**CAPSTONE PROJECT REPORT**

**Reg NO:192210718**

**Name: V.sanjay**

**Course Code:CSA0656**

**Course Name:Design and Analysis of Algorithms for Asymptotic Notations**

**SLOT: A**

## ABSTRACT

The problem of splitting an integer array into two non-empty subarrays with the same average can be approached by leveraging mathematical properties and combinatorial techniques. The goal is to determine if it's feasible to divide the array nums into two subsets A and B such that the average of elements in A is equal to the average of elements in B.

Given the array nums, the key observation is that for such a split to exist, the sum of elements and the size of the subsets must be proportional. Mathematically, this translates to finding subsets where the sum of the subset elements divided by the subset size equals the overall average of the array.

The solution involves:

1. Calculating the overall average of the array.
2. Using dynamic programming or backtracking techniques to explore possible subsets and check if they meet the required conditions.
3. Ensuring the subsets are non-empty and properly partitioned from the original array.

The algorithm must consider all possible subset combinations and validate their averages, making it a complex problem often requiring optimized approaches to handle large arrays efficiently. The example provided, [1,2,3,4,5,6,7,8], demonstrates a successful split into [1,4,5,8] and [2,3,6,7], both averaging to 4.5.

# INTRODUCTION

The problem of determining whether an integer array can be split into two non-empty subarrays with the same average is a fascinating challenge that combines elements of number theory and combinatorics. This problem asks if it is possible to partition a given array `nums` into two subsets `A` and `B` such that the average of the elements in `A` equals the average of the elements in `B`.

The average of a subset is defined as the sum of its elements divided by the number of elements in the subset. Hence, for the two subsets to have the same average, their sums must be proportional to their sizes. This translates into finding subsets whose sum and size ratios match the overall array's sum and size ratio.

For instance, given the array `[1,2,3,4,5,6,7,8]`, one can split it into subsets `[1,4,5,8]` and `[2,3,6,7]`, both having an average of `4.5`. This problem is not only theoretically interesting but also computationally challenging due to the combinatorial nature of finding suitable subsets from the array. The solution involves a mix of mathematical insights and algorithmic strategies such as dynamic programming or backtracking to efficiently explore possible partitions and validate their averages. This makes it a compelling topic for those interested in algorithm design and discrete mathematics.

# PROBLEM STATEMENT

* Split Array With Same Average You are given an integer array nums. You should move each element of nums into one of the two arrays A and B such that A and B are non-empty, and average(A) == average(B). Return true if it is possible to achieve that and false otherwise. Note that for an array arr, average(arr) is the sum of all the elements of arr over the length of arr.
* **Example 1**: Input: nums = [1,2,3,4,5,6,7,8] Output: true
* **Explanation**: We can split the array into [1,4,5,8] and [2,3,6,7], and both of them have an average of 4.5.

# APPROACH

To solve the problem of splitting an array into two non-empty subarrays with the same average, follow these steps:

**Mathematical Insight:**

* 1. Calculate the overall sum SSS and length nnn of the array numsnumsnums.
  2. The average of the entire array is average(nums)=Sn\text{average}(nums) = \frac{S}{n}average(nums)=nS​.
  3. For two subsets AAA and BBB with sizes kkk and n−kn-kn−k to have the same average, the sum of subset AAA must be k×Sn\frac{k \times S}{n}nk×S​ and the sum of subset BBB must be (n−k)×Sn\frac{(n-k) \times S}{n}n(n−k)×S​.

**Feasibility Check:**

* 1. For k×Sn\frac{k \times S}{n}nk×S​ to be an integer, k×Sk \times Sk×S must be divisible by nnn. This means there must exist a subset size kkk such that k×S%n=0k \times S \% n = 0k×S%n=0.

**Dynamic Programming/Backtracking Approach:**

* 1. Use dynamic programming (DP) or backtracking to explore all possible subsets of different sizes and check if their sums meet the required conditions.
  2. Use a DP array dp[i][j] where i represents the number of elements considered, and j represents the possible sum with those i elements.

**Implementation Steps:**

* 1. Iterate over possible sizes kkk from 1 to n−1n-1n−1.
  2. For each kkk, calculate the required sum target\_sum=k×Sntarget\\_sum = \frac{k \times S}{n}target\_sum=nk×S​.
  3. Use a DP approach to check if there exists a subset of size kkk with sum target\_sumtarget\\_sumtarget\_sum.

