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

Given an integer array nums sorted in **non-decreasing order**, remove some duplicates **in-place** such that each unique element appears **at most twice**. 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** 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 = [1,1,1,2,2,3]

Output: 5, nums = [1,1,2,2,3,\_]

Explanation: Your function should return k = 5, with the first five elements of nums being 1, 1, 2, 2 and 3 respectively.

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

**Example 2:**

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

Output: 7, nums = [0,0,1,1,2,3,3,\_,\_]

Explanation: Your function should return k = 7, with the first seven elements of nums being 0, 0, 1, 1, 2, 3 and 3 respectively.

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

**Constraints:**

1 <= nums.length <= 3 \* 10^4

-10^4 <= nums[i] <= 10^4

nums is sorted in **non-decreasing** order.

**Attempt 1: 2023-02-23**

**Solution 1: Two Pointers (30 min)**

class Solution {

    public int removeDuplicates(int[] nums) {

        int i = 0;

        for(int n : nums) {

            if(i < 2 || n > nums[i - 2]) {

                nums[i] = n;

                i++;

            }

        }

        return i;

    }

}

Time Complexity: O(N)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/remove-duplicates-from-sorted-array-ii/solutions/27976/3-6-easy-lines-c-java-python-ruby/comments/119993>

**i和n两个指针，一开始齐头并进，同时指向一个位置，当当前的数值不比他前前一个数值大的时候，意味着出现了3个或3个以上的相同值，此时不满足if条件，i停留在不满足的位置，等待下一个更大的数来替换，当出现下一个更大的数字时再次满足if条件，将i所指向的位置替换为该数字，i指向下一个等待替换，此时if条件再次用以检测用来替换的数字，以保证不出现两次以上的重复。**

**Refer to**

<https://algo.monster/liteproblems/80>

**Problem Description**

Imagine you have a list of numbers that are sorted in ascending order, but some numbers appear more than once. Your task is to modify this list so that each unique number appears no more than twice. However, the challenge is to do this without using any additional space and to modify the original list directly—that means you can't create a new list to hold the result. You need to ensure the final list still remains sorted.

For example, if your list is [1,1,1,2,2,3,3,3,3], your goal is to change it to something like [1,1,2,2,3,3,**,**,\_\_] (with the underscores representing spaces you don't care about).

Ultimately, you'll return the length of the modified list (in the example above, it would be 6 since there are six numbers in the list after duplicates beyond the second instance are removed).

**Intuition**

The solution uses a two-pointer approach that exploits the fact that the input array is already sorted. The essence of the approach is to iterate over the array and make sure that we're copying over each unique element at most twice to the 'front' part of the array.

We use a variable k as a pointer to keep track of the '有效段' (or the valid segment) of the list – the portion that contains no more than two instances of any number. When we find a number that should be part of the valid segment, we copy it to the position indicated by k and increment k.

As we go through the list, for each new element we check:

If k is less than 2, which means we're still filling up the first two slots, we can safely add the number without any checks.

If the current number is not the same as the element two places before it in our '有效段' (valid segment of the list), we know that we haven't yet seen it twice, so we copy it to the current k position.

This way, once we've gone through the entire list, k points just past the last element of the desired valid segment. We don't care what's beyond it, and we return k as the new length of the non-duplicated (up to twice) list.

**Solution Approach**

The solution provided employs a simple algorithm with no additional data structures, adhering to an in-place modification constraint which is necessary for this problem. We use the two-pointer technique, but with just one variable k as the slow-runner pointer, while the for x in nums loop acts as the fast-runner pointer.

Here's a step-by-step explanation of the implementation:

We initialize the pointer k to zero. This will keep track of the position in the array where we will place the next unique element that we want to keep, which should appear at most twice.

We iterate over each number x in nums using a for loop.

For each number, we have two conditions to check:

If k < 2: This means we are at the beginning of nums, and since we can have at least two of the same element, we don't need to check for duplicates yet.

If x != nums[k - 2]: This is checked when k is greater than or equal to 2. Since the array is sorted, if the current number x is different from the two places before the current k index, it means x is different from at least the last two numbers in our "valid segment", so we can safely include x in our result.

If either condition is true, we assign the current number x to the kth position in the array, thereby ensuring it is part of the final array, and increment k.

After the loop finishes, k is now the length of the array with no duplicates (allowing up to two instances of the same number). We return k as the result.

The key to this algorithm is understanding that since the array is sorted, duplicates are always adjacent. By checking two steps back, we ensure that we only keep at most two instances of any element. Furthermore, using only the variable k to manage the valid part of the array ensures that we comply with the O(1) extra space constraint of the problem, as we're just rearranging the elements and not using any extra space.

This is the heart of the solution—the algorithm relies solely on the sorted nature of the array and the clever use of a single index to keep track of our "valid segment".

**Example Walkthrough**

Let's illustrate the solution approach with a small example. Suppose our input array is:

nums = [1, 1, 1, 2, 3, 3, 4]

According to the problem description, we want to modify the array so that no number appears more than twice and we want to do this in place. Here's how we apply the two-pointer technique with the variable k:

We initialize k to zero.

Start iterating over each number in nums.

a. x = 1, k = 0 (k < 2 is True). Place 1 at nums[0], increment k to 1.

b. x = 1, k = 1 (k < 2 is True). Place 1 at nums[1], increment k to 2.

c. x = 1, k = 2 (k < 2 is False, but x != nums[k - 2] is False since nums[0] is 1). We skip this step.

d. x = 2, k = 2 (Since x is different from nums[k - 2] => 2 != nums[0]). Place 2 at nums[2], increment k to 3.

e. x = 3, k = 3 (Since x is different from nums[k - 2] => 3 != nums[1]). Place 3 at nums[3], increment k to 4.

f. x = 3, k = 4 (Since x is different from nums[k - 2] => 3 != nums[2]). Place 3 at nums[4], increment k to 5.

g. x = 4, k = 5 (Since x is different from nums[k - 2] => 4 != nums[3]). Place 4 at nums[5], increment k to 6.

By the end of the iteration, our array looks like this:

nums = [1, 1, 2, 3, 3, 4, \_\_]

Here, \_\_ represents the space we don't care about. The array nums now contains each unique number no more than twice, in sorted order, and k (which is 6) indicates the length of the modified array that we are concerned with. Therefore, we return k as the answer which is 6 in this case.

**Solution Implementation**

class Solution {

public int removeDuplicates(int[] nums) {

// 'k' is the index for placing the next unique element

// or the second occurrence of an existing element

int index = 0;

// Iterate over each element in the array

for (int num : nums) {

// If the current position is less than 2 (i.e., we are at the start of the array)

// or if the current element is different than the element two positions behind

// then consider it for inclusion in the array

if (index < 2 || num != nums[index - 2]) {

// Place the current element at the 'index' position and increment 'index'

nums[index] = num;

index++;

}

}

// The 'index' represents the length of the array without duplicates

// allowing up to two occurrences

return index;

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the code is O(n), where n is the number of elements in the input list nums. This is because the code consists of a single loop that goes through all elements of the list exactly once.

**Space Complexity**

The space complexity of the code is O(1). No additional space is required that is dependent on the input size. The variable k is used to keep track of the position in the array while overwriting duplicates, but this does not scale with the size of the input.