2216. Minimum Deletions to Make Array Beautiful

Medium Oracle Grocery Array

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:

1. Identify and remove elements to ensure all even-indexed positions have different values than their immediate next (odd-indexed) neighbors.
2. 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.

Example Walkthrough

Let's consider the example array `nums = [1, 1, 2, 3, 3, 4]` and walk through the solution approach.

- Initialize `n` to 6 (length of `nums`) and `ans` to 0.
- Set the traversal index `i` to 0.

As we iterate:

- For `i = 0`: `nums[0]` is 1 and `nums[0 + 1]` is also 1.
  - They are the same, so we must increment `ans` to 1, and `i` to 1.
- Now, `i = 1`: we skip the check here because it's an odd index and increment `i` to 2.
- For `i = 2`: `nums[2]` is 2 and `nums[2 + 1]` is 3.
  - They are different, so we move on to the next even index by incrementing `i` to 4.
- For `i = 4`: `nums[4]` is 3 and `nums[4 + 1]` is 4.
  - They are different, so we increment `i` to 6 and exit the loop (since `i` is not less than `n - 1`).

After completing the loop, we do the final check on the length of the array after deletions:

- `n - ans` would give us `6 - 1 = 5`, an odd number.
- Because it's not even, we increment `ans` by 1 to make it 2.

The array `nums` could be made beautiful by removing 2 elements. The final answer, `ans`, is 2.

The elements that would be removed are the first `1` to resolve the duplicate at the start, and any single element from the remaining ones to ensure the array length is even. Thus, the minimum number of elements to remove to make `nums` beautiful is `2`.

Solution Implementation

Python

```python
class Solution:
    def minDeletion(self, nums: List[int]) -> int:
        # Initialize the length of the input list
        length = len(nums)

        # Initialize the index and the counter for the number of deletions
        index = deletions_count = 0

        # Loop through the list while the current index is less than the index of the last pair
        while index < length - 1:
            # If the current element is the same as the next one,
            # it violates the alternating condition, so we need to delete one of them
            if nums[index] == nums[index + 1]:
                # Increment the deletion count
                deletions_count += 1
                # Move to the next element
                index += 1
            # If the current element is different from the next one,
            # we can just move to the next pair
            else:
                index += 2

        # If the updated length, after deletions, is still odd, we need to remove one more element
        if (length - deletions_count) % 2:
            deletions_count += 1

        # Return the total number of deletions
        return deletions_count
```

Java

```java
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;
    }
}
```

C++

```cpp
class Solution {
public:
    int minDeletion(vector<int>& nums) {
        // Initialize the length of the input vector.
        int length = nums.size();

        // This variable will store the minimum number of deletions required.
        int minDeletionsRequired = 0;

        // Iterate over the array elements.
        for (int i = 0; i < length - 1; ++i) {
            // If the current element is equal to the next one, we need
            // to delete one of them.
            if (nums[i] == nums[i + 1]) {
                ++minDeletionsRequired;
            } else {
                // If they are not equal, skip the next element as it
                // should form a pair with its subsequent element.
                ++i;
            }
        }

        // After removing pairs, if the array has an odd number of elements,
        // delete one more element to make the length even.
        if ((length - minDeletionsRequired) % 2 == 1) {
            ++minDeletionsRequired;
        }

        // Return the total number of deletions required.
        return minDeletionsRequired;
    }
};
```

TypeScript

```typescript
function minDeletion(nums: number[]): number {
    // Initialize the length of the nums array.
    const length = nums.length;

    // Initialize the variable to count the number of deletions required.
    let deletionCount = 0;

    // Initialize an index counter to iterate through the array.
    let index = 0;

    // Loop over the array elements, excluding the last one,
    // since we check the current and next element in each iteration.
    while (index < length - 1) {
        // If the current and next elements are the same, increment index by 1
        // to skip the current element and increment deletionCount.
        if (nums[index] === nums[index + 1]) {
            index++;
            deletionCount++;
        } else {
            // If they're different, move forward by 2 as we need pairs of distinct elements.
            index += 2;
        }
    }

    // Check if the array length after deletions is odd.
    // If so, more deletions are needed to make the length even.
    if ((length - deletionCount) % 2 === 1) {
        deletionCount++;
    }

    // Return the final number of deletions required.
    return deletionCount;
}
```

```python
class Solution:
    def minDeletion(self, nums: List[int]) -> int:
        # Initialize the length of the input list
        length = len(nums)

        # Initialize the index and the counter for the number of deletions
        index = deletions_count = 0

        # Loop through the list while the current index is less than the index of the last pair
        while index < length - 1:
            # If the current element is the same as the next one,
            # it violates the alternating condition, so we need to delete one of them
            if nums[index] == nums[index + 1]:
                # Increment the deletion count
                deletions_count += 1
                # Move to the next element
                index += 1
            # If the current element is different from the next one,
            # we can just move to the next pair
            else:
                index += 2

        # If the updated length, after deletions, is still odd, we need to remove one more element
        if (length - deletions_count) % 2:
            deletions_count += 1

        # Return the total number of deletions
        return deletions_count
```

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.