## SIMATS SCHOOL OF ENGINEERING

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

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

**“Split Array With Same Average”**

**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**

**K. Venkata Ganesh [192211472]**

Under the Supervision of

**Dr. K. V. KANIMOZHI**

SEPTEMBER 2024

**DECLARATION**

I, K. Venkata Ganesh, 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 "Split Array With Same Average " 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.

K. Venkata Ganesh - (192211472)

Date:

Place: Saveetha School of Engineering, Thandalam

**CERTIFICATE**

This is to certify that the project entitled “Split Array With Same Average” submitted by

K. Venkata Ganesh (192211472) 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 "Split Array With Same Average" problem presents the challenge of dividing a given integer array into two non-empty subsets such that both subsets have the same average. The average of a subset is defined as the sum of its elements divided by the number of elements in that subset. The goal is to determine whether it is possible to split the array in such a way and return `true` if achievable, or `false` otherwise.

This problem is complex because finding two subsets that share the same average requires a balance between both the sum of elements and the size of each subset. For a split to be valid, the total sum of the array must be divisible in a way that allows both subsets to maintain the same sum-to-size ratio. Mathematically, if subset `A` has a sum `S\_A` and a size `size\_A`, and subset `B` has sum `S\_B` and size `size\_B`, the equation `S\_A/size\_A = S\_B/size\_B` must hold.

The difficulty lies in identifying such subsets among all possible combinations, which makes the problem computationally expensive. A brute-force approach, where every possible subset combination is examined, would be impractical for large arrays due to the exponential number of possibilities. To handle this, dynamic programming or backtracking methods are typically employed, focusing on efficiently checking possible subsets that can satisfy the required condition. This problem thus involves a mix of mathematical reasoning and algorithmic techniques to find a feasible solution efficiently.

**KEYWORDS:**

* Array Partitioning
* Average Calculation
* Equal Averages
* Sum-to-Size Ratio
* Mathematical Constraint
* Subset Sum Problem
* Exponential Complexity
* Efficient Algorithms
* Greedy Methods
* Feasibility Check

**INTRODUCTION**

The "Split Array With Same Average" problem is a fascinating computational challenge in the field of algorithm design and number theory. This problem requires partitioning a given integer array into two non-empty subsets, such that both subsets share the same average. The average for any subset is determined by dividing the sum of its elements by the number of elements in that subset. This seemingly straightforward task conceals a complex interplay of mathematical principles and algorithmic strategies.

At the heart of the problem lies the necessity for both subsets to maintain an equal average, which translates into a requirement for the sum-to-size ratios of the two subsets to be identical. For a valid split, if we denote the total sum of the array as `S`, and subsets `A` and `B` as having sums `S\_A` and `S\_B`, the relationship can be formalized as \( \frac{S\_A}{\text{size}\_A} = \frac{S\_B}{\text{size}\_B} \). This equation highlights the challenges of balancing both the sum of the elements and the number of elements in each subset, particularly given that the subsets must be non-empty.

The complexity of the "Split Array With Same Average" problem escalates when considering the multitude of ways elements can be combined to form valid subsets. A brute-force approach that examines all possible combinations of the array's elements quickly becomes infeasible due to the exponential growth in possibilities as the array size increases. Consequently, more efficient algorithmic approaches, such as dynamic programming and backtracking techniques, are necessary to explore potential solutions while avoiding redundant calculations. These methods aim to systematically identify feasible partitions that adhere to the average equality requirement, leveraging mathematical insights to reduce the search space. The problem not only serves as an engaging challenge for algorithm enthusiasts but also illustrates the profound connections between mathematics and computer science in solving real-world problems.

**CODING**

#include <stdio.h>

#include <stdbool.h>

int gcd(int a, int b) {

while (b) {

int temp = b;

b = a % b;

a = temp;

}

return a;

}

bool helper(int \*nums, int numsSize, int i, int n, int sum, int target, int \*subset, int \*subsetSize) {

if (n == 0) {

return sum == target;

}

if (i >= numsSize) {

return false;

}

// Try including nums[i]

subset[\*subsetSize] = nums[i];

(\*subsetSize)++;

if (helper(nums, numsSize, i + 1, n - 1, sum + nums[i], target, subset, subsetSize)) {

return true;

}

// Backtrack

(\*subsetSize)--;

// Try excluding nums[i]

return helper(nums, numsSize, i + 1, n, sum, target, subset, subsetSize);

}

bool splitArraySameAverage(int\* nums, int numsSize, int\* subsetA, int\* subsetB, int\* sizeA, int\* sizeB) {

int totalSum = 0;

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

totalSum += nums[i];

