2870. Minimum Number of Operations to Make Array Empty

Medium Greedy Array Hash Table Counting

Leetcode Link

Problem Description

You're provided with an array of positive integers, each of which has an index starting from 0. To make the array empty, you can perform two kinds of operations any number of times:

- Select two elements with the same value and remove them from the array.
- Select three elements with the same value and remove them from the array.

The goal is to find the minimum number of these operations needed to empty the entire array. If it's impossible to empty the array using these operations, the answer should be -1.

Intuition

To solve this problem efficiently, we need a way to keep track of how many times each element occurs in the array. A **hash table** is perfect for this, as it allows us to count the occurrences of each element quickly. Once we know the count of each element, we can use a **greedy approach** to solve the problem.

Using the greedy method, for each unique element in the array, we repeatedly perform the deletion operation that removes the most elements. Since removing three elements is better than removing two to minimize operations, we prefer to perform the operation of removing three elements whenever possible. If an element occurs only once, it's impossible to perform either operation, so we immediately know we must return -1.

For any element occurring a number of times greater than one, we can perform the deletion operation as many times as this formula allows: $\lfloor \frac{c+2}{3} \rfloor$, where c is the number of occurrences. This formula ensures we always perform the optimal number of three-element deletions while accounting for leftovers that might only allow for a two-element deletion.

Finally, we sum up the operations for all elements to get the minimum number of operations required. If any element's count is one, we return -1. If not, we return the total sum of operations.

Solution Approach

The solution implements a **hash table** using Python's `Counter` from the `collections` module, which creates a dictionary where each key is a unique element from the array, and its corresponding value is the count of how many times that element appears in the array.

The next step in the solution is to iterate over each unique element's count in the hash table. The iteration checks the count for each element:

- If an element appears exactly once, (`c == 1`), the function immediately returns `-1`, since it's impossible to perform either of the two allowed deletion operations.
- For other cases, where the count `c` is greater than one, it calculates the minimum number of operations needed to delete each element using the formula `(c + 2) // 3`. The addition of `2` before the integer division by `3` effectively rounds up to the nearest whole number that isn't greater than `c / 3`, allowing us to make the maximal use of the operation that removes three elements at a time.

After processing all elements, the solution sums up all individual minimum operation counts and returns the total as the answer. This way, it ensures the use of the most efficient deletions while coping with differing element counts.

In summary, the algorithm makes intelligent use of a hash table to count occurrences and applies a greedy strategy by maximizing the use of the more efficient three-element deletion where possible, and then taking the leftover element pairs—if any—for two-element deletions.

Example Walkthrough

Let us walk through an example to illustrate the solution approach:

Assume we are given the following array of positive integers: `[3, 3, 3, 3, 3, 1, 1, 2, 2, 2, 2]`.

Step 1: Count the occurrences of each unique element using a hash table (Counter in Python).

- The counts would be: `{3: 5, 1: 2, 2: 5}`.

Step 2: Evaluate the possibility of removing the elements using the described operations:

- For element `3`, the count is `5`. According to our formula, the minimum number of operations needed is `(5 + 2) // 3 = 7 // 3 = 2`. We can remove three '3s' in one operation and the remaining two '3s' in another operation.
- For element `1`, the count is `2`. We can remove both '1s' using one operation of the second kind, as `(2 + 2) // 3 = 4 // 3 = 1`. If instead there was only one '1', we would have had to return `-1` since it's impossible to remove a single occurrence.
- For element `2`, analogous to element `3`, the count is `5`. So, taking the formula `(5 + 2) // 3`, we get `7 // 3 = 2` operations for element `2` as well.

Step 3: Sum up all of the operations required.

- The total minimum number of operations would be `2 (for 3s) + 1 (for 1s) + 2 (for 2s) = 5`.

Therefore, for the given array `[3, 3, 3, 3, 3, 1, 1, 2, 2, 2, 2]`, the minimum number of operations required to empty the entire array is `5`.

