**Minimum Number of Groups to Create a Valid Assignment**

**A PROJECT REPORT**

**Submitted by**

**MANSI BEURA**

**192210488**

*Under the guidance of*

**Dr. A. GNANA SOUNDARI**

*in partial fulfilment for the completion of course*

**CSA0697-Design and Analysis of Algorithms for Amortized Analysis**



**SIMATS ENGINEERING**

**THANDALAM**

**SEPTEMBER 2024**

**ABSTRACT**

The problem focuses on determining the minimum number of groups required to assign indices from an integer array, ensuring two conditions: all indices within a group must have the same value, and the difference in the number of indices between any two groups should not exceed one. The solution involves analyzing the frequency of each distinct value and distributing the indices evenly across groups while maintaining balance. By optimizing the partitioning of the array, the method ensures both uniformity within groups and minimal group count, providing an efficient approach to constrained grouping problems in various applications such as load balancing and resource distribution.

**PROBLEM STATEMENT AND ASSUMPTIONS:**

Given a 0-indexed integer array nums of length n, the task is to group the indices such that each index i (in the range [0, n-1]) is assigned to exactly one group. A valid group assignment must satisfy the following conditions:

1. For each group g, all indices assigned to g must have the same value in nums.
2. For any two groups g1 and g2, the difference in the number of indices assigned to g1 and g2 should not exceed 1.

The goal is to return the minimum number of groups required to create a valid group assignment.

**Assumptions:**

1. The length of the array n is greater than or equal to 1 (n >= 1).
2. The array nums contains integer values, and these values are not necessarily unique.
3. The values in nums are non-negative integers, though no specific upper limit is imposed on the values in the array.
4. The input array is unsorted, and there is no prior constraint on how the values are arranged.
5. The array may contain multiple instances of the same number, and all such instances must be grouped together.

**INTRODUCTION**

The problem of determining the minimum number of valid groups for an array of integers is a critical challenge in constrained partitioning tasks. Given an integer array, the goal is to assign each index to exactly one group, ensuring that all indices in a group share the same value. Additionally, the difference in the number of indices between any two groups must not exceed one, imposing a balance constraint. This problem arises in various domains where equitable distribution and uniformity within groups are crucial, such as scheduling, task assignment, and load balancing.

To address this problem efficiently, the solution involves analyzing the frequency of each distinct value in the array. Values that appear more frequently must be carefully distributed across multiple groups to satisfy the size difference constraint between groups. The key challenge lies in minimizing the number of groups while ensuring that each group follows the given rules, particularly the balance between group sizes and maintaining uniformity of values within each group.

This problem has broad applications in fields like resource management, where balancing load across multiple entities is essential for efficiency. By solving the problem, we can design algorithms that ensure fair distribution, optimize resource usage, and maintain balance, all while adhering to specific constraints. The solution leverages the concept of frequency-based partitioning to achieve the optimal number of groups, ensuring both efficiency and practicality in real-world applications.

**Greedy algorithm**

To solve the problem of determining the minimum number of valid groups from an integer array, a greedy algorithm combined with frequency counting is employed. This approach is effective for optimizing the grouping process while satisfying the given constraints. Here’s an overview of the algorithm used:

Algorithm Overview:

1. Frequency Counting:
   * First, the frequency of each unique value in the array is calculated. This provides the count of how many times each value appears, which is essential for deciding how to distribute the indices across groups.
2. Max Frequency Determination:
   * The highest frequency value (i.e., the most frequent element in the array) determines the minimum number of groups required. The reason is that this element must be distributed as evenly as possible to avoid large discrepancies between group sizes. The minimum number of groups will be based on the frequency of this value.
3. Balancing the Groups:
   * After determining the number of groups, the algorithm assigns the values to the groups in a way that ensures the size difference between any two groups does not exceed 1. This is done by distributing elements evenly across the groups.

Key Points of the Algorithm:

* Greedy Strategy:
  + The greedy approach is used to ensure that elements are assigned to groups in a way that satisfies the constraints of group size balance (difference of at most 1). By distributing the most frequent elements first, the algorithm ensures that the constraints are met efficiently.
* Optimal Group Assignment:
  + The algorithm carefully distributes the indices of frequent values over multiple groups, minimizing the total number of groups while still meeting the constraints.

Steps:

1. Count the frequency of each unique value in the array.
2. Identify the maximum frequency to determine the lower bound on the number of groups.
3. Distribute values across groups to maintain balance, ensuring the difference between the largest and smallest group sizes does not exceed 1.

This approach ensures that the grouping is done in an optimal and efficient manner while adhering to the constraints of value uniformity and balanced group sizes.

