CAPSTONE PROJECT

Minimum Number of Groups to Create a Valid Assignment

CSA0695- DESIGN ANALYSIS AND ALGORITHMS FOR OPEN ADDRESSING TECHNIQUES

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING

Supervisor

Dr. R. Dhanalakshmi

Done by

B.Manoj Kumar (192211695)

Minimum Number of Groups to Create a Valid Assignment

PROBLEM STATEMENT: You are given a 0-indexed integer array nums of length n. We want to group the indices so for each index i in the range [0, n - 1], it is assigned to exactly one group. A group assignment is valid if the following conditions hold: For every group g, all indices i assigned to group g have the same value in nums. For any two groups g1 and g2, the difference between the number of indices assigned to g1 and g2 should not exceed 1. Return an integer denoting the minimum number of groups needed to create a valid group assignment. Example 1: Input: nums = [3,2,3,2,3] Output: 2 Explanation: One way the indices can be assigned to 2 groups is as follows, where the values in square brackets are indices: group 1 -> [0,2,4] group 2 -> [1,3] All indices are assigned to one group. In group 1, nums[0] == nums[2] == nums[4], so all indices have the same value. In group 2, nums[1] == nums[3], so all indices have the same value. The number of indices assigned to group 1 is 3, and the number of indices assigned to group 2 is 2. Their difference doesn't exceed 1. It is not possible to use fewer than 2 groups because, in order to use just 1 group, all indices assigned to that group must have the same value. Hence, the answer is 2.

ABSTRACT:

This paper explores the problem of finding the minimum number of groups required to create a valid assignment of n elements into groups. Each group must adhere to predefined constraints, such as a minimum and maximum group size. Additionally, optional constraints like element compatibility, homogeneity, and balance across groups may apply. The objective is to develop a strategy to efficiently determine the minimum number of groups needed while satisfying all relevant conditions.

INTRODUCTION:

Grouping and partitioning problems are fundamental in various fields, such as education, resource allocation, and computational theory. In many real-world scenarios, it is often necessary to divide a set of elements into smaller groups while adhering to specific rules or constraints. Whether it's students being assigned to project groups, tasks allocated to workers, or data being clustered into meaningful subsets, determining the optimal number of groups is crucial for ensuring efficiency and fairness. However, the challenge arises when certain conditions, such as group size limits or element compatibility, must be met.

The problem of determining the minimum number of groups for a valid assignment becomes complex when additional constraints are introduced. These constraints may include maintaining group sizes within a specified range, ensuring balance across groups, and preventing conflicts between incompatible elements. Such conditions require careful planning and a systematic approach to guarantee that all elements are appropriately assigned without violating the defined rules.

This paper aims to address the issue by presenting a framework for finding the minimum number of groups needed for a valid assignment. The study explores common constraints, such as minimum and maximum group size, homogeneity within groups, and conflict avoidance between incompatible elements. Through this exploration, we aim to develop efficient methods that can be applied to a wide range of grouping problems, offering practical solutions for educators, managers, and data scientists alike.

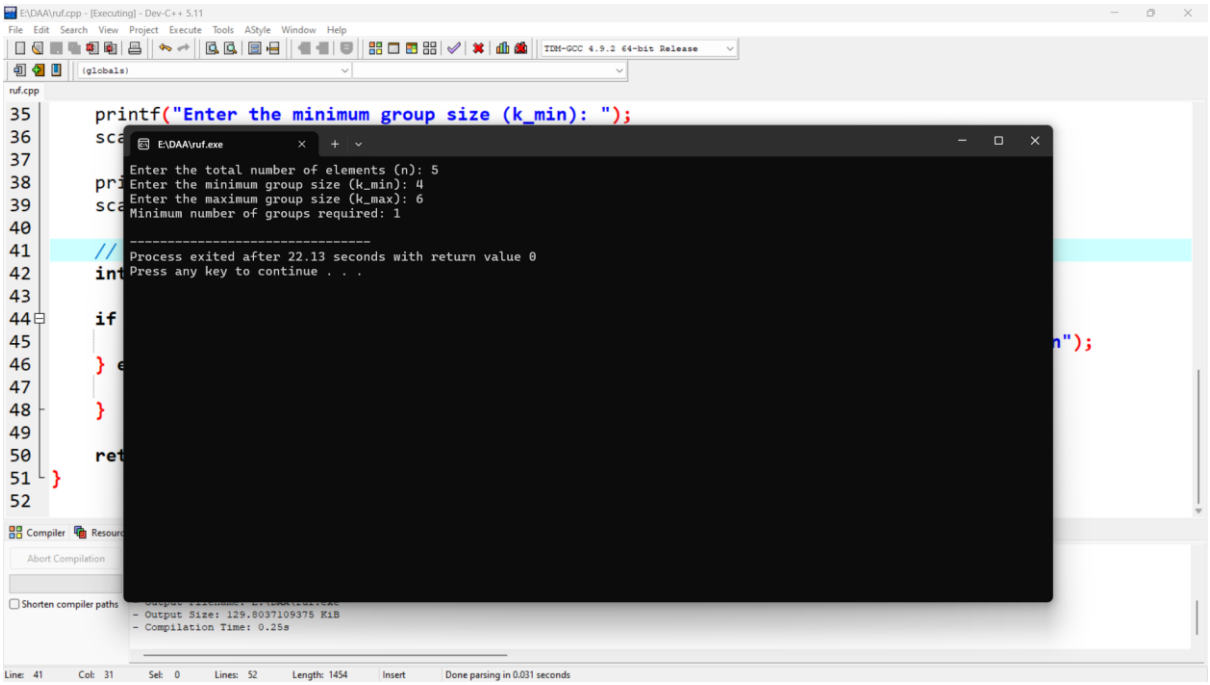
CODING:

