<https://leetcode.com/problems/minimum-deletions-to-make-array-beautiful/description/>

You are given a **0-indexed** integer array nums. The array nums is **beautiful** if:

nums.length is even.

nums[i] != nums[i + 1] for all i % 2 == 0.

Note that an empty array is considered beautiful.

You can delete any number of elements from nums. When you delete an element, all the elements to the right of the deleted element will be **shifted one unit to the left** to fill the gap created and all the elements to the left of the deleted element will remain **unchanged**.

Return the **minimum** number of elements to delete from nums to make it beautiful.

**Example 1:**

Input: nums = [1,1,2,3,5]

Output: 1

Explanation: You can delete either nums[0] or nums[1] to make nums = [1,2,3,5] which is beautiful. It can be proven you need at least 1 deletion to make nums beautiful.

**Example 2:**

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

Output: 2

Explanation: You can delete nums[0] and nums[5] to make nums = [1,2,2,3] which is beautiful. It can be proven you need at least 2 deletions to make nums beautiful.

**Constraints:**

1 <= nums.length <= 10^5

0 <= nums[i] <= 10^5

**Attempt 1: 2024-12-27**

**Solution 1: Greedy + Math (180 min)**

class Solution {

    public int minDeletion(int[] nums) {

        int delete\_count = 0;

        for(int i = 0; i < nums.length - 1; i++) {

            if(nums[i] == nums[i + 1]) {

                delete\_count++;

            } else {

                i++;

            }

        }

        if((nums.length - delete\_count) % 2 == 1) {

            delete\_count++;

        }

        return delete\_count;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Step by Step**

**But in the solution i didn't see any action reflect handling of "When you delete an element, all the elements to the right of the deleted element will be shifted one unit to the left to fill the gap created" ?So how we bypass this ?**

**如果要物理上去掉重复的数字，效果如下，但是随之而来的问题是 index 的全面改动导致后续的连续比较相对困难**

index: 0 1 2 3 4 5 6 7 8

nums: 2 2 5 5 5 8 8 8 8

index: 0 1 2 3 4 5 6 7 8

Shift 1: 2 5 5 5 8 8 8 8

index: 0 1 2 3 4 5 6 7 8

Shift 2: 2 5 5 8 8 8 8

index: 0 1 2 3 4 5 6 7 8

Shift 3: 2 5 5 8 8 8

index: 0 1 2 3 4 5 6 7 8

Shift 4: 2 5 5 8 8

index: 0 1 2 3 4 5 6 7 8

1 more: 2 5 5 8 -> make it even length

**那么如何在不物理上去除重复数字的情况下，依然满足题目的连续后续去除要求呢？**

index: 0 1 2 3 4 5 6 7 8

nums: 2 2 5 5 5 8 8 8 8

index: 0 1 2 3 4 5 6 7 8

Delete1: x 2 5 5 5 8 8 8 8

i = 0, delete\_count = 0, because nums[i](nums[0] = 2) == nums[i + 1](nums[1] = 2)

No physical remove but naturally following for loop auto increase index 'i' from 0 to 1

the '2' at index 0 implicitly removed, or in reversed perspective, we didn't remove

the '2' at index 0 but remove the '2' at index 1 and then forwarding '2' at index 0

to index 1 to fill in the gap left by removing the '2' at index 1, delete\_count++ = 1,

and following comparison dealing with concreate subarray as below:

index: . 1 2 3 4 5 6 7 8

nums: . 2 5 5 5 8 8 8 8

i = 1, delete\_count = 1, because nums[i](nums[1] = 2) != nums[i + 1](nums[2] = 5)

Just additionally increase index 'i' again from 1 to 2 to skip the non-duplicate nums[2](compare with nums[1])

index: 0 1 2 3 4 5 6 7 8

Delete2: x 2 5 x 5 8 8 8 8

For loop auto increase 'i' from 2 to 3 (don't worry about index 'i' = 3 is not even index now,

since we don't physically delete the number but only on index level, current index 'i' = 3

actually equvalient to 'i' = 2 if we do real shift to left after first deletion happen at 'i' = 0)

i = 3, delete\_count = 1, because nums[i](nums[3] = 5) == nums[i + 1](nums[4] = 5)

No physical remove but naturally following for loop auto increase index 'i' from 2 to 3

the '5' at index 3 implicitly removed, or in reversed perspective, we didn't remove

the '5' at index 3 but remove the '5' at index 4 and then forwarding '5' at index 3

to index 4 to fill in the gap left by removing the '5' at index 4, delete\_count++ = 2,

and following comparison dealing with concreate subarray as below:

index: . . . 3 4 5 6 7 8

nums: . . . 5 5 8 8 8 8

i = 4, delete\_count = 2, because nums[i](nums[4] = 5) != nums[i + 1](nums[5] = 8)

Just additionally increase index again from 4 to 5 to skip the non-duplicate nums[5](compare with nums[4])

index: 0 1 2 3 4 5 6 7 8

Delete3: x 2 5 x 5 8 x 8 8

For loop auto increase 'i' from 5 to 6

i = 6, delete\_count = 2, because nums[i](nums[6] = 8) == nums[i + 1](nums[7] = 8)

No physical remove but naturally following for loop auto increase index 'i' from 5 to 6

the '8' at index 6 implicitly removed, or in reversed perspective, we didn't remove

the '8' at index 6 but remove the '8' at index 7 and then forwarding '8' at index 6

to index 7 to fill in the gap left by removing the '8' at index 7, delete\_count++ = 3,

and following comparison dealing with concreate subarray as below:

index: . . . . . . . 7 8

nums: . . . . . . . 8 8

index: 0 1 2 3 4 5 6 7 8

Delete4: x 2 5 x 5 8 x x 8

For loop auto increase 'i' from 6 to 7

i = 7, delete\_count = 3, because nums[i](nums[7] = 8) == nums[i + 1](nums[8] = 8)