# CODE

#include <stdio.h>

#include <stdbool.h>

bool canSplit(int\* nums, int numsSize, int sum, int count) {

if (count == 0) return sum == 0;

if (sum < 0 || count < 0) return false;

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

if (canSplit(nums + i + 1, numsSize - i - 1, sum - nums[i], count - 1)) {

return true;

}

}

return false;

}

bool splitArraySameAverage(int\* nums, int numsSize) {

int totalSum = 0;

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

totalSum += nums[i];

}

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

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

int sumA = totalSum \* lenA / numsSize;

if (canSplit(nums, numsSize, sumA, lenA)) {

return true;

}

}

}

return false;

}

int main() {

int nums[] = {1, 2, 3, 4, 5, 6, 7, 8};

int numsSize = sizeof(nums) / sizeof(nums[0]);

if (splitArraySameAverage(nums, numsSize)) {

printf("true\n");

} else {

printf("false\n");

}

return 0;

}

# RESULT

# 

**COMPLEXITY ANALYSIS**

### Naive Approach Complexity

**Time Complexity:**

1. **Triplet Selection:** The naive approach involves three nested loops to consider all possible triplets (x, y, z) where x < y < z. Each loop iterates up to n times, leading to a time complexity of O(n^3).
2. **Condition Check:** For each triplet, checking whether the elements in the two subarrays satisfy the condition (i.e., whether their averages are equal) is performed in constant time, O(1).

**Overall Time Complexity:** O(n^3)

**Space Complexity:**

1. **Input Array:** The space required to store the input array nums is O(n).
2. **Auxiliary Variables:** Additional space for variables used in computations and loop indices is constant, O(1).

**Overall Space Complexity:** O(n)

### Optimized Approach Using Fenwick Trees

**Time Complexity:**

1. **Index Mapping:** Creating position arrays for the elements of nums1 and nums2 takes linear time, O(n).
2. **Fenwick Tree Operations:**
   1. Each update and query operation on the Fenwick Tree takes logarithmic time, O(log n).
   2. Calculating the right count for each element requires O(n log n) time.
   3. Calculating the left count and counting valid triplets also requires O(n log n) time.

**Overall Time Complexity:** O(n log n)

**Space Complexity:**

1. **Fenwick Trees:** Two Fenwick Trees are used, each requiring O(n) space.
2. **Position Arrays:** Two position arrays, each of length n, require O(n) space.
3. **Auxiliary Arrays:** Arrays like right\_counts require O(n) space.

**Overall Space Complexity:** O(n)

# CONCLUSION

The analysis of the problem "Count Good Triplets in an Array" using both the naive and optimized approaches provides a comprehensive understanding of their performance and efficiency. Here are the key conclusions:

### Naive Approach:

* + **Time Complexity:** The naive approach has a time complexity of O(n^3) in the best, worst, and average cases. This involves checking all possible triplets in the array, which becomes impractical for large values of nnn.
  + **Space Complexity:** The space complexity of the naive approach is O(1), as it

does not require any additional data structures beyond the input arrays.

* + **Suitability:** Due to its high time complexity, the naive approach is only suitable for very small arrays where nnn is relatively small.

### Optimized Approach Using Fenwick Trees:

* + **Time Complexity:** The optimized approach significantly improves the time complexity to O(nlogn) in the best, worst, and average cases. This improvement is achieved by using Fenwick Trees (Binary Indexed Trees) to efficiently count and manage the elements in the arrays.
  + **Space Complexity:** The space complexity of the optimized approach is O(n) due

to the additional data structures (Fenwick Trees) required.

* + **Suitability:** The optimized approach is highly suitable for larger arrays, providing a much more practical solution for real-world applications where nnn can be large.

### Overall Comparison:

* + The naive approach, while straightforward and simple to implement, is not efficient for larger datasets due to its cubic time complexity.
  + The optimized approach, leveraging advanced data structures like Fenwick Trees, offers a substantial improvement in performance, making it a viable solution for larger input sizes.

### Real-World Implications:

* + In real-world scenarios, where performance and efficiency are critical, the optimized approach is the preferred method. It provides a balance between time and space complexity, ensuring that the solution is both fast and scalable.

### Future Considerations:

* + While the optimized approach using Fenwick Trees is effective, further optimizations or alternative data structures (such as Segment Trees) could be explored to potentially enhance performance even further.
  + Additionally, parallel processing or distributed computing techniques could be

considered for handling extremely large datasets.