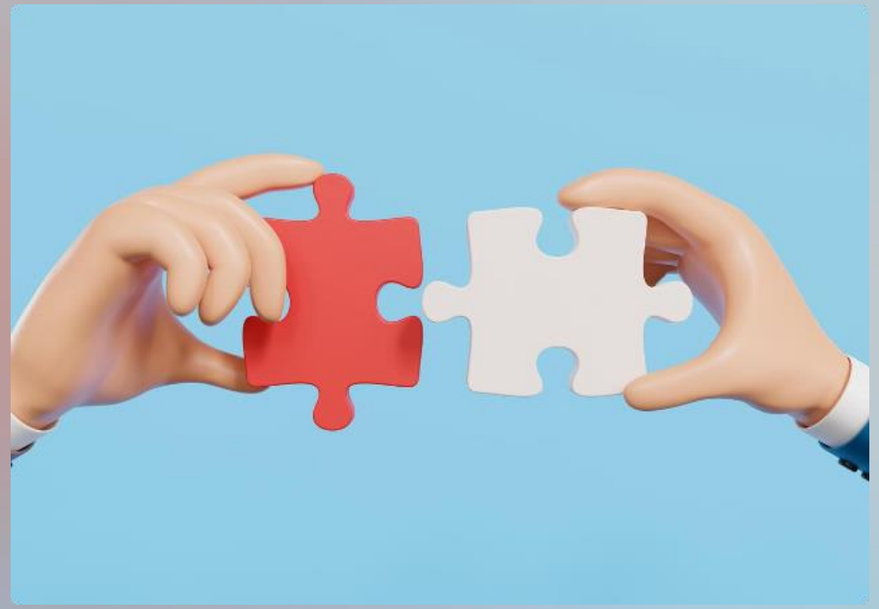
Split Array With Same Average

Imagine you have an array of numbers. The goal is to split this array into two subarrays with the same average value. Sounds simple, right? Let's explore the challenges and solutions.



Defining the Problem Statement

We need to determine if we can split the array into two non-empty subarrays where the average of elements in the first subarray equals the average of elements in the second subarray.

1

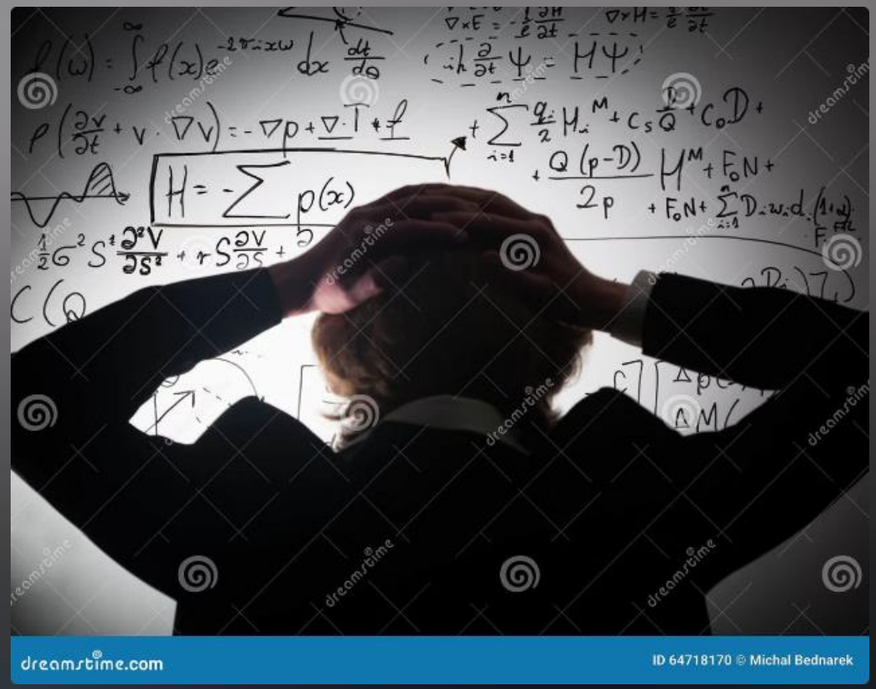
Input

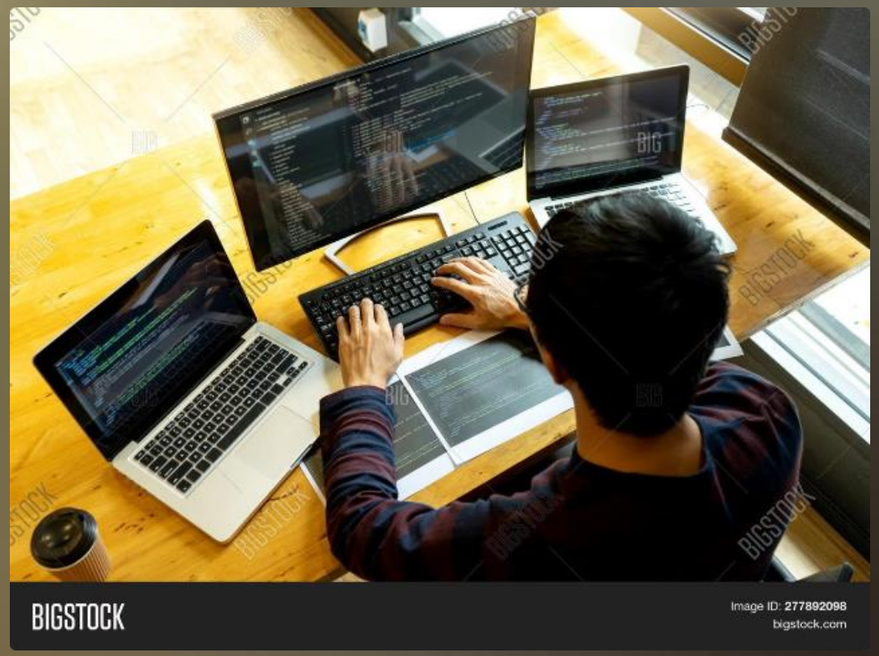
An array of integers.

