three basic operations that are supported by almost every data structure; **insertion**, **deletion**, and **search**.

**Array Insertions**

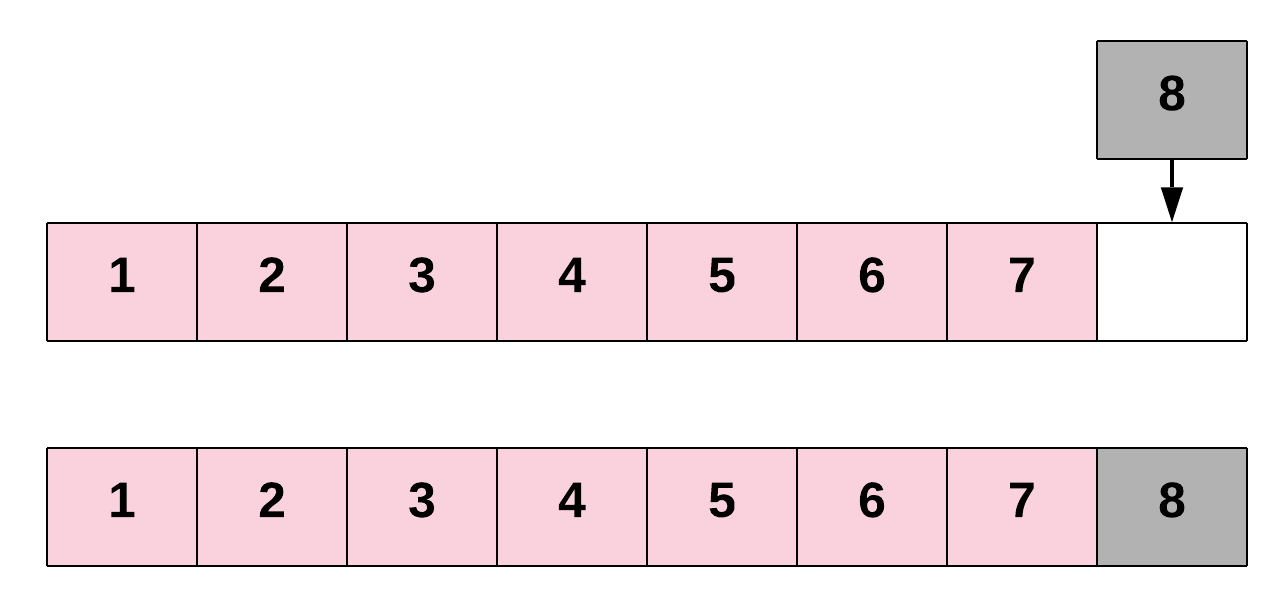
In the previous chapter, we looked at how to write elements to an Array. There is a lot more to inserting elements though, as we're about to see!

Inserting a new element into an Array can take many forms:

1. Inserting a new element at the end of the Array.
2. Inserting a new element at the beginning of the Array.
3. Inserting a new element at any given index inside the Array.

### **Inserting at the End of an Array**

At any point in time, we know the index of the last element of the Array, as we've kept track of it in our length variable. All we need to do for inserting an element at the end is to assign the new element to one index past the current last element.



This is pretty much the same as we've already seen. Here's the code to make a new Array that can hold up to 6 items, and then add items into the first 3 three indexes.

|  |
| --- |
| // Declare an integer array of 6 elements  int intArray = new int[6];  int length = 0;  // Add 3 elements to the Array  for (int i = 0; i < 3; i++) {  intArray[length] = i;  length++;  } |

Let's define a function, printArray, to help us visualise what's happening.

|  |
| --- |
| for (int i = 0; i < intArray.length; i++) {  System.out.println("Index " + i + " contains " + intArray[i]);  } |

If we run our printArray function, we'll get the following output.

Index 0 contains 0.

Index 1 contains 1.

Index 2 contains 2.

Index 3 contains 0.

Index 4 contains 0.

Index 5 contains 0.

Notice how indexes 3, 4, and 5 all contain 0? This is because Java fills unused int Array slots with 0s.

Let's now add a 4th element. We'll add the number 10.

|  |
| --- |
| // Insert a new element at the end of the Array. Again,  // it's important to ensure that there is enough space  // in the array for inserting a new element.  intArray[length] = 10;  length++; |

Notice how we also incremented the length? This is very important, next time when we add another element, we'll accidentally overwrite the one we just added!

Running printArray again, we'll get the following:

Index 0 contains 0.

Index 1 contains 1.

Index 2 contains 2.

Index 3 contains 10.

Index 4 contains 0.

Index 5 contains 0.

### **Inserting at the Start of an Array**

To insert an element at the start of an Array, we'll need to shift all other elements in the Array to the right by one index to create space for the new element. This is a very costly operation, since each of the existing elements has to be shifted one step to the right. The need to shift everything implies that this is not a constant time operation. In fact, the time taken for insertion at the beginning of an Array will be proportional to the length of the Array. In terms of time complexity analysis, this is a linear time complexity: *O*(*N*), where *N* is the length of the Array.

Diagram

Description automatically generated

Here's what this looks like in code.

|  |
| --- |
| // First, we will have to create space for a new element.  // We do that by shifting each element one index to the right.  // This will firstly move the element at index 3, then 2, then 1, then finally 0.  // We need to go backwards to avoid overwriting any elements.  for (int i = 3; i >= 0; i--) {  intArray[i + 1] = intArray[i];  }  // Now that we have created space for the new element,  // we can insert it at the beginning.  intArray[0] = 20; |

And here's the result of running printArray.

Index 0 contains 20.

Index 1 contains 0.

Index 2 contains 1.

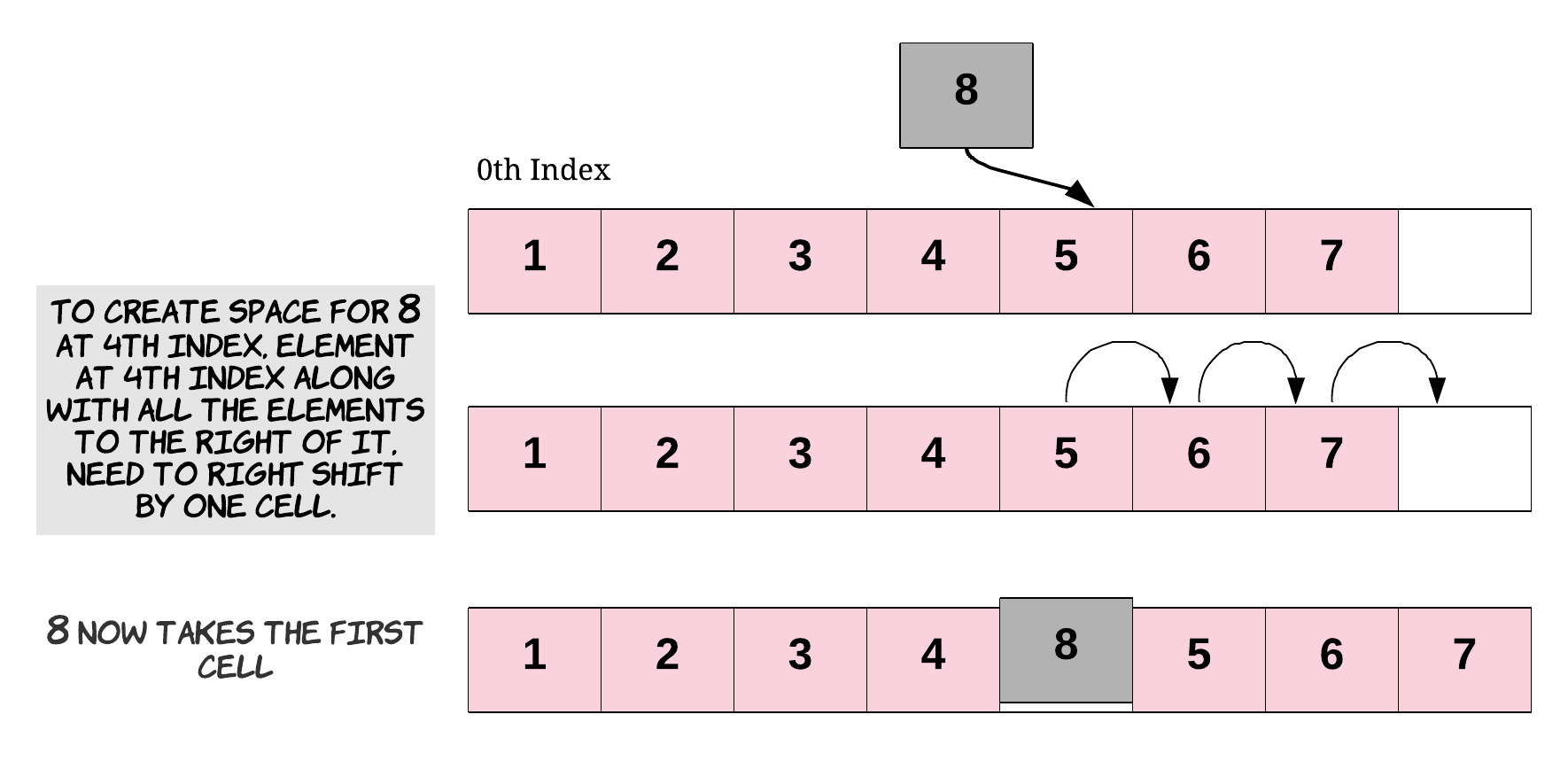
Index 3 contains 2.

Index 4 contains 10.

Index 5 contains 0.

### **Inserting Anywhere in the Array**

Similarly, for inserting at any given index, we first need to shift all the elements from that index onwards one position to the right. Once the space is created for the new element, we proceed with the insertion. If you think about it, insertion at the beginning is basically a special case of inserting an element at a given index—in that case, the given index was 0.



Again, this is also a costly operation since we could potentially have to shift almost all the other elements to the right before actually inserting the new element. As your saw above, shifting lots of elements one place to the right adds to the time complexity of the insertion task.

Here's what it looks like in code.

|  |
| --- |
| // Say we want to insert the element at index 2.  // First, we will have to create space for the new element.  for (int i = 4; i >= 2; i--)  {  // Shift each element one position to the right.  intArray[i + 1] = intArray[i];  }  // Now that we have created space for the new element,  // we can insert it at the required index.  intArray[2] = 30; |

And here's the result of running printArray.

Index 0 contains 20.

Index 1 contains 0.

Index 2 contains 30.

Index 3 contains 1.

Index 4 contains 2.

Index 5 contains 10.

Does that all sound good? The main thing to be careful of is remembering that array.length gives you the *total capacity* of the Array. If you want to know the last *used* slot, you'll need to keep track of this yourself using a length variable. Other than that, just be careful to read any elements you want to keep, before you overwrite them!

We now have a fun problem for you to test your understanding on. Enjoy!

**Duplicate Zeros**

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

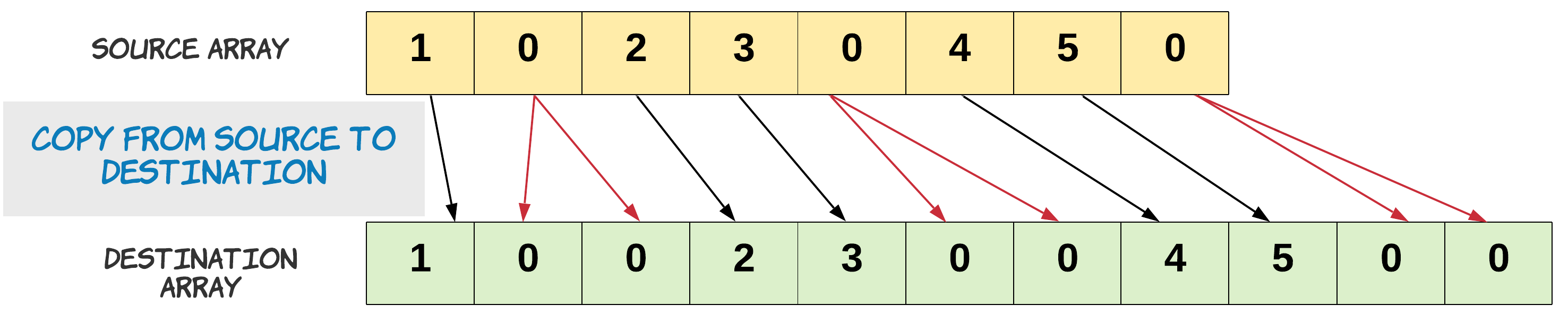
The main problem with not using additional memory is that we might override elements due to the zero duplication requirement of the problem statement. How do we get around that?

