## SIMATS SCHOOL OF ENGINEERING

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

**CSA0603-Design and Analysis of Algorithms for Vertex Cover Problem**

**“Maximum Number of Groups with increasing length”**

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfilment for the award of the degree of*

**Bachelor of Engineering**

**in**

**Computer Science Engineering**

**Submitted by**

**M. Varshith [192211780]**

Under the Supervision of

**Dr. K. V. KANIMOZHI**

SEPTEMBER 2024

**DECLARATION**

I, M. Varshith, student of Bachelor of Engineering in Computer Science Engineering at Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled "Maximum Number of Groups with increasing length" is the outcome of my own bonafide work. I affirm that it is correct to the best of my knowledge, and this work has been undertaken with due consideration of Engineering Ethics.

M. Varshith-(192211780)

Date:

Place: Saveetha School of Engineering, Thandalam

**CERTIFICATE**

This is to certify that the project entitled “Maximum Number of Groups with increasing length” submitted by M. Varshith (192211780) has been carried out under my supervision. The project has been submitted as per the requirements in the current semester of B.E Computer science engineering

Faculty-in-charge

Dr. K.V.KANIMOZHI

**ABSTRACT**

The problem of forming the maximum number of distinct groups with increasing lengths arises in scenarios where a limited set of resources must be divided into subsets while adhering to certain constraints. Given an array `usageLimits` of length `n`, the task is to create groups from numbers 0 to `n-1`, ensuring that each number is used no more than the allowed times specified in `usageLimits`. Additionally, each group must consist of distinct numbers, and the length of each group must be strictly greater than the previous one. The challenge lies in balancing the limited usage of each number while maximizing the number of valid groups.

A greedy algorithmic approach is typically employed to solve this problem. The algorithm begins by forming the smallest possible group and gradually increases the group size, ensuring that the conditions of distinctness and increasing length are met. Each time a group is formed, the available usage of the numbers involved is decremented, preventing them from being overused in future groups. The process continues until it becomes impossible to form a group of the required size due to the exhaustion of usable numbers or exceeding the allowed usage limits.

One of the key complexities of the problem is handling edge cases. For instance, certain elements in the array `usageLimits` may be zero, which means those numbers cannot be used in any group. Additionally, when the input array is very small, the possible number of groups is constrained by the limited number of elements, while for large arrays with high usage limits, the algorithm must efficiently track and manage the usage of each element. These variations require careful consideration to ensure the algorithm is both correct and optimal.

**KEYWORDS:**

 Distinct groups

 Increasing lengths

 Usage limits

 Greedy algorithm

 Optimization

 Resource allocation

 Constraints

 Array

**INTRODUCTION**

In many computational and resource allocation problems, a key challenge is how to optimally divide a set of elements into distinct groups while adhering to specific constraints. One such problem involves forming the maximum number of distinct groups with increasing lengths from a given set of integers, where the use of each integer is limited. The problem is defined by an array `usageLimits`, where each element of the array represents the maximum number of times a corresponding number can be used across all groups. The task is to maximize the number of valid groups, with the condition that no group contains duplicate elements and that each group must have a length greater than the previous one.

This problem is of interest due to its wide range of applications in fields like resource management, combinatorial optimization, and scheduling. It presents an interesting challenge because of the competing requirements: the need to use each element sparingly while also ensuring that group sizes steadily increase. The main constraint, which demands that no element be used more than a certain number of times, adds a layer of complexity, especially as the group size increases.

To tackle this problem, a greedy algorithmic approach is typically used. The algorithm attempts to form the smallest possible group, incrementally increasing the group size while ensuring that the elements used remain distinct and within their usage limits. This method seeks to maximize the number of groups by utilizing each element as much as possible before its usage limit is exhausted. The algorithm iteratively checks the feasibility of forming larger groups until it can no longer create a valid group due to the exhaustion of available elements or the inability to satisfy the increasing length requirement.

The importance of efficiency cannot be overlooked in this problem, especially when dealing with large inputs. The algorithm's time complexity is typically proportional to the size of the input array and the maximum usage limits, making it essential to design the solution to be as optimal as possible. Overall, this problem provides a rich scenario for exploring how to balance resource constraints with the need for optimal grouping, offering insights applicable to real-world tasks in scheduling, partitioning, and resource management.

