**A Better Repeated Deletion Algorithm - Intro**

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 if 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)*O*(1). However, the time complexity's not so flash, at O(N2)*O*(*N*2). This is because of the nested loop.

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

A diagram of a number

AI-generated content may be incorrect.

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)*O*(1) and a time complexity of O(N)*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)*O*(*N*) time. Good luck!