   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]

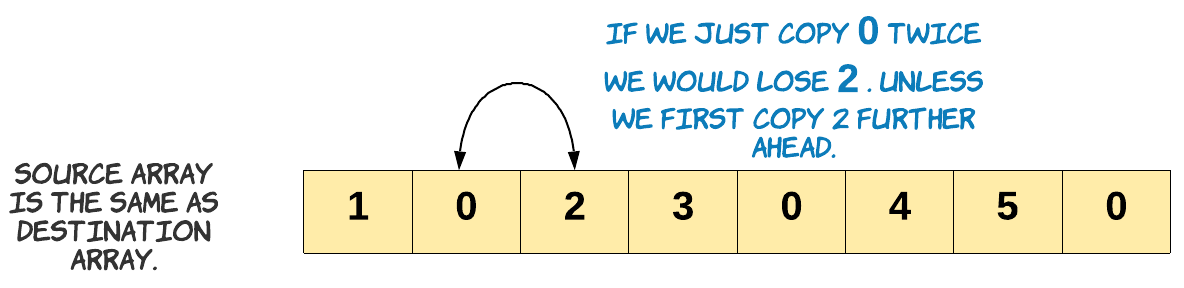
d += 1

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.

Diagram

Description automatically generated

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**

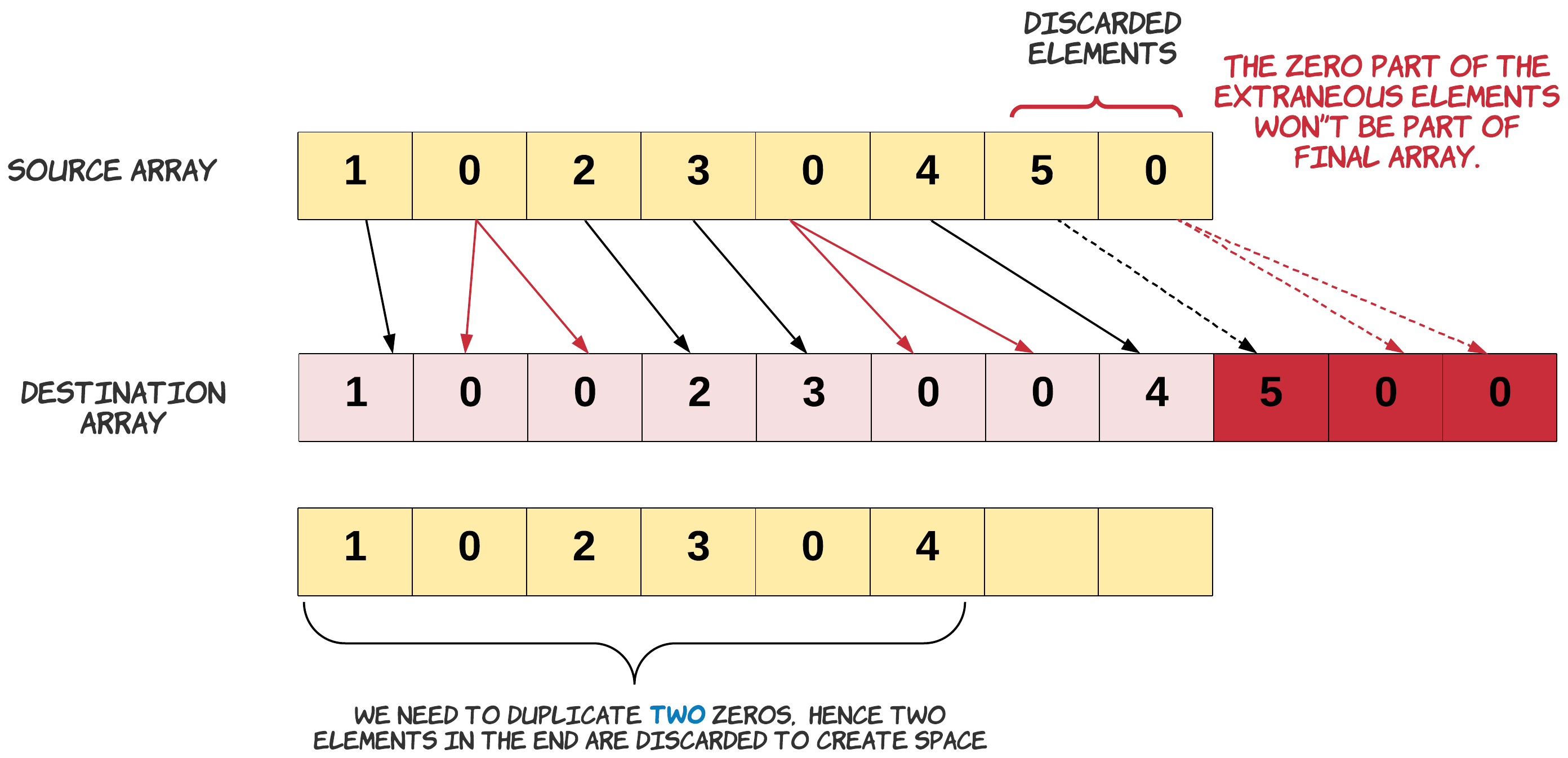
**Intuition**

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.



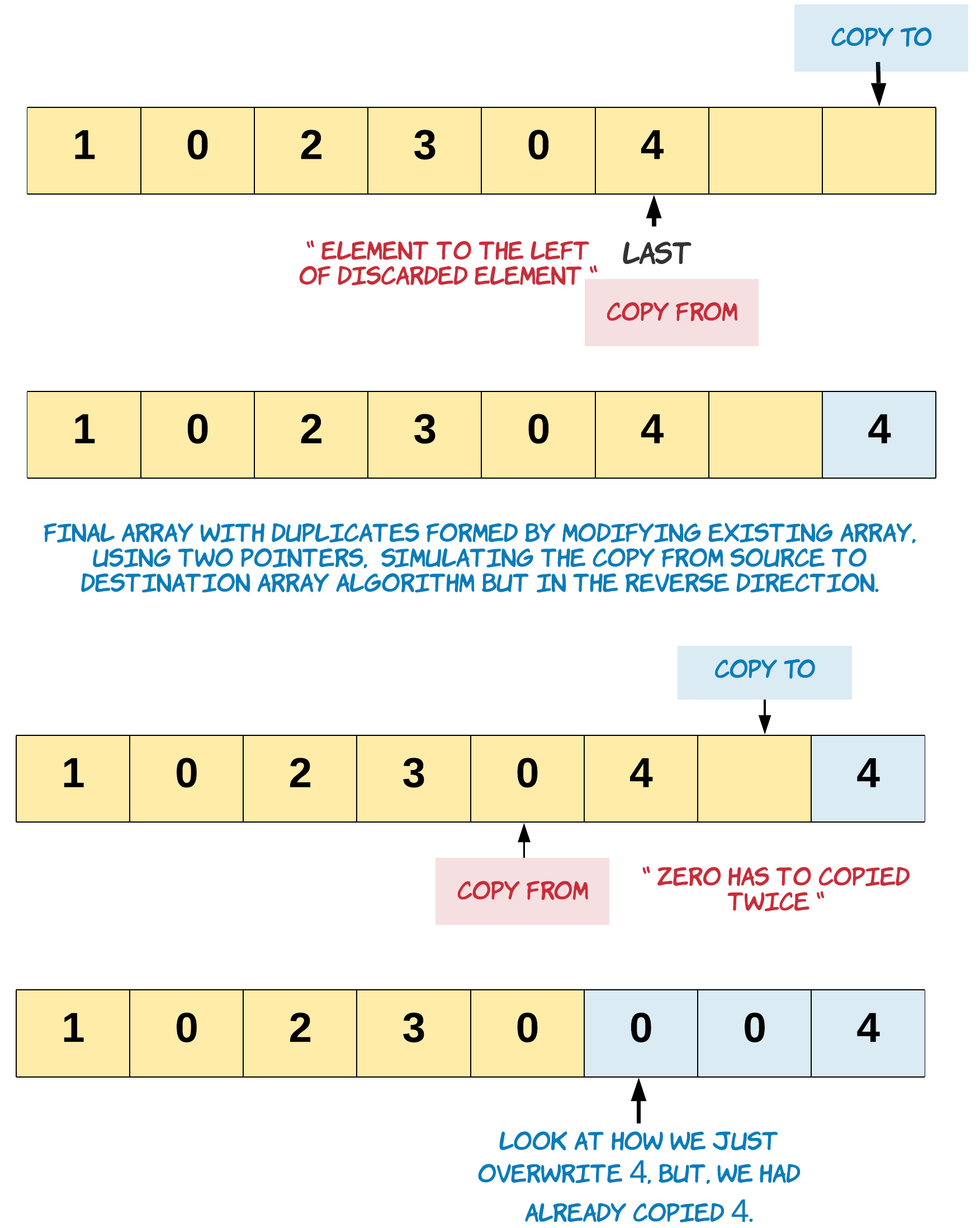
Note: In the diagram above we just show source and destination array for understanding purpose. We will be doing these operations only on one array.

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];  }  }  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*), where *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). We do not use any extra space.

**Merge Sorted Array**

Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

The number of elements initialized in nums1 and nums2 are m and n respectively. You may assume that nums1 has a size equal to m + n such that it has enough space to hold additional elements from nums2.

**Example 1:**

**Input:** nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3

**Output:** [1,2,2,3,5,6]

**Example 2:**

**Input:** nums1 = [1], m = 1, nums2 = [], n = 0

**Output:** [1]

**Constraints:**

* nums1.length == m + n
* nums2.length == n
* 0 <= m, n <= 200
* 1 <= m + n <= 200
* -109 <= nums1[i], nums2[i] <= 109

   Hide Hint #1

You can easily solve this problem if you simply think about two elements at a time rather than two arrays. We know that each of the individual arrays is sorted. What we don't know is how they will intertwine. Can we take a local decision and arrive at an optimal solution?

   Hide Hint #2

If you simply consider one element each at a time from the two arrays and make a decision and proceed accordingly, you will arrive at the optimal solution.

## Solution

#### **Approach 1 : Merge and sort**

**Intuition**

A naive approach would be to simply write the values from nums2 into the end of nums1, and then sort nums1. Remember that we do not need to return a value, as we should modify nums1 in-place. While straightforward to code, this approach has a high time complexity as we're not taking advantage of the existing sorting.

**Implementation**

|  |
| --- |
| class Solution {  public void merge(int[] nums1, int m, int[] nums2, int n) {  for (int i = 0; i < n; i++) {  nums1[i + m] = nums2[i];  }  Arrays.sort(nums1);  }  } |

* Time complexity : O((*n*+*m*)log(*n*+*m*)).

The cost of sorting a list of length x*x* using a built-in sorting algorithm is O(*x*log*x*). Because in this case we're sorting a list of length *m*+*n*, we get a total time complexity of O((*n*+*m*)log(*n*+*m*)).

* Space complexity : O(*n*), but it can vary.

Most programming languages have a built-in sorting algorithm that uses O(*n*) space.

