**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

**Capstone Project**

**SPLIT ARRAY WITH SAME AVERAGE**

**Submitted by**:

**S. Praveen Kumar (192210546)**

**Under the guidance of:**

DR. Gnana Soundari

**CSA0652-Design and analysis of Algorithms for Amortized Analysis**



**SIMATS ENGINEERING**

**THANDALAM**

**SEPTEMBER 2024**

**Abstract:**

The "Split Array with Same Average" problem involves dividing a given array into two non-empty subarrays such that the average of the elements in both subarrays is the same. This problem can be approached using a combination of backtracking and dynamic programming. By leveraging backtracking, we explore all possible ways to partition the array, while dynamic programming is used to optimize repeated calculations, minimizing time complexity. This paper presents an efficient solution using these techniques, along with a discussion of the challenges involved and the performance of the approach in comparison to brute-force methods.

**Problem Statement:**

You have given an array **nums**having n elements, You have to move each element of **nums**to one of the array **A**or **B**such that **average(A)** equal to **average(B)**if it is possible to achieve then return**TRUE**else **FALSE**. constraints are nums array can not have more than 30 elements and all the values are positive integer and less then 10001 (10⁴ +1).

**Introduction:**

The problem of splitting an array into two subarrays with the same average presents an interesting computational challenge in both theoretical and practical applications. While simple on the surface, finding an optimal partition without brute-force enumeration is non-trivial due to the potential for large arrays and numerous combinations. A brute-force approach would involve testing all possible partitions, which quickly becomes infeasible as the array size increases.

To address this, a combination of backtracking and dynamic programming provides a more efficient solution. Backtracking allows us to explore potential partitions recursively, pruning paths that are unlikely to lead to valid solutions, while dynamic programming helps avoid redundant calculations by storing intermediate results. This approach balances exploration with optimization, offering a more scalable solution.

In this paper, we will describe the problem formally, introduce the proposed algorithm using backtracking and dynamic programming, and analyze its time complexity and efficiency compared to alternative methods.

**Proposed design work:**

The proposed design utilizes backtracking with dynamic programming to explore all possible splits of the array, ensuring equal averages for both resulting arrays. Memoization optimizes the process by storing intermediate results, facilitating efficient determination of the feasibility of such splits.

**I)Identifying the key components:**

Backtracking Algorithm: Utilized to explore all possible splits of the array into two non-empty arrays while maintaining the condition of equal averages for both.

Dynamic Programming (Memoization): Employed to optimize the backtracking process by storing and reusing intermediate results, preventing redundant calculations and improving efficiency.

Base Case Handling: Ensures proper handling of edge scenarios, such as when the length of the array is less than 2, providing an early exit condition.

Total Sum Calculation: Computes the total sum of elements in the array, which is essential for evaluating the condition of equal averages for both resulting arrays.

Recursive Function: Defined to perform the backtracking process, taking into account the current index, the number of elements selected so far, and the sum of elements selected so far in each recursive call.

Condition Checking: Verifies whether the current selection of elements satisfies the condition of having the same average for both arrays, guiding the exploration process.

Return Statement: Determines the feasibility of achieving equal averages by returning True if a valid split is found, and False otherwise.

**2)Functionality:**

The functionality of the proposed design can be summarized as follows:

Splitting Array: The function attempts to split the given array into two non-empty arrays while ensuring that both resulting arrays have the same average.

Exploration of Splits: It systematically explores all possible combinations of splitting the array using backtracking, considering different elements for each array and checking if the condition of equal averages is met.

Optimization with Memoization: The dynamic programming approach optimizes the exploration process by memorizing intermediate results, avoiding redundant calculations, and improving overall efficiency.

Base Case Handling: It handles base cases, such as when the length of the array is insufficient for splitting, ensuring proper termination of the algorithm.

Return Value: The function returns True if a valid split with equal averages is found, indicating that such a split is feasible. Otherwise, it returns False, indicating that no such split exists for the given array.