No physical remove but naturally following for loop auto increase index 'i' from 6 to 7

the '8' at index 7 implicitly removed, or in reversed perspective, we didn't remove

the '8' at index 8 but remove the '8' at index 7 and then forwarding '8' at index 7

to index 8 to fill in the gap left by removing the '8' at index 8, delete\_count++ = 4,

and no more following comparison required

1 more delete required: 2 5 5 8 8 -> 2 5 5 8 make it even length

**Refer to**

<https://algo.monster/liteproblems/2216>

**Problem Description**

You are given an integer array nums with 0-based indexing. To be considered **beautiful**, the array must have the following properties:

nums.length is an even number.

nums[i] is not equal to nums[i + 1] for all i where i % 2 == 0.

An empty array also meets the criteria for being beautiful.

Your task is to remove the minimum number of elements from nums to make it beautiful. When you remove an element, all elements on the right shift one position to the left to fill the void, while the ones on the left stay unchanged.

You must return the smallest number of elements that need to be deleted for nums to become beautiful.

**Intuition**

The key to solving this problem involves a two-part approach. We need to:

Identify and remove elements to ensure all even-indexed positions have different values than their immediate next (odd-indexed) neighbors.

Ensure that after the above deletion, the length of the array is even.

We can achieve the first goal by iterating through the array and comparing the element at each even index i with its next element i + 1.

**If they are the same, we increment a counter (ans), representing the number of deletions required, and then move onto the next index.**

**If they are different, we simply step over both indices and continue checking the subsequent pairs. By incrementing the counter only when necessary, we are effectively "deleting" elements to satisfy the pairing requirement.**

The second goal is to check if the length of the modified array (original length minus total deletions) is even. If not, we must remove one more element to meet the criteria for a beautiful array. This check is conducted after the pairing loop with a simple modulo operation.

Combining these steps ensures that we return the minimum number of deletions necessary to make nums beautiful.

**Solution Approach**

The implementation of the provided solution follows a straightforward index-based iterative approach to attain the desired beautiful array. Here is a detailed explanation of the algorithm:

We initialize two variables, n to hold the length of the array nums, and ans to keep track of the count of deletions, which is initially set to 0.

An index variable i is also initialized to 0, which is used to traverse the array.

The iterative process begins and continues while i is less than n - 1 because we always need to check the current element and its next element, and hence we can't process the last element if i were equal to n - 1.

Within the loop:

We first check if the current element nums[i] is the same as the next element nums[i + 1]. If they are the same:

We increment ans, since we would need to remove one of these elements to fulfill the condition where even-indexed elements (i % 2 == 0) must not be equal to their immediate next elements.

We then increment i by 1 to move past the duplicate element.

If the current element nums[i] is not the same as nums[i + 1], then no deletion is required here, and we increment i by 2 to skip both elements and move to the next pair.

After iterating through the array, we need to perform a final check to make sure the resulting array has an even number of elements. We calculate the potential new length of nums by subtracting the count of deletions (ans) from the original length (n). If this number is odd:

We increment ans by 1, implying one more element needs to be deleted to ensure even length.

The ans variable, which records the minimum number of deletions needed, is then returned.

Important to note is that there are no additional data structures required, and this solution is built upon simple variables and conditional logic, showcasing an in-place approach with a linear time complexity of O(n), where n is the number of elements in the array. This efficiency is due to the single-pass loop over the array elements.

**Solution Implementation**

class Solution {

/\*\*

\* This method finds the minimum number of deletions required to make the

\* array beautiful. An array is beautiful if it has an even length and

\* no two adjacent elements are equal.

\*

\* @param nums Array of integers to be made beautiful.

\* @return The minimum number of deletions required.

\*/

public int minDeletion(int[] nums) {

int arrayLength = nums.length; // Total length of the input array.

int deletionsNeeded = 0; // Counter for the required deletions.

// Iterate through the array to find pairs of equal adjacent elements.

for (int i = 0; i < arrayLength - 1; ++i) {

if (nums[i] == nums[i + 1]) {

// Increment the count if a pair of equal adjacent elements is found.

++deletionsNeeded;

} else {

// Skip the next element if the current and next elements are not equal.

++i;

}

}

// Check if the length of the array after deletions is even.

// If it's odd, we need an additional deletion to make the length even.

if ((arrayLength - deletionsNeeded) % 2 == 1) {

++deletionsNeeded;

}

// Return the total number of deletions needed to make the array beautiful.

return deletionsNeeded;

}

}

**Time and Space Complexity**

**Time Complexity**

The given Python code iterates over the list nums once. During each iteration, it either increments i by one if the current element and the next element are the same, or increments i by two if they are different. In the worst case scenario (no two adjacent elements are the same), i will be incremented by two for each check, except possibly the last element, resulting in n/2 checks.

Despite this, the complexity is not reduced and the algorithm processes each element at most once. Therefore, regardless of the pattern of elements in nums, the time complexity is O(n), where n is the number of elements in nums.

**Space Complexity**

The space complexity of the algorithm is O(1). It only uses a fixed number of integer variables (n, i, ans) that do not depend on the size of the input list nums. There is no additional space used that grows with the input size. The input list itself is not modified, so the space complexity remains constant.

**Refer to**

[L1647.Minimum Deletions to Make Character Frequencies Unique](note://WEB8a683715985e8e8ffa6c1edb2a9f6273)