#### **Approach 2 : Three Pointers (Start From the Beginning)**

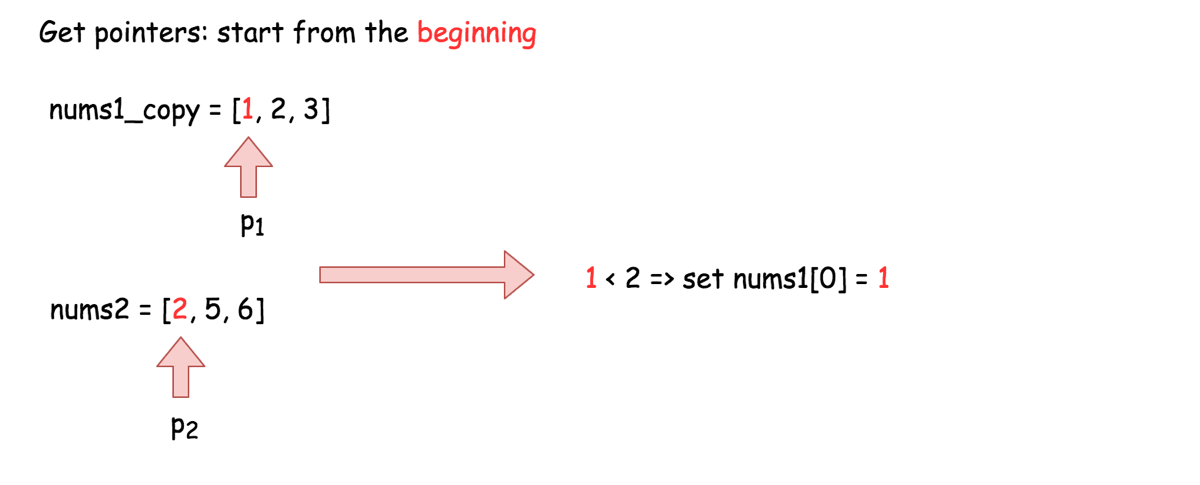
**Intuition**

Because each array is already sorted, we can achieve an \mathcal{O}(n + m)O(*n*+*m*) time complexity with the help of the two-pointer technique.

**Algorithm**

The simplest implementation would be to make a copy of the values in nums1, called nums1Copy, and then use two **read** pointers and one **write** pointer to read values from nums1Copy and nums2 and write them into nums1.

* Initialize nums1Copy to be a new array containing the first m values of nums1.
* Initialize read pointer p1 to the beginning of nums1Copy.
* Initialize read pointer p2 to the beginning of nums2.
* Initialize write pointer p to the beginning of nums1.
* While p is still within nums1:
  + If nums1Copy[p1] exists and is less than or equal to nums2[p2]:
    - Write nums1Copy[p1] into nums1[p], and increment p1 by 1.
  + Else
    - Write nums2[p2] into nums1[p], and increment p2 by 1.
  + Increment p by 1.



**Implementation**

|  |
| --- |
| class Solution {  public void merge(int[] nums1, int m, int[] nums2, int n) {  // Make a copy of the first m elements of nums1.  int[] nums1Copy = new int[m];  for (int i = 0; i < m; i++) {  nums1Copy[i] = nums1[i];  }  // Read pointers for nums1Copy and nums2 respectively.  int p1 = 0;  int p2 = 0;    // Compare elements from nums1Copy and nums2 and write the smallest to nums1.  for (int p = 0; p < m + n; p++) {  // We also need to ensure that p1 and p2 aren't over the boundaries  // of their respective arrays.  if (p2 >= n || (p1 < m && nums1Copy[p1] < nums2[p2])) {  nums1[p] = nums1Copy[p1++];  } else {  nums1[p] = nums2[p2++];  }  }  }  } |

**Complexity Analysis**

* Time complexity : O(*n*+*m*).

We are performing *n*+2⋅*m* reads and *n*+2⋅*m* writes. Because constants are ignored in Big O notation, this gives us a time complexity of O(*n*+*m*).

* Space complexity : O(*m*).

We are allocating an additional array of length m*m*.

#### **Approach 3 : Three Pointers (Start From the End)**

**Intuition**

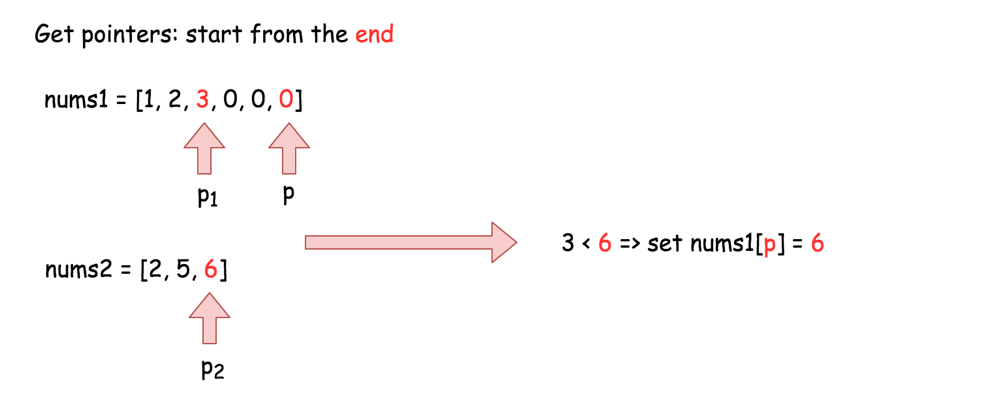
**Interview Tip**: This is a medium-level solution to an easy-level problem. Many of LeetCode's easy-level problems have more difficult solutions, and good candidates are expected to find them.

Approach 2 already demonstrates the best possible time complexity, O(*n*+*m*), but still uses additional space. This is because the elements of array nums1 have to be stored somwhere so that they aren't overwritten.

So, what if instead we start to overwrite nums1 from the end, where there is no information yet?

The algorithm is similar to before, except this time we set p1 to point at index m - 1 of nums1, p2 to point at index n - 1 of nums2, and p to point at index m + n - 1 of nums1. This way, it is guaranteed that once we start overwriting the first m values in nums1, we will have already written each into its new position. In this way, we can eliminate the additional space.

**Interview Tip**: Whenever you're trying to solve an array problem in-place, always consider the possibility of iterating backwards instead of forwards through the array. It can completely change the problem, and make it a lot easier.



**Implementation**

Graphical user interface, text, application

Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated with medium confidence

Graphical user interface, application

Description automatically generated with medium confidence

Graphical user interface

Description automatically generated with low confidence

Text

Description automatically generated with low confidence

Graphical user interface, text, application

Description automatically generated

|  |
| --- |
| class Solution {  public void merge(int[] nums1, int m, int[] nums2, int n) {  // Set p1 and p2 to point to the end of their respective arrays.  int p1 = m - 1;  int p2 = n - 1;    // And move p backwards through the array, each time writing  // the smallest value pointed at by p1 or p2.  for (int p = m + n - 1; p >= 0; p--) {  if (p2 < 0) {  break;  }  if (p1 >= 0 && nums1[p1] > nums2[p2]) {  nums1[p] = nums1[p1--];  } else {  nums1[p] = nums2[p2--];  }  }  }  } |

**Complexity Analysis**

* Time complexity : O(*n*+*m*).

Same as Approach 2.

* Space complexity : O(1).

Unlike Approach 2, we're not using an extra array.

**Proof (optional)**

You might be a bit sceptical of this claim. Does it really work in every case? Is it safe to be making such a bold claim?

This way, it is guaranteed that once we start overwriting the first m values in nums1, we will have already written each into its new position. In this way, we can eliminate the additional space.

Great question! So, why does this work? To prove it, we need to ensure that p never overwrites a value in nums1 that p1 hasn't yet read from nums1.

**Words of Advice**: Terrified of proofs? Many software engineers are. Good proofs are simply a series of logical assertions, each building on the next. In this way, we can go from "obvious" statements, all the way to the one we want to prove. I recommend reading each statement one-by-one, ensuring that you understand each before moving onto the next.

1. *We know that* upon initialization, p is n steps ahead of p1 (in other words, p1 + n = p).
2. *We also know that* during each of the p iterations this algorithm performs, p is always decremented by 1, and *either* p1 *or* p2 is decremented by 1.
3. *We can deduce that* when p1 decremented, the gap between p and p1 stays the same, so there can't be an "overtake" in that case.
4. *We can also deduce that* when p2 is decremented though, the gap between p and p1 shrinks by 1 as p moves, but not p1.
5. *And from that, we can deduce that* the maximum number of times that p2 can be decremented is n. In other words, the gap between p and p1 can shrink by 1, at most n times.
6. *In conclusion*, it's impossible for an overtake to occur, as they started n apart. And when p = p1, the gap has to have shrunk n times. This means that all of nums2 have been merged in, and so there is nothing more to do.

Deleting Items From an Array

Next on the agenda is insertion's complement—deletion.

**Array Deletions**

Now that we know how insertion works, it's time to look at its complement—deletion!

Deletion in an Array works in a very similar manner to insertion, and has the same three different cases:

1. Deleting the last element of the Array.
2. Deleting the first element of the Array.
3. Deletion at any given index.

### **Deleting From the End of an Array**

Deletion at the end of an Array is similar to people standing in a line, also known as a queue. The person who most recently joined the queue (at the end) can leave at any time without disturbing the rest of the queue. Deleting from the end of an Array is the least time consuming of the three cases. Recall that insertion at the end of an Array was also the least time-consuming case for insertion.

Table

Description automatically generated

So, how does this work in code? Before we look at this, let's quickly remind ourselves what the length of an Array means. Here is some code that creates an Array with room for 10 elements, and then adds elements into the first 6 indexes of it.

|  |
| --- |
| // Declare an integer array of 10 elements.  int[] intArray = new int[10];  // The array currently contains 0 elements  int length = 0;  // Add elements at the first 6 indexes of the Array.  for(int i = 0; i < 6; i++) {  intArray[length] = i;  length++;  } |

Notice the length variable. Essentially, this variable keeps track of the next index that is free for inserting a new element. This is always the same value as the overall length of the Array. Note that when we declare an Array of a certain size, we simply fix the maximum number of elements it could contain. Initially, the Array doesn't contain anything. Thus, when we add new elements, we also increment the length variable accordingly.

Anyway, here's the code for deleting the last element of an Array.

|  |
| --- |
| // Deletion from the end is as simple as reducing the length  // of the array by 1.  length--; |

Remember how insertion we were using this printArray function?

|  |
| --- |
| for (int i = 0; i < intArray.length; i++) {  System.out.println("Index " + i + " contains " + intArray[i]);  } |

Well, if we run it here, we'll get the following result, regardless of whether we run it before or after removing the last element.

Index 0 contains 0.

Index 1 contains 1.

Index 2 contains 2.

Index 3 contains 3.

Index 4 contains 4.

Index 5 contains 5.

Index 6 contains 0.

Index 7 contains 0.

Index 8 contains 0.

Index 9 contains 0.

What's gone wrong? Well, remember how there's two different definitions of length? When we use intArray.length, we're looking every valid index of the Array. When in fact, we only want to look at the ones that we've put values into. The fix is easy, we just iterate up to our own length variable instead.

|  |
| --- |
| for (int i = 0; i < length; i++) {  System.out.println("Index " + i + " contains " + intArray[i]);  } |

Run this, and you'll get the following before the deletion:

Index 0 contains 0.

Index 1 contains 1.

Index 2 contains 2.

Index 3 contains 3.

Index 4 contains 4.

Index 5 contains 5.

And the following after:

Index 0 contains 0.

Index 1 contains 1.

Index 2 contains 2.

Index 3 contains 3.

Index 4 contains 4.

Yup, that's it! Even though we call it a deletion, its not like we actually freed up the space for a new element, right? This is because we don't actually need to free up any space. Simply overwriting the value at a certain index deletes the element at that index. Seeing as the length variable in our examples tells us the next index where we can insert a new element, reducing it by one ensures the next new element is written over the deleted one. This also indicates that the Array now contains one less element, which is exactly what we want programmatically.

### **Deleting From the Start of an Array**

Next comes the costliest of all deletion operations for an Array—deleting the first element. If we want to delete the first element of the Array, that will create a vacant spot at the 0th index. To fill that spot, we will shift the element at index 1 one step to the left. Going by the ripple effect, every element all the way to the last one will be shifted one place to the left. This shift of elements takes *O*(*N*) time, where *N* is the number of elements in the Array.

