<https://leetcode.com/problems/remove-duplicates-from-sorted-array/>

Given an integer array nums sorted in **non-decreasing order**, remove the duplicates **[in-place](https://en.wikipedia.org/wiki/In-place_algorithm)** such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the **first part** of the array nums. More formally, if there are k elements after removing the duplicates, then the first k elements of nums should hold the final result. It does not matter what you leave beyond the first k elements.

Return k *after placing the final result in the first* k *slots of* nums.

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.

**Custom Judge:**

The judge will test your solution with the following code:

int[] nums = [...]; // Input array

int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;

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

assert nums[i] == expectedNums[i];

}

If all assertions pass, then your solution will be **accepted**.

**Example 1:**

Input: nums = [1,1,2]

Output: 2, nums = [1,2,\_]

Explanation: Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

**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 k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

**Constraints:**

* 1 <= nums.length <= 3 \* 104
* -100 <= nums[i] <= 100
* nums is sorted in **non-decreasing** order.

**Attempt 1: 2023-02-26**

**Solution 1:  Two Pointers (10 min)**

**Style 1: nums[i++] = nums[j - 1] which always record previous digit**

class Solution {

public int removeDuplicates(int[] nums) {

int i = 0;

int j = 1;

while(j < nums.length) {

if(nums[j] != nums[j - 1]) {

nums[i++] = nums[j - 1];

}

j++;

}

// Important: To make up the last digit since

// "nums[i++] = nums[j - 1]" only record previous

// digit when two adjacent digits are different,

// it will cause last digit missing record, to fix

// just make it up out of the while loop

// Test out: nums = [0,0,1,1,1,2,2,3,3,4]

// If no such line: output = [0,1,2,3,1,2,2,3,3,4]

// expected: nums = [0,1,2,3,4,2,2,3,3,4]

nums[i] = nums[j - 1];

return i + 1;

}

}

Time Complexity:O(N), since we only have 2 pointers, and both the pointers will traverse the array at most once.

Space Complexity:O(1), since we are not using any extra space.

**Style 2: nums[i++] = nums[j] which always record current digit**

class Solution {

public int removeDuplicates(int[] nums) {

int i = 0;

int j = 0;

while(j < nums.length - 1) {

if(nums[j] != nums[j + 1]) {

nums[i++] = nums[j];

}

j++;

}

// Make up the last different digit

nums[i] = nums[j];

return i + 1;

}

}

Time Complexity:O(N), since we only have 2 pointers, and both the pointers will traverse the array at most once.

Space Complexity:O(1), since we are not using any extra space.

**Refer to**

<https://leetcode.com/problems/remove-duplicates-from-sorted-array/editorial/>

### **Overview**

The problem would have been simpler if we are allowed to use extra space.

We can create a map which stores all **unique array elements** as the key and **element frequency** as the value.

After populating our map, we get all the unique elements from our array.

We then iterate our map and push all the keys in our input array

However, without using extra space it makes it a bit tricky as we have to modify the existing input array

### **Approach 1: Two indexes approach**

#### **Intuition**

To solve the problem, let's look at the condition carefully,

It is **guaranteed** that the given array is a **sorted array**.

Let k be the count of unique elements in our input array.

It doesn't matter what elements we place after the first k elements.

From the condition, we can have a few observations here,

* Since the array we have is sorted, all duplicate values will be one after the other.
* We need to update the first k elements in an array with unique values and return the value of k.

Using the following intuition, let's understand how to address this problem.

* The problem states that we need to fill the first k elements of an array with unique values
* For doing so, we modify the input array in-place so that we don't use extra space
* In order to perform in-place operations, we use the **Two indexes** approach
* The **first index** updates the value in our input array while reading the data from the **second index**

First Index is responsible for writing unique values in our input array, while Second Index will read the input array and pass all the distinct elements to First Index.

* We continue the above steps until the **second index** reaches the end of an array

#### **Algorithm**

By analyzing the above three key observations, we can derive the following algorithm,

Start both indexes (insertIndex, i) from 1.

insertIndex and i represents our First and second Index respectively.

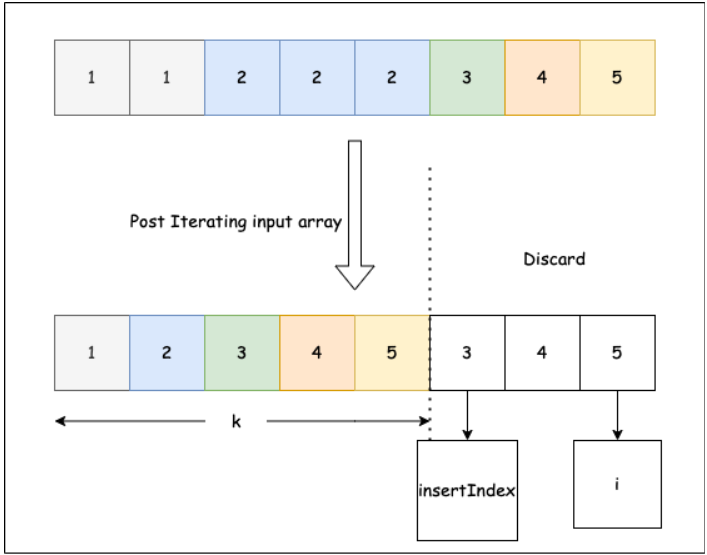
Check if the previous element is different from the current element

The previous element is the element just before our i index i.e element present at arr[i-1]

If found different then perform arr[insertIndex] = arr[i] and increment insertIndex by 1

Increment i index by 1 till we reach end of the array

Note: After reaching the end of the array, our **insertIndex** variable will hold the count of unique elements in our input array.



class Solution {

public int removeDuplicates(int[] nums) {

int insertIndex = 1;

for(int i = 1; i < nums.length; i++){

// We skip to next index if we see a duplicate element

if(nums[i - 1] != nums[i]) {

/\* Storing the unique element at insertIndex index and incrementing

the insertIndex by 1 \*/

nums[insertIndex] = nums[i];

insertIndex++;

}

}

return insertIndex;

}

}

#### **Complexity Analysis**

Let N be the size of the input array.

Time Complexity: O(N), since we only have 2 pointers, and both the pointers will traverse the array at most once.

Space Complexity: O(1), since we are not using any extra space.