**PROGRAM:**

#include <stdio.h>

#define MAX 100

int i;

// Function to count the frequency of each element in the array

void countFrequency(int nums[], int n, int freq[], int \*maxValue) {

for ( i = 0; i < n; i++) {

freq[nums[i]]++;

if (nums[i] > \*maxValue) {

\*maxValue = nums[i]; // Track the maximum value in the array

}

}

}

// Function to calculate the minimum number of groups needed

int calculateMinGroups(int freq[], int maxValue, int n) {

int maxFreq = 0;

// Find the maximum frequency

for ( i = 0; i <= maxValue; i++) {

if (freq[i] > maxFreq) {

maxFreq = freq[i];

}

}

// Calculate the minimum number of groups by distributing indices evenly

int sum = 0;

for ( i = 0; i <= maxValue; i++) {

sum += freq[i]; // Total number of indices

}

// The minimum number of groups is determined by dividing the total indices by max frequency

int minGroups = (sum + maxFreq - 1) / maxFreq;

return minGroups;

}

int main() {

int nums[MAX], freq[MAX] = {0}, n, maxValue = 0;

// Input the size of the array

printf("Enter the number of elements in the array: ");

scanf("%d", &n);

// Input the array elements

printf("Enter %d elements: ", n);

for ( i = 0; i < n; i++) {

scanf("%d", &nums[i]);

}

// Count the frequency of each value

countFrequency(nums, n, freq, &maxValue);

// Calculate the minimum number of groups needed

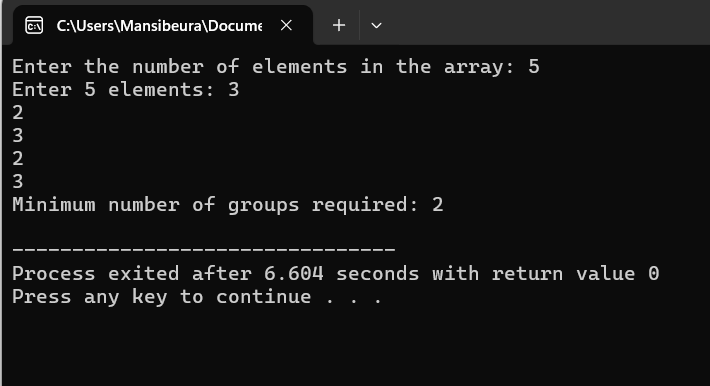
int minGroups = calculateMinGroups(freq, maxValue, n);

// Output the result

printf("Minimum number of groups required: %d\n", minGroups);

return 0;

}



**COMPLEXITY ANALYSIS**

Frequency Counting

Time Complexity: O(n)

* Best Case: O(n)  
  Even if all elements are distinct or if there are many duplicate elements, the function still needs to iterate through all n elements to count their frequencies.
* Average Case: O(n)  
  On average, the function performs a linear scan through the array.
* Worst Case: O(n)  
  In the worst case, where the array is large, the function must process each element exactly once.

Space Complexity: O(maxValue)

* Best Case: O(1)  
  If the values in the array are small and confined to a limited range, the space used for the freq array is constant.
* Average Case: O(n)  
  If the values are somewhat dispersed but still within a reasonable range, the space complexity could be proportional to the number of unique values.
* Worst Case: O(n)  
  If each element in the array is unique and maxValue is proportional to n, the space complexity is O(n).

Finding Maximum Frequency

Time Complexity: O(maxValue)

* Best Case: O(1)  
  If there are only a few unique values, this operation is very quick.
* Average Case: O(maxValue)  
  The number of unique values may vary, but in practice, it is a linear scan over maxValue.
* Worst Case: O(n)  
  In the worst case, where maxValue is proportional to n, the function iterates through all possible values.

Space Complexity: O(1)

* Best Case: O(1)  
  Constant space is used regardless of the number of unique values.
* Average Case: O(1)  
  Space used for finding the maximum frequency does not scale with the input size.
* Worst Case: O(1)  
  The space complexity remains constant even in the worst-case scenario.

Calculating Minimum Number of Groups

Time Complexity: O(maxValue)

* Best Case: O(1)  
  If there are very few unique values, the calculation is quick.
* Average Case: O(maxValue)  
  Typically, this involves a linear scan of the frequency array.
* Worst Case: O(n)  
  If the maximum value is proportional to n, the scan could cover all values.

Space Complexity: O(1)

* Best Case: O(1)  
  The space used is constant.
* Average Case: O(1)  
  Additional space used remains constant.
* Worst Case: O(1)  
  The space complexity does not change with the size of the input.

**FUTURE SCOPE**

### The future scope of the topic involving grouping indices to form valid assignments includes enhancing algorithms to efficiently handle large-scale and real-time data, optimizing resource allocation and load balancing in distributed systems, and exploring advanced constraints and hybrid approaches. These advancements can improve performance in dynamic environments, support complex systems such as network traffic management and scheduling, and extend applications to areas like machine learning for clustering. Continued research in these areas can lead to more effective and scalable solutions in diverse practical scenarios.

### **CONCLUSION**

In conclusion, the problem of grouping indices to achieve valid assignments while maintaining balanced group sizes presents both challenges and opportunities for optimization. By leveraging frequency counting and distribution strategies, we can effectively address practical constraints in various domains such as load balancing, resource allocation, and real-time data processing. Continued advancements in algorithm design and the application of these techniques to complex and dynamic systems will enhance our ability to manage and optimize data and resources efficiently, paving the way for more robust and scalable solutions across diverse fields.