Table

Description automatically generated

Here is how deleting the first element looks in code.

|  |
| --- |
| // Starting at index 1, we shift each element one position  // to the left.  for (int i = 1; i < length; i++) {  // Shift each element one position to the left  int\_array[i - 1] = int\_array[i];  }  // Note that it's important to reduce the length of the array by 1.  // Otherwise, we'll lose consistency of the size. This length  // variable is the only thing controlling where new elements might  // get added.  length--; |

Starting from index 0, we'll move every element one position to its left, effectively "deleting" the element at index 0. We also need to reduce length by 1 so that the next new element is inserted in the correct position.

And here's the output we'll get, with our updated printArray function.

Index 0 contains 1.

Index 1 contains 2.

Index 2 contains 3.

Index 3 contains 4.

### **Deleting From Anywhere in the Array**

For deletion at any given index, the empty space created by the deleted item will need to be filled. Each of the elements to the right of the index we're deleting at will get shifted to the left by one. Deleting the first element of an Array is a special case of deletion at a given index, where the index is 0. This shift of elements takes *O*(*K*) time where *K* is the number of elements to the right of the given index. Since potentially *K*=*N*, we say that the time complexity of this operation is also *O*(*N*).

Table

Description automatically generated

Here is the code to delete the element at index 1. To do this, we'll need to move over the elements after it in the Array.

|  |
| --- |
| // Say we want to delete the element at index 1  for (int i = 2; i < length; i++) {  // Shift each element one position to the left  int\_array[i - 1] = int\_array[i];  }  // Again, the length needs to be consistent with the current  // state of the array.  length--; |

Notice that this works exactly like deleting the first element, except that we don't touch the elements that are at *lower* indexes than the element we're deleting.

Here is the output from the printArray function.

Index 0 contains 1.

Index 1 contains 3.

Index 2 contains 4.

Did that all make sense? To help you cement what you've learned, here's a couple of programming problems for you to try. You should try to solve them *without* making a new Array. Do this by using the deletion techniques we've investigated above.

Once you're done, we'll look at searching Arrays!

**Remove Element**

Given an array nums and a value val, remove all instances of that value [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) and return the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

**Clarification:**

Confused why the returned value is an integer but your answer is an array?

Note that the input array is passed in by **reference**, which means a modification to the input array will be known to the caller as well.

Internally you can think of this:

// **nums** is passed in by reference. (i.e., without making a copy)

int len = removeElement(nums, val);

// any modification to **nums** in your function would be known by the caller.

// using the length returned by your function, it prints the first **len** elements.

for (int i = 0; i < len; i++) {

    print(nums[i]);

}

**Example 1:**

**Input:** nums = [3,2,2,3], val = 3

**Output:** 2, nums = [2,2]

**Explanation:** Your function should return length = **2**, with the first two elements of nums being **2**.

It doesn't matter what you leave beyond the returned length. For example if you return 2 with nums = [2,2,3,3] or nums = [2,2,0,0], your answer will be accepted.

**Example 2:**

**Input:** nums = [0,1,2,2,3,0,4,2], val = 2

**Output:** 5, nums = [0,1,4,0,3]

**Explanation:** Your function should return length = **5**, with the first five elements of *nums* containing **0**, **1**, **3**, **0**, and **4**. Note that the order of those five elements can be arbitrary. It doesn't matter what values are set beyond the returned length.

**Constraints:**

* 0 <= nums.length <= 100
* 0 <= nums[i] <= 50
* 0 <= val <= 100

   Hide Hint #1

The problem statement clearly asks us to modify the array in-place and it also says that the element beyond the new length of the array can be anything. Given an element, we need to remove all the occurrences of it from the array. We don't technically need to **remove** that element per-say, right?

   Hide Hint #2

We can move all the occurrences of this element to the end of the array. Use two pointers!  
Diagram

Description automatically generated with medium confidence

   Hide Hint #3

Yet another direction of thought is to consider the elements to be removed as non-existent. In a single pass, if we keep copying the visible elements in-place, that should also solve this problem for us.

**Remove Duplicates from Sorted Array**

Given a sorted array nums, remove the duplicates [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) such that each element appears only once and returns the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

**Clarification:**

Confused why the returned value is an integer but your answer is an array?

Note that the input array is passed in by **reference**, which means a modification to the input array will be known to the caller as well.

Internally you can think of this:

// **nums** is passed in by reference. (i.e., without making a copy)

int len = removeDuplicates(nums);

// any modification to **nums** in your function would be known by the caller.

// using the length returned by your function, it prints the first **len** elements.

for (int i = 0; i < len; i++) {

    print(nums[i]);

}

**Example 1:**

**Input:** nums = [1,1,2]

**Output:** 2, nums = [1,2]

**Explanation:** Your function should return length = **2**, with the first two elements of *nums* being **1** and **2** respectively. It doesn't matter what you leave beyond the returned length.

**Example 2:**

**Input:** nums = [0,0,1,1,1,2,2,3,3,4]

**Output:** 5, nums = [0,1,2,3,4]

**Explanation:** Your function should return length = **5**, with the first five elements of *nums* being modified to **0**, **1**, **2**, **3**, and **4** respectively. It doesn't matter what values are set beyond the returned length.

**Constraints:**

* 0 <= nums.length <= 3 \* 104
* -104 <= nums[i] <= 104
* nums is sorted in ascending order.

   Hide Hint #1

In this problem, the key point to focus on is the input array being sorted. As far as duplicate elements are concerned, what is their positioning in the array when the given array is sorted? Look at the image above for the answer. If we know the position of one of the elements, do we also know the positioning of all the duplicate elements?  
Chart, waterfall chart

Description automatically generated

   Hide Hint #2

We need to modify the array in-place and the size of the final array would potentially be smaller than the size of the input array. So, we ought to use a two-pointer approach here. One, that would keep track of the current element in the original array and another one for just the unique elements.

   Hide Hint #3

Essentially, once an element is encountered, you simply need to **bypass** its duplicates and move on to the next unique element.

Searching for Items in an Array

Finally, we're going to look at the linear search algorithm—the most basic and versatile array search algorithm.

**Search in an Array**

Finally, we're going to look at the most important operation of all. More often than not, it comes down to the speed of searching for an element in a data structure that helps programmers make design choices for their codebases.

There's more than one way of searching an Array, but for now, we're going to focus on the simplest way. Searching means to find an occurrence of a particular element in the Array and return its position. We might need to search an Array to find out whether or not an element is present in the Array. We might also want to search an Array that is arranged in a specific fashion to determine which index to insert a new element at.

If we know the index in the Array that may contain the element we're looking for, then the search becomes a constant time operation—we simply go to the given index and check whether or not the element is there.

### **Linear Search**

If the index is not known, which is the case most of the time, then we can check every element in the Array. We continue checking elements until we find the element we're looking for, or we reach the end of the Array. This technique for finding an element by checking through all elements one by one is known as the **linear search** algorithm. In the worst case, a linear search ends up checking the entire Array. Therefore, the time complexity for a linear search is *O*(*N*).

Diagram

Description automatically generated

Let's see the linear search algorithm in action, with all the edge cases handled properly. When we say edge cases, we basically mean scenarios that you wouldn't expect to encounter. For example, the element you're searching for might not even exist in the Array. Or, an even rarer, but possible, scenario is that the input Array doesn't contain any elements at all, or perhaps it is null. It's important to handle all of these edge cases within the code.

|  |
| --- |
| public static boolean linearSearch(int[] array, int length, int element) {  // Check for edge cases. Is the array null or empty?  // If it is, then we return false because the element we're  // searching for couldn't possibly be in it.  if (array == null || length == 0) {  return false;  }  // Carry out the linear search by checking each element,  // starting from the first one.  for (int i = 0; i < length; i++) {  // We found the element at index i.  // So we return true to say it exists.  if (array[i] == element) {  return true;  }  }  // We didn't find the element in the array.  return false;  } |

That's the function we can call to determine whether or not a particular element is in an Array. Notice how we take care of the edge cases before proceeding with the actual search, and that we don't check the rest of the elements once we'd found the element we were looking for.

There are many variations to this algorithm, such as returning the first location, last location, or all the locations (an element could be in the Array more than once). Let's see what happens when we call the linearSearch function.

|  |
| --- |
| public class ArraySearch {  public static void main(String args[]) {  // Declare a new array of 6 elements  int[] array = new int[6];  // Variable to keep track of the length of the array  int length = 0;  // Iterate through the 6 indexes of the Array.  for (int i = 0; i < 6; i++) {  // Add a new element and increment the length as well  array[length++] = i;  }  // Print out the results of searching for 4 and 30.  System.out.println("Does the array contain the element 4? - " + ArraySearch.linearSearch(array, length, 4));  System.out.println("Does the array contain the element 30? - " + ArraySearch.linearSearch(array, length, 30));  // Does the array contain the element 4? - true  // Does the array contain the element 30? - false  }  public static boolean linearSearch(int[] array, int length, int element) {  // Check for edge cases  if (array == null || length == 0) {  return false;  }  // Check each element starting from the first one  for (int i = 0; i < length; i++) {  // We found the element at index i, so return true.  if (array[i] == element) {  return true;  }  }  // We didn't find the element in the array.  return false;  }  } |

As expected, we're able to find the element 4 in the Array, but not 30.

### **Binary Search**

This section is optional. It briefly introduces a more advanced searching algorithm that you will learn more about in a later Explore Card.

There is another way of searching an Array. If the elements in the Array are in sorted order, then we can use binary search. Binary search is where we repeatedly look at the middle element in the Array, and determine whether the element we're looking for must be to the left, or to the right. Each time we do this, we're able to halve the number of elements we still need to search, making binary search a lot faster than linear search!

The downside of binary search though is that it only works if the data is sorted. If we only need to perform a single search, then it's faster to just do a linear search, as it takes longer to sort than to linear search. If we're going to be performing a lot of searches, it is often worth sorting the data first so that we can use binary search for the repeated searches.

You can find out more about binary search on our [Binary Search Explore Card](https://leetcode.com/explore/learn/card/binary-search/). For Arrays 101, it is okay for you to use either linear search or binary search.

Hopefully, the three basic Array operations are clear now! Like always, there are a couple of problems for you to try for yourself now.

After that, we'll be having a look at In-Place Array Operations. What are those, you might be asking? Let's not get ahead of ourselves though—you'll find out soon!

**Check If N and Its Double Exist**

Given an array arr of integers, check if there exists two integers N and M such that N is the double of M ( i.e. N = 2 \* M).

More formally check if there exists two indices i and j such that :

* i != j
* 0 <= i, j < arr.length
* arr[i] == 2 \* arr[j]

**Example 1:**

**Input:** arr = [10,2,5,3]

**Output:** true

**Explanation:** N = 10 is the double of M = 5,that is, 10 = 2 \* 5.

**Example 2:**

**Input:** arr = [7,1,14,11]

**Output:** true

**Explanation:** N = 14 is the double of M = 7,that is, 14 = 2 \* 7.

**Example 3:**

**Input:** arr = [3,1,7,11]

**Output:** false

**Explanation:** In this case does not exist N and M, such that N = 2 \* M.

**Constraints:**

* 2 <= arr.length <= 500
* -10^3 <= arr[i] <= 10^3

Hide Hint #1    
Loop from i = 0 to arr.length, maintaining in a hashTable the array elements from [0, i - 1].

Hide Hint #2    
On each step of the loop check if we have seen the element 2 \* arr[i] so far or arr[i] / 2 was seen if arr[i] % 2 == 0.

**Valid Mountain Array**

Given an array of integers arr, return *true if and only if it is a valid mountain array*.