2

Output

True if the array can be split into two subarrays with the same average, otherwise False.





Brute Force Approach

The brute force method involves checking all possible combinations of subarrays.

Generate Subarrays

Generate all possible combinations of subarrays.

Compare Averages

Compare the averages of the two subarrays.

1

2

3

Calculate Averages

Calculate the average of each subarray.

Optimizing the Brute Force Approach

While brute force works, it's very inefficient due to the exponential number of combinations to consider.

1

Reduce Search Space

We can reduce the search space by considering only subarrays with a sum that is a multiple of the total sum of the array.

2

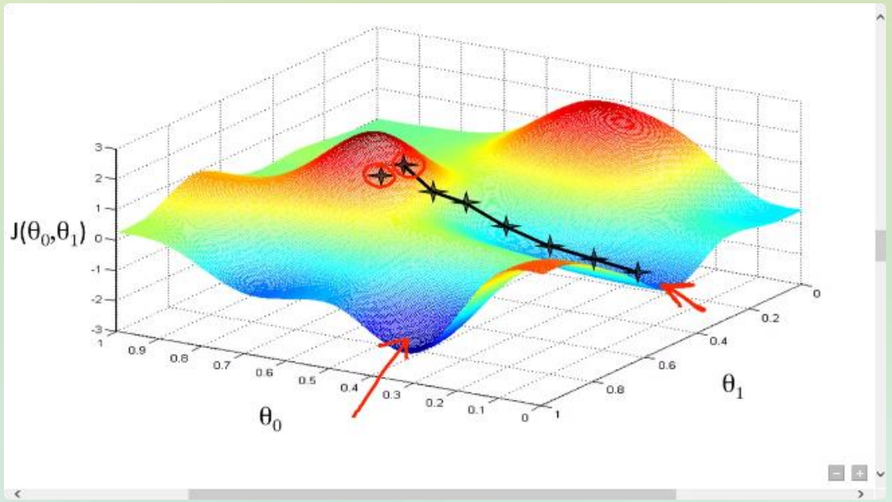
Calculate Subarray Sums

Calculate the sum of all possible subarrays.

3

Check for Equality

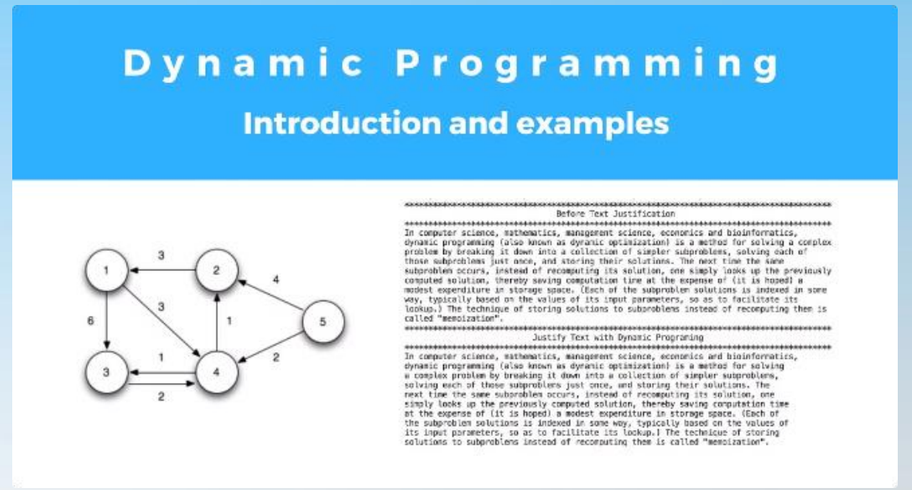
Check if any two subarray sums are equal and their total sum equals the total sum of the array.



Dynamic Programming Solution

Dynamic programming can be used to efficiently solve this problem by breaking it down into smaller overlapping subproblems.

Step	Action
1	Calculate the total sum of the array.
2	Create a DP table to store subarray sums for each possible sum value.
3	Iterate through the array and calculate the sum of all possible subarrays.
4	Check if there exist two subarray sums that are equal and their total sum equals the total sum of the array.



Coding and Screenshot

```c
#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;
}
```

Output:

```
true

--------------------------------
Process exited after 0.1046 seconds with return value 0
Press any key to continue . . .
```

Time Complexity

The time complexity of the dynamic programming solution is O(n * sum), where 'n' is the length of the array and 'sum' is the total sum of the array elements.

Dynamic Programming

The DP solution iterates through the array and calculates subarray sums for each possible sum value.

Brute Force

The brute force approach involves generating all possible combinations of subarrays, resulting in an exponential time complexity.

Complexity Analysis

The dynamic programming solution offers a significant improvement in time complexity compared to the brute force approach.

Dynamic Programming

O(n * sum)

Brute Force

O(2^n)

Practical Applications

This problem has practical applications in various domains, including resource allocation, data partitioning, and optimizing network configurations.

Network Optimization

Splitting network traffic into balanced groups for efficient routing.

Data Partitioning

Distributing data across multiple servers for load balancing.

Resource Allocation

Assigning tasks to team members based on their skills and workload.

Conclusion

The split array with the same average problem is a classic example of how dynamic programming can significantly optimize solutions.

1 **Key Takeaway**

Dynamic programming can be used to efficiently solve problems with overlapping subproblems.

2 **Complexity Analysis**

The time complexity of the dynamic programming solution is significantly lower than brute force.