Python Solution

```python
from collections import Counter

class Solution:
    def minOperations(self, nums: List[int]) -> int:
        # Create a counter object to count occurrences of each number in the list
        num_counts = Counter(nums)

        # Initialize operations counter to 0
        operations = 0

        for count in num_counts.values():
            # If any number occurs only once, it's impossible to form a sequence,
            # so return -1 according to the problem statement.
            if count == 1:
                return -1

            # Calculate the minimum number of operations needed to form a sequence
            # Each operation can decrease a number by either 1 or 2.
            # So the formula "(count + 2) // 3" is used to find the minimum operations
            # for each group of identical numbers, assuming the best action is taken.
            operations += (count + 2) // 3

        # Return the total number of operations
        return operations
```

Java Solution

```java
class Solution {
    public int minOperations(int[] nums) {
        // Create a map to store the frequencies of each number
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        // Populate the map with the frequencies
        for (int num : nums) {
            frequencyMap.merge(num, 1, Integer::sum);
        }

        // Initialize operations counter
        int operations = 0;

        // Iterate over the values in the frequency map
        for (int frequency : frequencyMap.values()) {
            // If the frequency is less than 2, it's not possible to perform operations.
            if (frequency < 2) {
                return -1;
            }
            // Calculate the remainder and quotient of dividing the frequency by 3
            int remainder = frequency % 3;
            int quotient = frequency / 3;

            // Use switch expression to determine the number of operations needed
            switch (remainder) {
                // If there is no remainder, the number of operations is equal to the quotient
                case 0 -> operations += quotient;
                // For any remainder, an additional operation is needed
                default -> operations += quotient + 1;
            }
        }

        // Return the total numberOfOperations required
        return operations;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>

class Solution {
public:
    // Function to find the minimum number of operations required
    int minOperations(std::vector<int>& nums) {
        // Create a hash map to store the frequency of each number
        std::unordered_map<int, int> frequencyMap;

        // Count the frequency of each number in the vector
        for (int num : nums) {
            ++frequencyMap[num];
        }

        int operations = 0; // Initialize the number of operations to 0

        // Iterate through the frequency map
        for (auto& keyValue : frequencyMap) {
            int count = keyValue.second;  // Get the frequency count

            // If there's a number with less than 2 occurrences, we cannot perform the operation
            if (count < 2) {
                return -1;
            }

            // Find the minimum number of operations required
            // The number of operations for each number is (count + 2) / 3
            operations += (count + 2) / 3;
        }

        // Return the total number of operations needed
        return operations;
    }
};
```

Typescript Solution

```typescript
function minOperations(nums: number[]): number {
    // A map to store the frequency of each number in the `nums` array.
    const frequencyMap: Map<number, number> = new Map();

    // Populate the frequencyMap with the frequency of each number in the `nums` array.
    for (const num of nums) {
        frequencyMap.set(num, (frequencyMap.get(num) ?? 0) + 1);
    }

    // Initialize the minimum number of operations required.
    let minOperationsRequired = 0;

    // Iterate through the frequencyMap to calculate the total operations.
    for (const frequency of frequencyMap.values()) {
        // If any number occurs only once, it's not possible to form a strictly increasing sequence.
        if (frequency < 2) {
            return -1;
        }

        // Calculate the number of operations needed for the current frequency and add to the total.
        // The operation is equivalent to divide by 3, round down, which is achieved by bitwise OR with 0.
        minOperationsRequired += (Math.floor((frequency + 2) / 3));
    }

    // Return the minimum number of operations required to make the `nums` array strictly increasing.
    return minOperationsRequired;
}
```

Time and Space Complexity

The given Python code snippet defines a function `minOperations` that aims to determine the minimum number of operations required to ensure that every number appears more than once in the given list `nums`.

Time Complexity

The time complexity of the function is `O(n)` where `n` is the length of the array `nums`. This is because the function iterates over the elements in `nums` just once in order to create the `count` dictionary using `Counter(nums)`, which is `O(n)`, and then iterates over the values of the `count` dictionary to calculate `ans`. In the worst case, all elements in `nums` are unique, meaning the count dictionary also has `n` entries and thereby iterating over the `count.values()` would take `O(n)`.

Space Complexity

The space complexity of the function is `O(n)`. This is due to the use of `Counter(nums)` which can potentially store each unique element from `nums` as a key, leading to a space usage proportional to the number of unique elements, which, in the worst case, can be as large as `n`.