Recall that arr is a mountain array if and only if:

* arr.length >= 3
* There exists some i with 0 < i < arr.length - 1 such that:
  + arr[0] < arr[1] < ... < arr[i - 1] < arr[i]
  + arr[i] > arr[i + 1] > ... > arr[arr.length - 1]

A picture containing line chart

Description automatically generated

**Example 1:**

**Input:** arr = [2,1]

**Output:** false

**Example 2:**

**Input:** arr = [3,5,5]

**Output:** false

**Example 3:**

**Input:** arr = [0,3,2,1]

**Output:** true

**Constraints:**

* 1 <= arr.length <= 104
* 0 <= arr[i] <= 104

Hide Hint #1

It's very easy to keep track of a monotonically increasing or decreasing ordering of elements. You just need to be able to determine the start of the valley in the mountain and from that point onwards, it should be a valley i.e. no mini-hills after that. Use this information in regards to the values in the array and you will be able to come up with a straightforward solution.

## Solution

#### **Approach 1: One Pass**

**Intuition**

If we walk along the mountain from left to right, we have to move strictly up, then strictly down.

**Algorithm**

Let's walk up from left to right until we can't: that has to be the peak. We should ensure the peak is not the first or last element. Then, we walk down. If we reach the end, the array is valid, otherwise its not.

|  |
| --- |
| class Solution {  public boolean validMountainArray(int[] A) {  int N = A.length;  int i = 0;  // walk up  while (i+1 < N && A[i] < A[i+1])  i++;  // peak can't be first or last  if (i == 0 || i == N-1)  return false;  // walk down  while (i+1 < N && A[i] > A[i+1])  i++;  return i == N-1;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*), where *N* is the length of A.  
  Space Complexity: *O*(1).

## In-Place Operations

Now that we've got all the basic Array operations out of the way, we're going to have a look at **in-place Array operations**. These are very important from an interviewing standpoint.

In-place Array operations are where we modify an Array, *without creating a new Array*. We've already seen some examples of this: inserting and removing items by shifting existing items along. There are many other situations where they come up though, so we will look at a few of them now.

**In-Place Array Operations Introduction**

In programming interviews, the interviewer often expects you to minimise the time and space complexity of your implementation. In-place Array operations help to reduce space complexity, and so are a class of techniques that pretty much everybody encounters regularly in interviews.

So, what *are* in-place array operations?

The best way of answering this question is to look at an example.

Given an Array of integers, return an Array where every element at an even-indexed position is squared.

**Input:** array = [9, -2, -9, 11, 56, -12, -3]

**Output:** [81, -2, 81, 11, 3136, -12, 9]

**Explanation:** The numbers at even indexes (0, 2, 4, 6) have been squared,

whereas the numbers at odd indexes (1, 3, 5) have been left the same.

This problem is hopefully very straightforward. Have a quick think about how you would implement it as an algorithm though, possibly jotting down some code on a piece of paper.

Anyway, there are two ways we could approach it. The first is to create a new Array, of the same size as the original. Then, we should copy the odd-indexed elements and square the even-indexed elements, writing them into the new Array.

|  |
| --- |
| public int[] squareEven(int[] array, int length) {  // Check for edge cases.  if (array == null) {  return null;  }  // Create a resultant Array which would hold the result.  int result[] = new int[length];  // Iterate through the original Array.  for(int i = 0; i < length; i++) {  // Get the element from slot i of the input Array.  int element = array[i];  // If the index is an even number, then we need to square element.  if (i % 2 == 0) {  element \*= element;  }  // Write element into the result Array.  result[i] = element;  }  // Return the result Array.  return result;  } |

Timeline

Description automatically generated

The above approach, although correct, is an inefficient way of solving the problem. This is because it uses *O*(length) extra space.

Instead, we could iterate over the original input Array itself, overwriting every even-indexed element with its own square. This way, we won't need that extra space. It is this technique of working directly in the input Array, and not creating a new Array, that we call **in-place**. In-place Array operations are a big deal for programming interviews, where there is a big focus on minimising both time and space complexity of algorithms.

Here's the in-place implementation for our squareEven(...) function.

|  |
| --- |
| public int[] squareEven(int[] array, int length) {  // Check for edge cases.  if (array == null) {  return array;  }  // Iterate through the original array.  for(int i = 0; i < length; i++) {  // If the index is an even number, then we need to square the element  // and replace the original value in the Array.  if (i % 2 == 0) {  array[i] \*= array[i];  }  // Notice how we don't need to do \*anything\* for the odd indexes? :-)  }  // We just return the original array. Some problems on leetcode require you  // to return it, and other's dont.  return array;  } |

Here's an animation showing the algorithm!

Table

Description automatically generated

Table

Description automatically generated

A picture containing diagram

Description automatically generated

Chart, waterfall chart

Description automatically generated

Chart, waterfall chart

Description automatically generated

Chart, waterfall chart

Description automatically generated

Chart, waterfall chart

Description automatically generated

A picture containing diagram

Description automatically generated

An important difference for in-place vs not in-place is that in-place *modifies the input Array*. This means that other functions *can no longer access the original data*, because it has been overwritten. We'll talk more about this in a bit.

We now have a couple of straightforward in-place problems for you to try. Remember, you aren't allowed to create any new Arrays (or any other data structures). If the return type of the question is an Array, then simply return the input Array once you've modified it.

**Replace Elements with Greatest Element on Right Side**

Given an array arr, replace every element in that array with the greatest element among the elements to its right, and replace the last element with -1.

After doing so, return the array.

**Example 1:**

**Input:** arr = [17,18,5,4,6,1]

**Output:** [18,6,6,6,1,-1]

**Explanation:**

- index 0 --> the greatest element to the right of index 0 is index 1 (18).

- index 1 --> the greatest element to the right of index 1 is index 4 (6).

- index 2 --> the greatest element to the right of index 2 is index 4 (6).

- index 3 --> the greatest element to the right of index 3 is index 4 (6).

- index 4 --> the greatest element to the right of index 4 is index 5 (1).

- index 5 --> there are no elements to the right of index 5, so we put -1.

**Example 2:**

**Input:** arr = [400]

**Output:** [-1]

**Explanation:** There are no elements to the right of index 0.

**Constraints:**

* 1 <= arr.length <= 104
* 1 <= arr[i] <= 105

Hide Hint #1

Loop through the array starting from the end.

   Hide Hint #2

Keep the maximum value seen so far.

**A Better Repeated Deletion Algorithm**

Let's look at one more example. This time, the result Array is smaller than the input Array! How's this going to work? Let's find out! Here's the problem description:

Given a sorted array, remove the duplicates such that each element appears only once.

**Input:** array = [1, 1, 2]

**Output:** [1, 2]

**Input:** array = [0, 0, 1, 1, 1, 2, 2, 3, 3, 4]

**Output:** [0, 1, 2, 3, 4]

You've hopefully already done this question, back when we were looking at deleting items from an Array. In that case, your algorithm might have looked something like this.

|  |
| --- |
| class Solution {  public int removeDuplicates(int[] nums) {    // The initial length is simply the capacity.  int length = nums.length;    // Assume the last element is always unique.  // Then for each element, delete it iff it is  // the same as the one after it. Use our deletion  // algorithm for deleting from any index.  for (int i = length - 2; i >= 0; i--) {  if (nums[i] == nums[i + 1]) {  // Delete the element at index i, using our standard  // deletion algorithm we learned.  for (int j = i + 1; j < length; j++) {  nums[j - 1] = nums[j];  }  // Reduce the length by 1.  length--;  }  }  // Return the new length.  return length;  }  } |

This is actually an in-place algorithm, because it doesn't require any extra space—its space complexity is *O*(1). However, the time complexity's not so flash, at O(N2). This is because of the nested loop.

We want to get the algorithm down to an *O*(*N*) time complexity.

If we *don't* try to do this in-place, then it's straightforward. We could simply iterate through the Array, adding all *unique* elements to a new Array. Seeing as the the input Array is sorted, we can easily identify all unique elements, as they are the first element, and then any element that is *different* to the one before it.

Diagram

Description automatically generated

One potential problem is that we actually don't know how long the result Array needs to be. Remember how that must be decided when the Array is created? The best solution for this problem is to do an initial pass, counting the number of unique elements. Then, we can create the result Array and do a second pass to add the elements into it. Here's the code for this approach.

|  |
| --- |
| public int[] copyWithRemovedDuplicates(int[] nums) {    // Check for edge cases.  if (nums == null || nums.length == 0) {  return nums;  }  // Count how many unique elements are in the Array.  int uniqueNumbers = 0;  for (int i = 0; i < nums.length; i++) {  // An element should be counted as unique if it's the first  // element in the Array, or is different to the one before it.  if (i == 0 || nums[i] != nums[i - 1]) {  uniqueNumbers++;  }  }  // Create a result Array.  int[] result = new int[uniqueNumbers];  // Write the unique elements into the result Array.  int positionInResult = 0;  for (int i = 0; i < nums.length; i++) {  // Same condition as in the previous loop. Except this time, we can write  // each unique number into the result Array instead of just counting them.  if (i == 0 || nums[i] != nums[i - 1]) {  result[positionInResult] = nums[i];  positionInResult++;  }  }  return result;  } |

Did you notice the fatal flaw with this approach though? It's the wrong return type! We could copy the result array back into the input array... and then return the length... but this is not what the question wants us to do. We want to instead do the deletions with a space complexity of *O*(1) and a time complexity of *O*(*N*).

Have a go at this for yourself, and then we'll talk about the solution. Your algorithm must be in-place, and take no more than *O*(*N*) time. Good luck!

**Remove Duplicates from Sorted Array**

Given a sorted array nums, remove the duplicates [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) such that each element appears only once and returns the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

**Clarification:**

Confused why the returned value is an integer but your answer is an array?

Note that the input array is passed in by **reference**, which means a modification to the input array will be known to the caller as well.

Internally you can think of this:

// **nums** is passed in by reference. (i.e., without making a copy)

int len = removeDuplicates(nums);

// any modification to **nums** in your function would be known by the caller.

// using the length returned by your function, it prints the first **len** elements.

for (int i = 0; i < len; i++) {

    print(nums[i]);

}

**Example 1:**

**Input:** nums = [1,1,2]

**Output:** 2, nums = [1,2]

**Explanation:** Your function should return length = **2**, with the first two elements of *nums* being **1** and **2** respectively. It doesn't matter what you leave beyond the returned length.

**Example 2:**

**Input:** nums = [0,0,1,1,1,2,2,3,3,4]

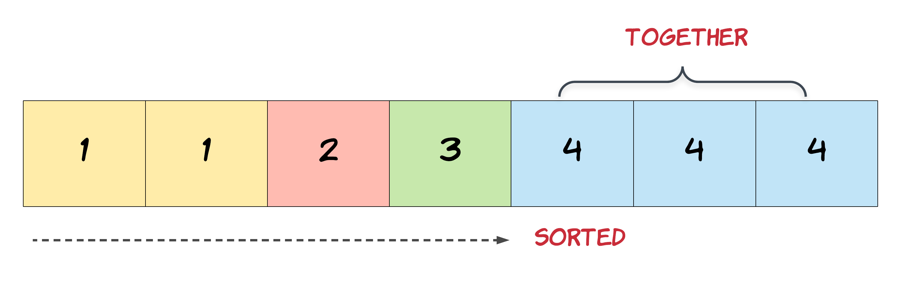
**Output:** 5, nums = [0,1,2,3,4]

**Explanation:** Your function should return length = **5**, with the first five elements of *nums* being modified to **0**, **1**, **2**, **3**, and **4** respectively. It doesn't matter what values are set beyond the returned length.

**Constraints:**

* 0 <= nums.length <= 3 \* 104
* -104 <= nums[i] <= 104
* nums is sorted in ascending order.

   Hide Hint #1

In this problem, the key point to focus on is the input array being sorted. As far as duplicate elements are concerned, what is their positioning in the array when the given array is sorted? Look at the image above for the answer. If we know the position of one of the elements, do we also know the positioning of all the duplicate elements?  