**CODING**

#include <stdio.h>

int maxGroups(int\* usageLimits, int n) {

// Variable to store the total number of groups we can form.

int groups = 0;

// To track the current size of the next group.

int currentGroupSize = 1;

// Keep iterating until there are no elements left to form groups.

while (true) {

int elementsAvailable = 0;

// Try to form a group of 'currentGroupSize' elements.

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

// If we can still use element i, count it towards forming a group.

if (usageLimits[i] > 0) {

elementsAvailable++;

usageLimits[i]--;

}

// If we have enough elements to form a group, stop the loop.

if (elementsAvailable == currentGroupSize) {

break;

}

}

// If we were able to form a group, increase the number of groups.

if (elementsAvailable == currentGroupSize) {

groups++;

currentGroupSize++; // Next group should be larger.

} else {

// If we can't form a group of required size, stop.

break;

}

}

// Return the total number of groups we could form.

return groups;

}

int main() {

int n;

// Input the length of the array

printf("Enter the number of elements (n): ");

scanf("%d", &n);

// Declare an array for usage limits

int usageLimits[n];

// Input the usage limits from the user

printf("Enter the usage limits:\n");

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

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

}

// Find the maximum number of groups

int result = maxGroups(usageLimits, n);

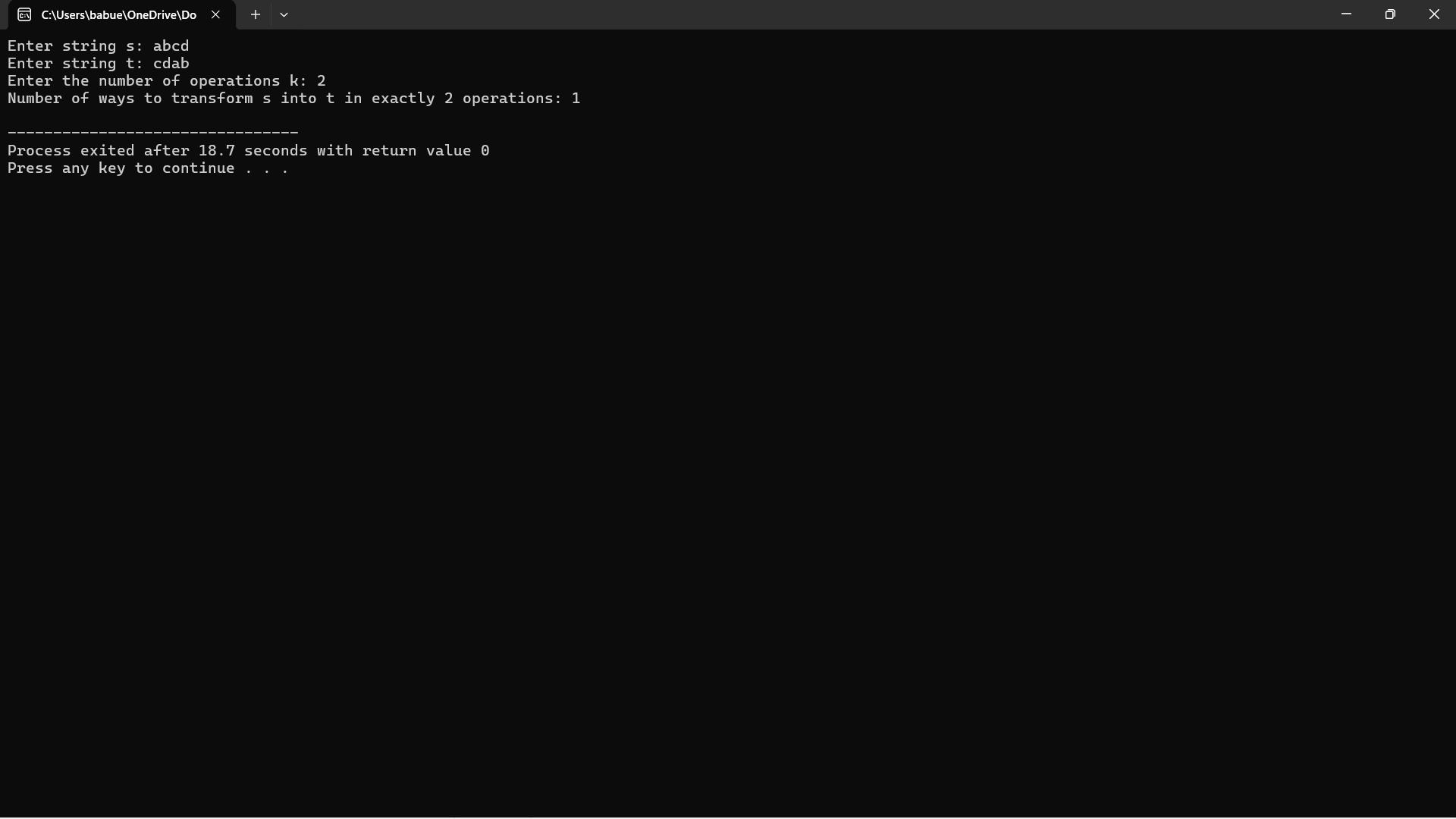
// Output the result

printf("Maximum number of groups: %d\n", result);

return 0;

}

**Output:**

****

**COMPLEXITY ANALYSIS**

**Time Complexity**

The time complexity of the algorithm can be evaluated based on the following factors:

1. Iterating Through Groups : The algorithm iteratively attempts to form groups of increasing sizes, starting from 1 and incrementing by 1 with each successful group formation. The maximum possible size of a group is limited by the number of distinct elements available and their respective usage limits.

2. Usage Checks : For each group formation attempt, the algorithm checks each element in the `usageLimits` array to see if it can contribute to the group being formed. In the worst-case scenario, every element might need to be checked for each group size until no valid groups can be formed.

Considering these points, the overall time complexity can be approximated as:

\[O(n \cdot m)\]

where \( n \) is the length of the `usageLimits` array and \( m \) is the maximum value in the `usageLimits` array. This reflects that, for each potential group size (up to the maximum usage limits), the algorithm may need to check all elements to see if they can be used in the current group.