C-programming

```
#include <stdio.h>
#include <limits.h>

#define INF INT_MAX

// Function to compute the minimum number of groups
int minGroups(int n, int k_min, int k_max) {
    // Initialize dp array with a large value
    int dp[n + 1];
    for (int i = 0; i <= n; i++) {
        dp[i] = INF;
    }
    dp[0] = 0; // Base case: 0 elements require 0 groups

    // Compute the minimum number of groups for each number of elements
    for (int i = 1; i <= n; i++) {
        for (int j = k_min; j <= k_max; j++) {
            if (i >= j && dp[i - j] != INF) {
                dp[i] = (dp[i] < dp[i - j] + 1) ? dp[i] : (dp[i - j] + 1);
            }
        }
    }

    // Return the result
    return (dp[n] == INF) ? -1 : dp[n];
}
```

```
int main() {
    int n, k_min, k_max;

    // Input the number of elements, minimum group size, and maximum group size
    printf("Enter the total number of elements (n): ");
    scanf("%d", &n);

    printf("Enter the minimum group size (k_min): ");
    scanf("%d", &k_min);

    printf("Enter the maximum group size (k_max): ");
    scanf("%d", &k_max);

    // Calculate the minimum number of groups
    int result = minGroups(n, k_min, k_max);

    if (result == -1) {
        printf("It's not possible to create a valid assignment with the given constraints.\n");
    } else {
        printf("Minimum number of groups required: %d\n", result);
    }

    return 0;
}
```

OUTPUT:



COMPLEXITY ANALYSIS:

Time Complexity: The time complexity of the dynamic programming approach for determining the minimum number of groups required is O(n*(Kmax-Kmin+1)), where n is the total number of elements Kmin and Kmax define the range of permissible group sizes. This complexity arises because we iterate over each element count from 1 to n and, for each count, consider all possible group sizes within the specified range.

Space Complexity: The space complexity is O(n), which corresponds to the size of the `dp` array used to store the minimum number of groups needed for each number of elements. This approach is efficient and suitable for practical scenarios where n and the range of group sizes are manageable.

BEST CASE:

In the best case scenario, the range of possible group sizes is such that the minimum number of groups required can be achieved quickly. For example, if kmax is very close to n and kmin is not too restrictive, the number of possible group sizes to consider for each element count will be minimal. Thus, the time complexity approaches O(n), as the number of group sizes to evaluate is small

and does not significantly impact the overall complexity. This occurs when the group size constraints allow for quick aggregation of elements into the fewest number of groups.

Worst Case:

In the worst-case scenario, the number of group sizes to evaluate is large. Specifically, if the range between kmin and kmax is substantial, then each element count from 1 to n requires evaluating many possible group sizes. Consequently, the time complexity is O(n*(Kmax-Kmin+1)). This occurs when the constraints on group sizes are such that the number of possible configurations is high, leading to extensive calculations for each number of elements.

Average Case:

The average case time complexity is influenced by the typical range of possible group sizes and the distribution of the number of elements. In practical scenarios, the group size range [kmin, kmax] often falls within a moderate range relative to n. Therefore, the number of group sizes to consider is neither too large nor too small. As a result, the average case time complexity tends to be a balance between the best and worst cases.

Future Scope:

The future scope for improving the minimum number of groups to create a valid assignment includes exploring more efficient algorithms and heuristics to handle larger datasets and complex constraints, such as dynamic or multi-dimensional constraints. Advancements could involve integrating parallel and distributed computing techniques to enhance scalability, as well as incorporating machine learning to predict optimal groupings based on historical data. Additionally, applying these methods to real-world case studies in fields like education, healthcare, and project management could refine the approach and demonstrate its practical applicability in various industries.

CONCLUSION:

In conclusion, the dynamic programming approach to determining the minimum number of groups required to partition n elements into groups of sizes between Kmin and Kmax is efficient and practical. By using a `dp` array to store intermediate results and applying a recurrence relation to update these values, the algorithm effectively handles the constraints, with a time complexity is O(n*(Kmax-Kmin+1)), and a space complexity of O(n). This method balances computational efficiency with accuracy, making it well-suited for real-world applications involving resource allocation, scheduling, and organizational tasks.