   Hide Hint #2

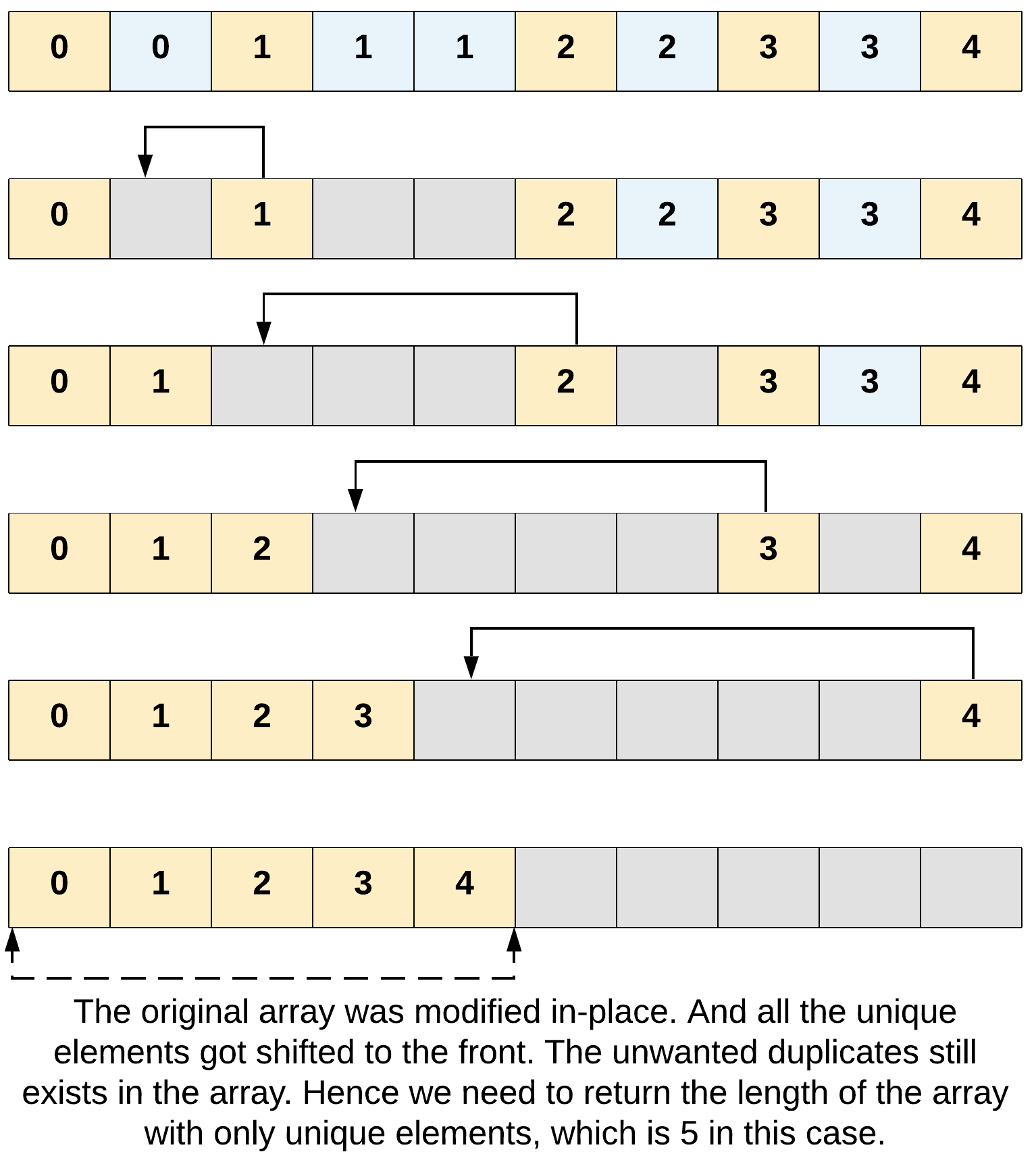
We need to modify the array in-place and the size of the final array would potentially be smaller than the size of the input array. So, we ought to use a two-pointer approach here. One, that would keep track of the current element in the original array and another one for just the unique elements.

   Hide Hint #3

Essentially, once an element is encountered, you simply need to **bypass** its duplicates and move on to the next unique element.

**A Better Repeated Deletion Algorithm – Answer**

Anyway, the algorithm with *O*(*N*) space is surprisingly similar to the one without. Interestingly, it's simpler though, because it doesn't need to firstly determine the size of the output.



Implementing this requires the use of the **two-pointer technique**. This is where we iterate over the Array in two different places at the same time.

1. Read all the elements like we did before, to identify the duplicates. We call this our readPointer.
2. Keep track of the next position in the front to write the next unique element we've found. We call this our writePointer.

Here's the algorithm in Java code.

|  |
| --- |
| public int removeDuplicates(int[] nums) {    // Check for edge cases.  if (nums == null) {  return 0;  }    // Use the two pointer technique to remove the duplicates in-place.  // The first element shouldn't be touched; it's already in its correct place.  int writePointer = 1;  // Go through each element in the Array.  for (int readPointer = 1; readPointer < nums.length; readPointer++) {  // If the current element we're reading is \*different\* to the previous  // element...  if (nums[readPointer] != nums[readPointer - 1]) {  // Copy it into the next position at the front, tracked by writePointer.  nums[writePointer] = nums[readPointer];  // And we need to now increment writePointer, because the next element  // should be written one space over.  writePointer++;  }  }    // This turns out to be the correct length value.  return writePointer;  } |

You're quite possibly surprised that this even works. How are we not overwriting any elements that we haven't yet looked at?! The key thing to notice is that the condition is such that it is impossible for writePointer to ever get ahead of the readPointer. This means that we would never overwrite a value that we haven't yet read

This was just a very brief introduction to the very versatile and widely used **two-pointer technique**. It is one of the main techniques used for in-place Array algorithms. We'll be looking at it further in the next Array explore card!

### **When to Use In-Place Array Operations**

It's important to know when to use in-place Array operations—they might not always be the way to go.

For example, if we'll need the original array values again later, then we shouldn't be overwriting them. In these cases, it's best to create a copy to work with, or to simply not use in-place techniques. It's important to be **very** careful when working with existing code that somebody else has written. If other code is depending on the original Array to work, then you might completely break the program if you modify that Array!

In-place operations are valuable when appropriate because they reduce the space complexity of an algorithm. Instead of requiring *O*(*N*) space, we can reduce it down to *O*(1).

Here's some more more challenging problems for you to try. Again, try not to make a new Array—instead use the in-place technique.

**Move Zeroes**

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.

**Sort Array By Parity**

Given an array A of non-negative integers, return an array consisting of all the even elements of A, followed by all the odd elements of A.

You may return any answer array that satisfies this condition.

**Example 1:**

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

**Output:** [2,4,3,1]

The outputs [4,2,3,1], [2,4,1,3], and [4,2,1,3] would also be accepted.

**Note:**

1. 1 <= A.length <= 5000
2. 0 <= A[i] <= 5000

## Solution

#### **Approach 1: Sort**

**Intuition and Algorithm**

Use a custom comparator when sorting, to sort by parity.

|  |
| --- |
| class Solution {  public int[] sortArrayByParity(int[] A) {  Integer[] B = new Integer[A.length];  for (int t = 0; t < A.length; ++t)  B[t] = A[t];  Arrays.sort(B, (a, b) -> Integer.compare(a%2, b%2));  for (int t = 0; t < A.length; ++t)  A[t] = B[t];  return A;  /\* Alternative:  return Arrays.stream(A)  .boxed()  .sorted((a, b) -> Integer.compare(a%2, b%2))  .mapToInt(i -> i)  .toArray();  \*/  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*log*N*), where *N* is the length of A.
* Space Complexity: *O*(*N*) for the sort, depending on the built-in implementation of sort.

#### **Approach 2: Two Pass**

**Intuition and Algorithm**

Write all the even elements first, then write all the odd elements.

|  |
| --- |
| class Solution {  public int[] sortArrayByParity(int[] A) {  int[] ans = new int[A.length];  int t = 0;  for (int i = 0; i < A.length; ++i)  if (A[i] % 2 == 0)  ans[t++] = A[i];  for (int i = 0; i < A.length; ++i)  if (A[i] % 2 == 1)  ans[t++] = A[i];  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*), where N*N* is the length of A.
* Space Complexity: *O*(*N*), the space used by the answer.

#### **Approach 3: In-Place**

**Intuition**

If we want to do the sort in-place, we can use quicksort, a standard textbook algorithm.

**Algorithm**

We'll maintain two pointers i and j. The loop invariant is everything below i has parity 0 (ie. A[k] % 2 == 0 when k < i), and everything above j has parity 1.

Then, there are 4 cases for (A[i] % 2, A[j] % 2):

* If it is (0, 1), then everything is correct: i++ and j--.
* If it is (1, 0), we swap them so they are correct, then continue.
* If it is (0, 0), only the i place is correct, so we i++ and continue.
* If it is (1, 1), only the j place is correct, so we j-- and continue.

Throughout all 4 cases, the loop invariant is maintained, and j-i is getting smaller. So eventually we will be done with the array sorted as desired.

|  |
| --- |
| class Solution {  public int[] sortArrayByParity(int[] A) {  int i = 0, j = A.length - 1;  while (i < j) {  if (A[i] % 2 > A[j] % 2) {  int tmp = A[i];  A[i] = A[j];  A[j] = tmp;  }  if (A[i] % 2 == 0) i++;  if (A[j] % 2 == 1) j--;  }  return A;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*), where *N* is the length of A. Each step of the while loop makes j-i decrease by at least one. (Note that while quicksort is *O*(*N*log*N*) normally, this is *O*(*N*) because we only need one pass to sort the elements.)
* Space Complexity: *O*(1) in additional space complexity.

**Remove Element**

Given an array nums and a value val, remove all instances of that value [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) and return the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

**Clarification:**

Confused why the returned value is an integer but your answer is an array?

Note that the input array is passed in by **reference**, which means a modification to the input array will be known to the caller as well.

Internally you can think of this:

// **nums** is passed in by reference. (i.e., without making a copy)

int len = removeElement(nums, val);

// any modification to **nums** in your function would be known by the caller.

// using the length returned by your function, it prints the first **len** elements.

for (int i = 0; i < len; i++) {

    print(nums[i]);

}

**Example 1:**

**Input:** nums = [3,2,2,3], val = 3

**Output:** 2, nums = [2,2]

**Explanation:** Your function should return length = **2**, with the first two elements of nums being **2**.

It doesn't matter what you leave beyond the returned length. For example if you return 2 with nums = [2,2,3,3] or nums = [2,2,0,0], your answer will be accepted.

**Example 2:**

**Input:** nums = [0,1,2,2,3,0,4,2], val = 2

**Output:** 5, nums = [0,1,4,0,3]

**Explanation:** Your function should return length = **5**, with the first five elements of *nums* containing **0**, **1**, **3**, **0**, and **4**. Note that the order of those five elements can be arbitrary. It doesn't matter what values are set beyond the returned length.

**Constraints:**

* 0 <= nums.length <= 100
* 0 <= nums[i] <= 50
* 0 <= val <= 100

Hide Hint #1

The problem statement clearly asks us to modify the array in-place and it also says that the element beyond the new length of the array can be anything. Given an element, we need to remove all the occurrences of it from the array. We don't technically need to **remove** that element per-say, right?

   Hide Hint #2

We can move all the occurrences of this element to the end of the array. Use two pointers!  
Diagram

Description automatically generated with medium confidence

   Hide Hint #3

Yet another direction of thought is to consider the elements to be removed as non-existent. In a single pass, if we keep copying the visible elements in-place, that should also solve this problem for us.

Conclusion

We hope you're now feeling confident with the basics of Arrays! Because practice makes perfect, here are a few more practice problems for you!

**What's Next?**

Let's recap what we looked at in this explore card:

* We explored what the Array data structure is all about.
* We looked at the Java syntax for creating Arrays.
* We looked at the Java syntax for reading and writing from Arrays.
* We designed basic insertion, deletion, and search algorithms for Arrays.
* We played around with in-place Array algorithms.
* We solved heaps of fun and exciting problems!