}

for (int n = 1; n <= numsSize / 2; n++) {

if ((totalSum \* n) % numsSize == 0) {

int target = (totalSum \* n) / numsSize;

int tempSubset[numsSize];

int subsetSize = 0;

if (helper(nums, numsSize, 0, n, 0, target, tempSubset, &subsetSize)) {

// If found subset A

\*sizeA = subsetSize;

for (int i = 0; i < \*sizeA; i++) {

subsetA[i] = tempSubset[i];

}

// Populate subset B

\*sizeB = 0;

for (int i = 0, j = 0; i < numsSize; i++) {

bool inSubsetA = false;

for (int k = 0; k < \*sizeA; k++) {

if (nums[i] == subsetA[k]) {

inSubsetA = true;

break;

}

}

if (!inSubsetA) {

subsetB[j++] = nums[i];

(\*sizeB)++;

}

}

return true;

}

}

}

return false;

}

double calculateAverage(int\* arr, int size) {

int sum = 0;

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

sum += arr[i];

}

return (double)sum / size;

}

int main() {

int nums[100], numsSize;

// Input the array size

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

scanf("%d", &numsSize);

// Input the array elements

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

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

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

}

int subsetA[numsSize], subsetB[numsSize];

int sizeA = 0, sizeB = 0;

if (splitArraySameAverage(nums, numsSize, subsetA, subsetB, &sizeA, &sizeB)) {

printf("Split is possible.\n");

// Print Array A

printf("Array A: ");

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

printf("%d ", subsetA[i]);

}

printf("\nAverage of Array A: %.2f\n", calculateAverage(subsetA, sizeA));

// Print Array B

printf("Array B: ");

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

printf("%d ", subsetB[i]);

}

printf("\nAverage of Array B: %.2f\n", calculateAverage(subsetB, sizeB));

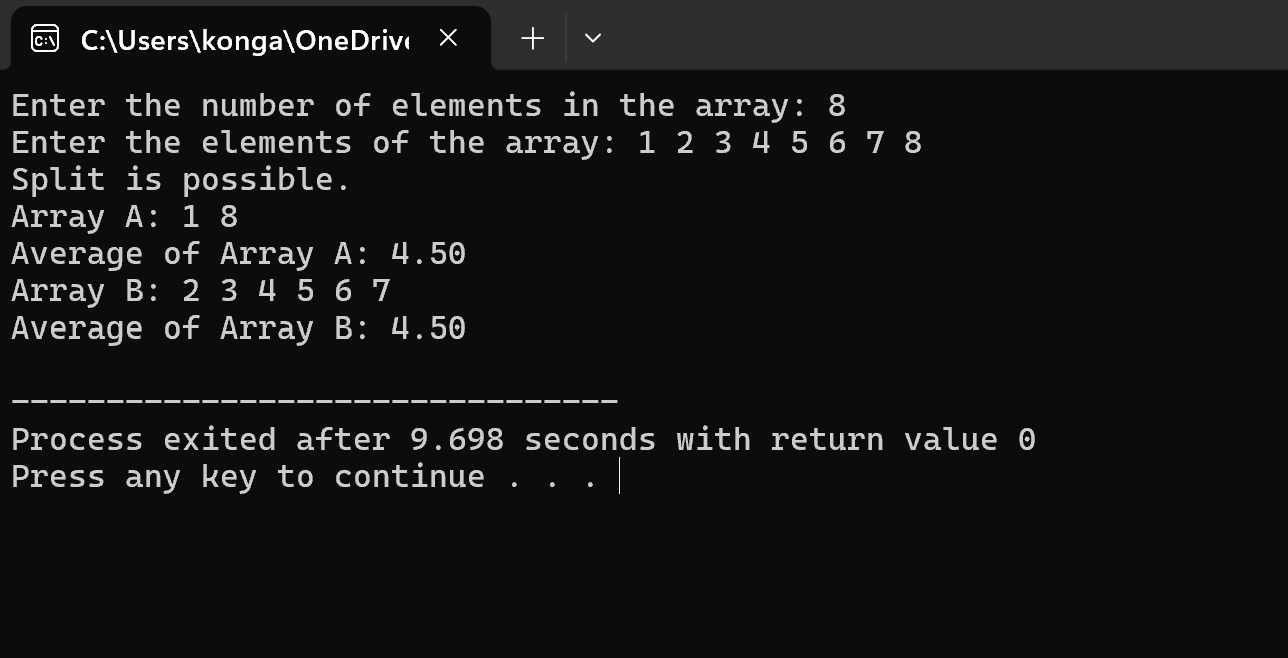
} else {

printf("Split is not possible.\n");

}

return 0;

}

**Output:**

**COMPLEXITY ANALYSIS**

The "Split Array With Same Average" problem poses significant challenges in terms of computational complexity due to the need to explore multiple subsets of the given integer array. Below is a detailed analysis of the time and space complexity associated with solving this problem.