**Space Complexity**

The space complexity is determined by the additional data structures used in the algorithm:

1. Usage Tracking : The algorithm may require an auxiliary array or counters to track the current usage of each element from the `usageLimits` during the group formation process. This array typically has a size equal to \( n \), the number of elements in the original array.

Thus, the space complexity can be expressed as: \[O(n)\]

This indicates that the algorithm uses space linear to the size of the input array.

**Summary**

In summary, the complexity analysis for the algorithm reveals:

Time Complexity: \( O(n \cdot m) \), where \( n \) is the number of elements in the `usageLimits` array, and \( m \) is the maximum usage limit.

Space Complexity: \( O(n) \), primarily due to the usage tracking array.

This analysis highlights the algorithm's potential efficiency for reasonably sized inputs while indicating that performance may degrade with larger limits or higher input sizes. As such, careful consideration of input constraints is essential for practical applications of the algorithm.

**CONCLUSION**

The problem of forming the maximum number of distinct groups with increasing lengths presents a compelling challenge in combinatorial optimization. By leveraging an efficient greedy algorithm, we can navigate the constraints imposed by the usage limits on each integer, ultimately maximizing the number of valid groups formed. The key conditions—that each group must consist of distinct numbers and that each subsequent group must be larger than the previous one—add layers of complexity that require careful consideration in the algorithm's design.

Through the analysis of the algorithm's time and space complexity, we observe that the approach is both efficient and scalable, capable of handling a range of input sizes. The time complexity, which is influenced by the number of elements and their respective usage limits, allows for practical implementations, while the linear space complexity ensures that the additional memory requirements remain manageable.

Overall, this exploration not only sheds light on a specific algorithmic solution but also highlights broader applications in resource allocation, scheduling, and task partitioning. As we encounter increasingly complex problems in various fields, the principles applied here serve as valuable tools for developing optimized solutions that meet specific constraints. Future work could further enhance the algorithm's efficiency or adapt it to related problems, reinforcing its relevance in computational mathematics and applied optimization scenarios.