Here at LeetCode, we've already started developing a follow-up Arrays Explore Card! In that card, we'll be going over some more advanced techniques for working with Arrays.

What other techniques could there be with Arrays, you might be wondering? Well, wonder no more. Here is a quick taster!  
**Two-Pointer**

Diagram

Description automatically generated

### **Circular Array**

A picture containing diagram

Description automatically generated

For now though, here's a few last questions for you to really practice what you've learned, and to get prepared for the next Arrays Explore Card!

**Height Checker**

Students are asked to stand in non-decreasing order of heights for an annual photo.

Return the minimum number of students that must move in order for all students to be standing in non-decreasing order of height.

Notice that when a group of students is selected they can reorder in any possible way between themselves and the non selected students remain on their seats.

**Example 1:**

**Input:** heights = [1,1,4,2,1,3]

**Output:** 3

**Explanation:**

Current array : [1,1,4,2,1,3]

Target array : [1,1,1,2,3,4]

On index 2 (0-based) we have 4 vs 1 so we have to move this student.

On index 4 (0-based) we have 1 vs 3 so we have to move this student.

On index 5 (0-based) we have 3 vs 4 so we have to move this student.

**Example 2:**

**Input:** heights = [5,1,2,3,4]

**Output:** 5

**Example 3:**

**Input:** heights = [1,2,3,4,5]

**Output:** 0

**Constraints:**

* 1 <= heights.length <= 100
* 1 <= heights[i] <= 100

Hint #1

Build the correct order of heights by sorting another array, then compare the two arrays.

**Max Consecutive Ones II**

Given a binary array, find the maximum number of consecutive 1s in this array if you can flip at most one 0.

**Example 1:**

**Input:** [1,0,1,1,0]

**Output:** 4

**Explanation:** Flip the first zero will get the the maximum number of consecutive 1s.

After flipping, the maximum number of consecutive 1s is 4.

**Note:**

* The input array will only contain 0 and 1.
* The length of input array is a positive integer and will not exceed 10,000

**Follow up:**  
What if the input numbers come in one by one as an **infinite stream**? In other words, you can't store all numbers coming from the stream as it's too large to hold in memory. Could you solve it efficiently?

## Solution

**Intuition**

First, let's understand our problem.

"Given a binary array, find the maximum number of consecutive 1s in this array..."

Okay makes sense so far.

"...if you can flip at most one 0."

Huh? What does that even mean?

Let's translate that into something more concrete. We can rephrase "if you can flip at most one 0" into "allowing at most one 0 within an otherwise consecutive run of 1s". These statements are equal because if we had one 0 in our consecutive array, we could flip it to satisfy our condition. Note, we're not actually going to flip the 0 which will make our approach simpler.

So our new problem statement is:

"Given a binary array, find the maximum number of consecutive 1s in this array, allowing at most one 0 within an otherwise consecutive run of 1s"

#### **Approach 1: Brute Force**

**Algorithm**

Let's start simple and work our way up.

A brute force solution usually involves trying to check every single possibility. It'll look something like this:

* Check every possible consecutive sequence
* Count how many 0's are in each sequence
* If our sequence has one or fewer 0's, check if that's the longest consecutive sequence of 1's.

|  |
| --- |
| class Solution {  public int findMaxConsecutiveOnes(int[] nums) {  int longestSequence = 0;  for (int left = 0; left < nums.length; left++) {  int numZeroes = 0;  // check every consecutive sequence  for (int right = left; right < nums.length; right++) {  // count how many 0's  if (nums[right] == 0) {  numZeroes += 1;  }  // # update answer if it's valid  if (numZeroes <= 1) {  longestSequence = Math.max(longestSequence, right - left + 1);  }  }  }  return longestSequence;  }  } |

**Interview Tip:** Often times the interviewer doesn't need to see you code the brute force solution. State the brute force approach out loud and discuss his/her expectations. Either way, communicating proactively will give you major bonus points.

**Complexity Analysis**

Let *n* be equal to the length of the input nums array.

* Time complexity : O(n^2). The nested for loops turn our approach into a quadratic solution because for every index, we have to check every other index in the array.
* Space complexity : *O*(1). We are using 4 variables: left, right, numZeroes, longestSequence. The number of variables are constant and do not change based on the size of the input.

#### **Approach 2: Sliding Window**

**Intuition**

The naive approach works but our interviewer is not convinced. Let's see how we can optimize the code we just wrote.

The brute force solution had a time complexity of O(n^2)*O*(*n*2). What was the bottleneck? Checking every single consecutive sequence. Intuitively, we know we're doing repeated work because sequences overlap. We are checking consecutive sequence blindly. We need to establish some rules on how to move our sequence forward.

* If our sequence is valid, let's continue expanding our sequence (because our goal is to get the largest sequence possible).
* If our sequence is invalid, let's stop expanding and contract our sequence (because an invalid sequence will never count towards our largest sequence).

The pattern that comes to mind for expanding/contracting sequences is the sliding window. Let's define valid and invalid states.

* Valid State = one or fewer 0's in our current sequence
* Invalid State = two 0's in our current sequence

**Algorithm**

Great. How do we apply all this to the sliding window?

Let's use left and right pointers to keep track of the current sequence a.k.a. our window. Let's expand our window by moving the right pointer forward until we reach a point where we have more than one 0 in our window. When we reach this invalid state, let's contract our window by moving the left pointer forward until we have a valid window again. By expanding and contracting our window from valid and invalid states, we are able to traverse the array efficiently without repeated overlapping work.

Now we can break this approach down into a few actionable steps:

While our window is in bounds of the array...

1. Add the rightmost element to our window
2. Check if our window is invalid. If so, contract the window until valid.
3. Update our the longest sequence we've seen so far
4. Continue to expand our window

This will look like this:

Application

Description automatically generated with medium confidenceDiagram

Description automatically generated with medium confidence

A picture containing table

Description automatically generatedChart

Description automatically generated with medium confidence

A picture containing chart

Description automatically generatedChart

Description automatically generated

Timeline

Description automatically generated with medium confidenceA picture containing diagram

Description automatically generated

A picture containing diagram

Description automatically generatedDiagram

Description automatically generated

|  |
| --- |
| class Solution {  public int findMaxConsecutiveOnes(int[] nums) {  int longestSequence = 0;  int left = 0;  int right = 0;  int numZeroes = 0;  // while our window is in bounds  while (right < nums.length) {  // add the right most element into our window  if (nums[right] == 0) {  numZeroes++;  }  // if our window is invalid, contract our window  while (numZeroes == 2) {  if (nums[left] == 0) {  numZeroes--;  }  left++;  }  // update our longest sequence answer  longestSequence = Math.max(longestSequence, right - left + 1);  // expand our window  right++;  }  return longestSequence;  }  } |

**Complexity Analysis**

Let *n* be equal to the length of the input nums array.

* Time complexity : *O*(*n*). Since both the pointers only move forward, each of the left and right pointer traverse a maximum of n steps. Therefore, the timecomplexity is *O*(*n*).
* Space complexity : *O*(1). Same as the previous approach. We don't store anything other than variables. Thus, the space we use is constant because it is not correlated to the length of the input array.

**Third Maximum Number**

Given integer array nums, return *the third maximum number in this array*. If the third maximum does not exist, return the maximum number.

**Example 1:**

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

**Output:** 1

**Explanation:** The third maximum is 1.

**Example 2:**

**Input:** nums = [1,2]

**Output:** 2

**Explanation:** The third maximum does not exist, so the maximum (2) is returned instead.

**Example 3:**

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

**Output:** 1

**Explanation:** Note that the third maximum here means the third maximum distinct number.

Both numbers with value 2 are both considered as second maximum.

**Constraints:**

* 1 <= nums.length <= 104
* -231 <= nums[i] <= 231 - 1

**Follow up:** Can you find an O(n) solution?

## Solution

#### **Approach 1: Use a Set and Delete Maximums**

**Intuition**

Firstly, note that we can't simply sort the values and then select the third-to-end one, because the time complexity of sorting is O(n \, \log \, n)*O*(*n*log*n*) (where n*n* is the length of the input Array). This problem clearly states our solution must have a time complexity of O(n)*O*(*n*) though.

Also, note that there are 2 important pieces of information in the problem description that are easily overlooked:

1. If the third maximum doesn't exist, we must return the ***maximum*** (never the second maximum like one might assume!).
2. Duplicates should be ignored. We want the third maximum distinct value. i.e. for [8, 8, 8, 3, 1] the third maximum is 1, despite the fact that 8 appears three times.

We'll work with the following example Array.

[12, 3, 8, 9, 12, 12, 7, 8, 12, 4, 3, 8, 1]

If there were no duplicates in the Array, then a logical strategy would be as follows:

Find the maximum. Delete it.

Find the new maximum. Delete it.

Return the \*new\* maximum.

However, the input Array we're working with could have duplicates. To handle this, we can convert the input into a Set first to remove the duplicates.

Converting our input Array example into a Set gives us the following:

{12, 3, 8, 9, 7, 4, 1}

We then need to find the maximum in the Set. This can be done using a library function, or if necessary, your own function that loops through the list keeping track of the maximum seen so far, and then returns the maximum at the end.

The maximum from our example is 12.

Now, we need to delete 12 from the Set. This leaves us with:

{3, 8, 9, 7, 4, 1}

We can then find and remove the second maximum, following the same process.

The second maximum is 9 (the maximum of what's left in the Set).

Removing it leaves us with the following:

{3, 8, 7, 4, 1}

Finally, we can return the maximum of what's left, which is 8.

Remember that if the third maximum doesn't exist, then we need to return the maximum of the original Array. We can detect this situation as soon as we have converted the input Array into a Set, because it will contain less than 3 values.

**Algorithm**

|  |
| --- |
| public int thirdMax(int[] nums) {  // Put the input integers into a HashSet.  Set<Integer> setNums = new HashSet<>();  for (int num : nums) setNums.add(num);  // Find the maximum.  int maximum = Collections.max(setNums);  // Check whether or not this is a case where we  // need to return the \*maximum\*.  if (setNums.size() < 3) {  return maximum;  }  // Otherwise, continue on to finding the third maximum.  setNums.remove(maximum);  int secondMaximum = Collections.max(setNums);  setNums.remove(secondMaximum);  return Collections.max(setNums);  } |

**Complexity Analysis**

* Time Complexity : *O*(*n*).

Putting the input Array values into a HashSet has a cost of *O*(*n*), as each value costs *O*(1) to place, and there are *n* of them.

Finding the maximum in a HashSet has a cost of *O*(*n*), as all the values need to be looped through. We do this 3 times, giving *O*(3⋅*n*)=*O*(*n*) as we drop constants in big-oh notation.

Deleting a value from a HashSet has a cost of *O*(1), so we can ignore this.

In total, we're left with *O*(*n*)+*O*(*n*)=*O*(*n*).

* Space Complexity : *O*(*n*).

In the worst case, the HashSet is the same size as the input Array, and so requires *O*(*n*) space to store.

#### **Approach 2: Seen-Maximums Set**

**Intuition**

In the previous approach, we deleted the maximum and second maximum so that we could easily find the third maximum. We had to convert the input Array into a Set so that duplicates weren't super complicated to handle.

Instead of deleting items though, we could instead keep a Set of maximums we've already seen. Then when we are searching for a maximum, we can ignore any values that are already in the seen Set.

This will also handle duplicates elegantly—if for example we had the input set [12, 12, 4, 2, 12, 1], then the first value we'd put into the seen maximums Set would be 12. Then when we find the second maximum, the algorithm knows to ignore all the 12s.

