**A Better Repeated Deletion Algorithm - Answer**

Anyway, the algorithm with O(N)*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.

A diagram of a number system

AI-generated content may be incorrect.

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)*O*(*N*) space, we can reduce it down to O(1)*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.