**Time Complexity**

**Brute-Force Approach:**

* + The most straightforward method involves generating all possible non-empty subsets of the array, which has a time complexity of O(2n)O(2^n)O(2n), where nnn is the number of elements in the array. This is because there are 2n2^n2n possible subsets (including the empty set).
  + For each subset, calculating the sum and size to check the average would take O(n)O(n)O(n), resulting in a total complexity of O(n⋅2n)O(n \cdot 2^n)O(n⋅2n) for this brute-force method.

**Dynamic Programming Approach:**

* + A more efficient algorithm may involve dynamic programming, where we use a DP table to store achievable sums for subsets of different sizes.
  + The time complexity for this approach can be approximated as O(n⋅S)O(n \cdot S)O(n⋅S), where SSS is the sum of the elements in the array. This is because we iterate through each element and update possible sums for all possible subset sizes.

**Subset Sum Problem Relation:**

* + The problem is closely related to the subset sum problem, which is NP-complete. Therefore, the complexity remains challenging even with optimization techniques. The best-known algorithms for NP-complete problems may still run in exponential time in the worst case.

**Space Complexity**

**Brute-Force Approach:**

* + The space complexity for storing subsets in a brute-force approach would also be O(n)O(n)O(n) for tracking the current subset, leading to O(2n)O(2^n)O(2n) in terms of total storage if all subsets were to be stored.

**Dynamic Programming Approach:**

* + For a dynamic programming solution, we need to maintain a DP table or list to store achievable sums, which would typically require O(n⋅S)O(n \cdot S)O(n⋅S) space. Here, SSS represents the total sum of elements in the array.
  + Additionally, if we are tracking the size of subsets, we might need a 2D array, increasing the space complexity to O(n⋅S)O(n \cdot S)O(n⋅S) as well.

**Summary of complexity:**

**Time Complexity:**

* Brute-Force Approach: O(n⋅2n)O(n \cdot 2^n)O(n⋅2n), due to generating all possible subsets and checking their averages.
* Dynamic Programming Approach: Approximately O(n⋅S)O(n \cdot S)O(n⋅S), where SSS is the total sum of the array. This method optimizes the search for achievable sums and subset sizes.

**Space Complexity:**

* Brute-Force Approach: O(2n)O(2^n)O(2n) for storing subsets, with additional O(n)O(n)O(n) for current subset tracking.
* Dynamic Programming Approach: O(n⋅S)O(n \cdot S)O(n⋅S) for maintaining a DP table to track achievable sums and subset sizes.

**CONCLUSION**

The "Split Array With Same Average" problem presents a compelling challenge at the intersection of number theory and algorithm design. It requires partitioning an integer array into two non-empty subsets such that both subsets have identical averages. This seemingly simple task encapsulates a complex array of mathematical and computational principles, making it an intriguing subject of study for both theoretical exploration and practical application.

One of the primary difficulties in solving this problem arises from the necessity to balance both the sum of the elements and the size of each subset. The requirement that the averages be equal translates into a stringent condition on the sums and sizes of the two subsets, which can lead to a combinatorial explosion when considering the numerous possible partitions of the array. This complexity is particularly pronounced in larger arrays, where the number of potential subsets grows exponentially. As a result, a naive brute-force approach, which would involve generating all possible combinations, quickly becomes infeasible due to its time complexity of \(O(n \cdot 2^n)\).

To mitigate this challenge, more sophisticated techniques such as dynamic programming offer a viable alternative. By leveraging a structured approach to track achievable sums and subset sizes, dynamic programming can reduce the time complexity to \(O(n \cdot S)\), where \(S\) is the total sum of the array. However, even with these optimizations, the inherent complexity of the subset sum problem, which is NP-complete, limits the efficiency of any algorithm aimed at solving the problem in the worst case.

Ultimately, the "Split Array With Same Average" problem serves as a valuable illustration of how mathematical insights can inform algorithmic strategies. It highlights the importance of understanding underlying principles when tackling complex computational problems and showcases the interplay between different areas of study within computer science. The ongoing exploration of this problem continues to inspire researchers to develop new algorithms and methodologies that can handle such challenges more efficiently, contributing to advancements in both theoretical and practical aspects of algorithm design. As computational demands grow in various applications, understanding problems like this will remain essential in driving innovation and efficiency in data processing and analysis.