**Algorithm**

|  |
| --- |
| class Solution {  public int thirdMax(int[] nums) {  Set<Integer> seenMaximums = new HashSet<>();    for (int i = 0; i < 3; i++) {  Integer curMaximum = maxIgnoringSeenMaximums(nums, seenMaximums);  if (curMaximum == null) {  return Collections.max(seenMaximums);  }  seenMaximums.add(curMaximum);  }  return Collections.min(seenMaximums);  }  private Integer maxIgnoringSeenMaximums(int[] nums, Set<Integer> seenMaximums) {    Integer maximum = null;  for (int num : nums) {  if (seenMaximums.contains(num)) {  continue;  }  if (maximum == null || num > maximum) {  maximum = num;  }  }  return maximum;  }  } |

**Complexity Analysis**

* Time Complexity : *O*(*n*).

For each of the three times we find the next maximum, we need to perform an *O*(*n*) scan. Because there are only, at most, three scans the total time complexity is just *O*(*n*).

The Set operations are all *O*(1) because there are only at most 3 items in the Set.

* Space Complexity : *O*(1).

Because seenMaximums can contain at most 3 items, the space complexity is only *O*(1).

#### **Approach 3: Keep Track of 3 Maximums Using a Set**

**Intuition**

So far, our approaches have required multiple parses through the input array. While this is still O(n)*O*(*n*) in big-oh notation, it'd be good if we could solve it in a single parse. One way is to simply use a Set to keep track of the 3 maximum values we've seen so far. While you could achieve something similar using 3 variables (maximum, secondMaximum, and thirdMaximum), this is messy to work with and is poor programming practice.

For each number in the Array, we add it into the Set of maximums. If this causes there to be more than 3 numbers in the Set, then we evict the smallest number.

At the end, we check whether or not there are 3 numbers in the Set. If there are, this means the third maximum exists, and will be the minimum in the Set. If not, this means there was no third maximum, and so we should return the maximum of the Set, as per the problem requirements.

Here is an animation showing the approach.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Application

Description automatically generated with low confidence

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

A picture containing diagram

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Application

Description automatically generated with medium confidence

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

**Algorithm**

|  |
| --- |
| public int thirdMax(int[] nums) {  Set<Integer> maximums = new HashSet<Integer>();  for (int num : nums) {  maximums.add(num);  if (maximums.size() > 3) {  maximums.remove(Collections.min(maximums));  }  }  if (maximums.size() == 3) {  return Collections.min(maximums);  }  return Collections.max(maximums);  } |

**Complexity Analysis**

* Time Complexity : *O*(*n*).

For each of the n*n* values in the input Array, we insert it into a Set for a cost of *O*(1). We then sometimes find and remove the minimum of the Set. Because there are never more than 33 items in the Set, the time complexity of doing this is *O*(1).

In total, we're left with *O*(*n*).

* Space Complexity : *O*(1).

Because maximums never holds more than 33 items at a time, it is considered to be constant O*O*(1).

#### **Related Problem - Kth Largest Element in an Array**

A related problem, which would possibly be used as a follow up question in an interview is [Kth Largest Element in an Array](https://leetcode.com/problems/kth-largest-element-in-an-array/).

**Find All Numbers Disappeared in an Array**

Given an array of integers where 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear twice and others appear once.

Find all the elements of [1, *n*] inclusive that do not appear in this array.

Could you do it without extra space and in O(*n*) runtime? You may assume the returned list does not count as extra space.

**Example:**

**Input:**

[4,3,2,7,8,2,3,1]

**Output:**

[5,6]

   Hide Hint #1

This is a really easy problem if you decide to use additional memory. For those trying to write an initial solution using additional memory, think **counters!**

   Hide Hint #2

However, the trick really is to not use any additional space than what is already available to use. Sometimes, multiple passes over the input array help find the solution. However, there's an interesting piece of information in this problem that makes it easy to re-use the input array itself for the solution.

   Hide Hint #3

The problem specifies that the numbers in the array will be in the range [1, n] where n is the number of elements in the array. Can we use this information and modify the array in-place somehow to find what we need?

## Solution

#### **Approach 1: Using Hash Map**

**Intuition**

The intuition behind using a hash map is pretty clear in this case. We are given that the array would be of size N and it should contain numbers from 1 to N. However, some of the numbers are missing. All we have to do is keep track of which numbers we encounter in the array and then iterate from 1⋯*N* and check which numbers did not appear in the hash table. Those will be our missing numbers. Let's look at a formal algorithm based on this idea and then an animation explaining the same with the help of a simple example.

**Algorithm**

1. Initialize a hash map, hash to keep track of the numbers that we encounter in the array. Note that we can use a set data structure as well in this case since we are not concerned about the frequency counts of elements.

A picture containing table

Description automatically generated

Note that for the purposes of illustration, we have use a hash map of size 14 and have ordered the keys of the hash map from 0 to 14. Also, we will be using a simple hash function that directly maps the array entries to their corresponding keys in the hash map. Usually, the mapping is not this simple and is dependent upon the hash function being used in the implementation of the hash map.

1. Next, iterate over the given array one element at a time and for each element, insert an entry in the hash map. Even if an entry were to exist before in the hash map, it will simply be over-written. For the above example, let's look at the final state of the hash map once we process the last element of the array.

Table

Description automatically generated with medium confidence

1. Now that we know the unique set of elements from the array, we can simply find out the missing elements from the range 1⋯*N*.
2. Iterate over all the numbers from 1⋯*N* and for each number, check if there's an entry in the hash map. If there is no entry, add that missing number to a result array that we will return from the function eventually.

Table

Description automatically generated with low confidence

A picture containing table

Description automatically generated

Graphical user interface, application

Description automatically generated

Application, table

Description automatically generated with medium confidence

Table

Description automatically generated with low confidence

Table

Description automatically generated with low confidence

Table

Description automatically generated with low confidence

Table

Description automatically generated

Table

Description automatically generated with medium confidence

|  |
| --- |
| class Solution {  public List<Integer> findDisappearedNumbers(int[] nums) {    // Hash table for keeping track of the numbers in the array  // Note that we can also use a set here since we are not  // really concerned with the frequency of numbers.  HashMap<Integer, Boolean> hashTable = new HashMap<Integer, Boolean>();    // Add each of the numbers to the hash table  for (int i = 0; i < nums.length; i++) {  hashTable.put(nums[i], true);  }    // Response array that would contain the missing numbers  List<Integer> result = new LinkedList<Integer>();    // Iterate over the numbers from 1 to N and add all those  // that don't appear in the hash table.  for (int i = 1; i <= nums.length; i++) {  if (!hashTable.containsKey(i)) {  result.add(i);  }  }    return result;  }  } |

**Complexity Analysis**

* Time Complexity : *O*(*N*)
* Space Complexity : *O*(*N*)

#### **Approach 2: O(1) Space InPlace Modification Solution**

**Intuition**

We definitely need to keep track of all the unique numbers that appear in the array. However, we don't want to use any extra space for it. This solution that we will look at in just a moment springs from the fact that

All the elements are in the range [1, N]

Since we are given this information, we can make use of the input array itself to somehow mark visited numbers and then find our missing numbers. Now, we don't want to change the actual data in the array but who's stopping us from changing the magnitude of numbers in the array? That is the basic idea behind this algorithm.

We will be negating the numbers seen in the array and use the sign of each of the numbers for finding our missing numbers. We will be treating numbers in the array as indices and mark corresponding locations in the array as negative.

**Algorithm**

1. Iterate over the input array one element at a time.
2. For each element nums[i], mark the element at the corresponding location negative if it's not already marked so i.e. nums[[*nums*[*i*]−1]×−1 .
3. Now, loop over numbers from 1⋯*N* and for each number check if nums[j] is negative. If it is negative, that means we've seen this number somewhere in the array.
4. Add all the numbers to the resultant array which don't have their corresponding locations marked as negative in the original array.

Diagram

Description automatically generated with medium confidence

Diagram

Description automatically generated

Diagram

Description automatically generated with low confidence

Chart

Description automatically generated

Chart, scatter chart

Description automatically generated

Chart

Description automatically generated

A picture containing diagram

Description automatically generated

Scatter chart

Description automatically generated

Calendar

Description automatically generated

Scatter chart

Description automatically generated with low confidence

Calendar

Description automatically generated

Calendar

Description automatically generated with medium confidence

Calendar

Description automatically generated with low confidence

Calendar

Description automatically generated

Calendar

Description automatically generated

Calendar

Description automatically generated with medium confidence

Calendar

Description automatically generated with low confidence

|  |
| --- |
| class Solution {  public List<Integer> findDisappearedNumbers(int[] nums) {    // Iterate over each of the elements in the original array  for (int i = 0; i < nums.length; i++) {    // Treat the value as the new index  int newIndex = Math.abs(nums[i]) - 1;    // Check the magnitude of value at this new index  // If the magnitude is positive, make it negative  // thus indicating that the number nums[i] has  // appeared or has been visited.  if (nums[newIndex] > 0) {  nums[newIndex] \*= -1;  }  }    // Response array that would contain the missing numbers  List<Integer> result = new LinkedList<Integer>();    // Iterate over the numbers from 1 to N and add all those  // that have positive magnitude in the array  for (int i = 1; i <= nums.length; i++) {    if (nums[i - 1] > 0) {  result.add(i);  }  }    return result;  }  } |

**Complexity Analysis**

* Time Complexity : *O*(*N*)
* Space Complexity : *O*(1) since we are reusing the input array itself as a hash table and the space occupied by the output array doesn't count toward the space complexity of the algorithm.

**Squares of a Sorted Array**

Given an integer array nums sorted in **non-decreasing** order, return *an array of****the squares of each number****sorted in non-decreasing order*.

**Example 1:**

**Input:** nums = [-4,-1,0,3,10]

**Output:** [0,1,9,16,100]

**Explanation:** After squaring, the array becomes [16,1,0,9,100].

After sorting, it becomes [0,1,9,16,100].

**Example 2:**

**Input:** nums = [-7,-3,2,3,11]

**Output:** [4,9,9,49,121]

**Constraints:**

* 1 <= nums.length <= 104
* -104 <= nums[i] <= 104
* nums is sorted in **non-decreasing** order.

**Follow up:** Squaring each element and sorting the new array is very trivial, could you find an O(n) solution using a different approach?

## Solution

#### **Approach 1: Sort**

**Intuition and Algorithm**

Create an array of the squares of each element, and sort them.

|  |
| --- |
| class Solution {  public int[] sortedSquares(int[] A) {  int N = A.length;  int[] ans = new int[N];  for (int i = 0; i < N; ++i)  ans[i] = A[i] \* A[i];  Arrays.sort(ans);  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*log*N*), where *N* is the length of A.
* Space complexity : O(*N*) or O(log*N*)
  + The space complexity of the sorting algorithm depends on the implementation of each program language.
  + For instance, the list.sort() function in Python is implemented with the [Timsort](https://en.wikipedia.org/wiki/Timsort) algorithm whose space complexity is O(*N*).
  + In Java, the [Arrays.sort()](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html" \l "sort-byte:A-) is implemented as a variant of quicksort algorithm whose space complexity is O(log*N*).

#### **Approach 2: Two Pointer**

**Intuition**

Since the array A is sorted, loosely speaking it has some negative elements with squares in decreasing order, then some non-negative elements with squares in increasing order.

For example, with [-3, -2, -1, 4, 5, 6], we have the negative part [-3, -2, -1] with squares [9, 4, 1], and the positive part [4, 5, 6] with squares [16, 25, 36]. Our strategy is to iterate over the negative part in reverse, and the positive part in the forward direction.

**Algorithm**

We can use two pointers to read the positive and negative parts of the array - one pointer j in the positive direction, and another i in the negative direction.

Now that we are reading two increasing arrays (the squares of the elements), we can merge these arrays together using a two-pointer technique.

|  |
| --- |
| class Solution {  public int[] sortedSquares(int[] nums) {  int n = nums.length;  int[] result = new int[n];  int left = 0;  int right = n - 1;  for (int i = n - 1; i >= 0; i--) {  int square;  if (Math.abs(nums[left]) < Math.abs(nums[right])) {  square = nums[right];  right--;  } else {  square = nums[left];  left++;  }  result[i] = square \* square;  }  return result;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*), where *N* is the length of A.
* Space Complexity: *O*(*N*) if you take output into account and *O*(1) otherwise.