**3)Program**

#include <stdio.h>

#include <stdbool.h>

bool canSumTo(int nums[], int n, int target\_sum, int k) {

bool dp[target\_sum + 1];

for (int i = 0; i <= target\_sum; i++) {

dp[i] = false;

}

dp[0] = true;

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

for (int i = target\_sum; i >= nums[j]; i--) {

dp[i] |= dp[i - nums[j]];

}

}

return dp[target\_sum];

}

bool splitArraySameAverage(int nums[], int n) {

int total\_sum = 0;

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

total\_sum += nums[i];

}

for (int size\_A = 1; size\_A <= n / 2; size\_A++) {

if (total\_sum \* size\_A % n == 0) {

int target\_sum\_A = total\_sum \* size\_A / n;

if (canSumTo(nums, n, target\_sum\_A, size\_A)) {

return true;

}

}

}

return false;

}

int main() {

int nums[100], n;

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

scanf("%d", &n);

printf("Enter the elements of the array separated by space: ");

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

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

}

if (splitArraySameAverage(nums, n)) {

printf("true\n");

} else {

printf("false\n");

}

return 0;

}

**UI design:**

**1.Layoutdesign:**  
**Flexible layout:** To solve the problem of splitting the given integer array **nums** into two non-empty arrays A and B such that their averages are equal, one approach is to iterate through all possible sizes of array A and check if there exists a subset of elements in **nums** whose sum equals a target sum derived from the total sum of **nums** and the size of A. This can be done using dynamic programming to efficiently find if such a subset exists. If a valid split is found, the function returns True; otherwise, it returns False.

**User Friendly:**

Even Split Validation: The code checks if it's possible to fairly split the list of numbers into two groups with the same average, ensuring fairness in distribution.

Efficient Subset Sum Search: Utilizing a smart algorithm, the code efficiently searches for subsets whose sum equals a target value, ensuring quick and accurate determination of a balanced split.

**Feasable elements used:**

**Elements positioning:** The code strategically positions elements from the input list into two groups, exploring various combinations to achieve equal averages. It intelligently examines subsets of elements and their positions within the list, ensuring a balanced distribution to satisfy the condition of having two groups with the same average.

**Assessibility:** The code ensures accessibility by employing straightforward logic and clear variable naming, facilitating easy understanding and interaction for users of all levels.

**Elements Function:**

Sum Calculation: The code performs summation operations on elements within the input list, facilitating the computation of total sums essential for determining average values.

Subset Generation: Utilizing elements from the list, the code generates subsets representing potential combinations of elements to be distributed among the two groups, enabling comprehensive exploration of possible splits.

Feasibility Assessment: By applying operations on subsets and calculating their sums, the code assesses the feasibility of splitting the elements into two groups with equivalent averages, ensuring fairness and balance in the distribution process.

**Login template:**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Login - Split Array With Same Average</title>

<style>

body {

font-family: Arial, sans-serif;

margin: 0;

padding: 0;

height: 100vh;

background-image: url('background\_image.jpg');

background-size: cover;

display: flex;

justify-content: center;

align-items: center;

}

.login-container {

background-color: rgba(255, 255, 255, 0.9);

padding: 20px;

border-radius: 8px;

box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);

width: 320px;

}

.login-container h2 {

margin-bottom: 20px;

text-align: center;

color: #007bff;

}

.login-container form {

display: flex;

flex-direction: column;

}

.login-container label {

margin-bottom: 10px;

color: #495057;

}

.login-container input[type="text"],

.login-container input[type="password"] {

padding: 10px;

margin-bottom: 15px;

border: 1px solid #ced4da;

border-radius: 4px;

font-size: 16px;

}

.login-container input[type="submit"] {

padding: 10px;

border: none;

border-radius: 4px;

font-size: 16px;

background-color: #007bff;

color: #fff;

cursor: pointer;

transition: background-color 0.3s;

}

.login-container input[type="submit"]:hover {

background-color: #0056b3;

}

</style>

</head>

<body>

<div class="login-container">

<h2>Login - Split Array With Same Average</h2>

<form action="#" method="post" id="login-form">

<label for="username">Username:</label>

<input type="text" id="username" name="username" required>

<label for="password">Password:</label>

<input type="password" id="password" name="password" required>

<input type="submit" value="Login">

</form>

</div>

</body>

</html>

**Conclusion:**

In the problem of Split Array With Same Average, the objective is to distribute the given integer array **nums** into two non-empty arrays A and B, ensuring that both have the same average. This entails exploring various combinations of elements to achieve a fair distribution. By checking if such a split is feasible through efficient calculations, the algorithm determines whether it's possible to attain equal averages for both arrays A and B, returning true if achievable, and false otherwise.