# 1712. Ways to Split Array Into Three Subarrays

Medium Array Two Pointers Binary Search Prefix Sum Leetcode Link

## Problem Description

In this problem, we need to find out how many ways we can split an array of non-negative integers into three non-empty contiguous subarrays wherein each subarray is named as `left`, `mid`, and `right` from left to right. The splitting must satisfy the condition that the sum of the elements in `left` is less than or equal to the sum of the elements in `mid`, and in turn, the sum in `mid` should be less than or equal to the sum of the elements in `right`. If we find a way to split the array that meets these conditions, we call it a "good" split.

Our goal is to count the total number of these good splits and return that count modulo `10^9 + 7` as the number can be very large.

## Intuition

The intuition behind the solution is based on the concept of prefix sums and binary search. The array `nums` is processed to create another array, say `s`, which is a cumulative sum array (using `accumulate(nums)`). The cumulative sum array helps us quickly calculate the sum of any contiguous subarray.

Given the prefix sum array `s`, for each index `i` that could potentially be the end of the `left` subarray, we want to find suitable indices `j` and `k` where `j` marks the end of the `mid` subarray, and `k` represents the first index such that the sum of elements in the `right` subarray is not less than the sum of `mid`.

We use binary search (with `bisect_left` and `bisect_right`) to find the positions `j` and `k` based on the sum of `left`.

1. We start by fixing the end of the `left` subarray at index `i`.
2. We search for the smallest index `j` after `i` where the sum of `mid` is at least twice the sum of `left`. This guarantees that sum(`left`) ≤ sum(`mid`).
3. We then search for the largest index `k` where the sum of `mid` is still less than or equal to the sum of `right`. This is ensured by the sum of the entire array minus the sum up to `i`, divided by 2.
4. The number of positions for `k` minus the number of positions for `j` gives us the count of good splits for a fixed `i`.
5. We loop over all possible ends for the `left` subarray and sum up the counts to get the answer.
6. Finally, we return the answer modulo `10^9 + 7` to respect the constraints of the problem.

This approach is efficient as it trims down the search space using binary search rather than checking every possible split by brute-force which would be too slow.

## Solution Approach

The solution approach is a combination of algorithmic strategies involving prefix sums, binary search, and modular arithmetic. Let's go through the steps of the implementation:

1. **Prefix Sum Calculation**: The first step involves calculating the prefix sums of the `nums` array and storing it in the array `s`. This is done using `accumulate(nums)`. Prefix sums allow us to calculate the sum of any continuous subarray in constant time.

2. **Modular Arithmetic Constant**: A constant `mod = 10**9 + 7` is defined at the beginning of the solution. This is used to perform modular arithmetic to prevent integer overflow and to return the final result modulo `10^9 + 7` as required by the problem statement.

3. **Initialization**: We initialize `ans` to store the count of good ways to split the array, and `n` to store the length of the `nums` array.

4. **Iterating Over Potential Left Subarray Ends**: Using a `for` loop, we iterate over each index `i` which could be the end of the `left` subarray. We are careful to stop at `n-2` because we need at least two more elements to form the `mid` and `right` subarrays as they must be non-empty.

5. **Finding the Lower and Upper Bounds for Mid Subarray Ends**:

   - To find the lower bound `j` for `mid`'s end, we use `bisect_left`. We search the prefix sum array `s` for the smallest index `j` such that `2 * s[i] <= s[j]`. This ensures that the sum of `mid` is at least as much as the sum of `left`.
   - Similarly, to find the upper bound `k` for `mid`'s end, we use `bisect_right`. Here we look for the last index `k` such that `s[k] <= (s[-1] + s[i]) / 2`. This ensures that sum(`mid`) ≤ sum(`right`).

6. **Counting Good Splits**: For each position of `i`, the good splits number is given by `k - j`, since for each index from `j` to `k-1`, we can form a valid `mid` subarray maintaining the condition of good split.

7. **Accumulating and Modulo Operation**: The number of good splits calculated for each `i` is added to `ans`. After the loop is finished, `ans` may be a very large number, so we return `ans % mod` to keep the result within the specified numeric range.

This solution is efficient due to binary search, which performs `O(log n)` comparisons rather than linear scans, reducing the overall complexity to `O(n log n)` for the problem. Without binary search, a brute-force approach would have a complexity of `O(n^2)`, which would not be practical for large arrays.

### Example Walkthrough

Let's walk through a small example to illustrate the solution approach described above. Suppose we have the following array `nums` of non-negative integers:

```
nums = [1, 2, 2, 2, 5, 0]
```

Following the steps of the solution:

1. **Prefix Sum Calculation**: Create a prefix sum array `s` using `accumulate`:

```
s = [1, 3, 5, 7, 12, 12] // The cumulative sum of `nums`
```

2. **Modular Arithmetic Constant**: Define `mod = 10**9 + 7` for later use.

3. **Initialization**: Initialize `ans = 0` and `n = 6` (length of `nums`).

4. **Iterating Over Potential Left Subarray Ends**: We consider the potential ends of the `left` subarray. Index `i` can go from 0 to `n-3` (inclusive) to leave room for `mid` and `right`:

```
For i = 0, s[i] = 1
For i = 1, s[i] = 3
For i = 2, s[i] = 5
For i = 3, s[i] = 7
```

We stop at `i = 3` because `i = n-2` would not leave enough elements for `mid` and `right`.

5. **Finding the Lower and Upper Bounds for Mid Subarray Ends**:

   - For each `i`, find `j` and `k` using binary search:

```
For i = 0 (s[i] = 1):
- Search `j` such that `2 * s[i] <= s[j]`: `j = 2`
- Search `k` such that `s[k] <= (s[-1] + s[i]) / 2`: `k = 4`
- Good splits for i = 0: `k - j` = `4 - 2` = 2

For i = 1 (s[i] = 3):
- `j = 3`
- `k = 5`
- Good splits for i = 1: `k - j` = `5 - 3` = 2

 For i = 2 (s[i] = 5):
 - `j = 4`
 - `k = 5`
 - Good splits for i = 2: `k - j` = `5 - 4` = 1

 For i = 3 (s[i] = 7):
 - `j = 4`
 - `k = 5`
 - Good splits for i = 3: `k - j` = `5 - 4` = 1
```

6. **Counting Good Splits**: Sum up all the good splits:

```
ans = 2 + 2 + 1 + 1 = 6
```

7. **Accumulating and Modulo Operation**: Since our `ans` is small, `ans % mod` is trivially `6`. If `ans` were large, the modulo operation would ensure we get a result within the numeric limits of `10^9 + 7`.

Therefore, there are `6` good splits of the array `nums` following the rules set out by the problem.

## Python Solution

```python
from itertools import accumulate
from bisect import bisect_left, bisect_right
from typing import List

class Solution:
    def ways_to_split(self, nums: List[int]) -> int:
        # Constant for modulo operation to prevent overflow
        MOD = 10**9 + 7

        # Calculate the prefix sum of numbers to efficiently compute sums of subarrays
        prefix_sums = list(accumulate(nums))

        answer = 0  # Initialize answer which will hold the number of valid ways to split
        num_elements = len(nums)  # Total number of elements in nums

        # Iterate through the array (except the last two elements, as those are needed for the second and third part)
        for i in range(num_elements - 2):
            # Find the left boundary 'j' for the second part where the sum of the second part is not less than the sum of the first p
            j = bisect_left(prefix_sums, 2 * prefix_sums[i], i + 1, num_elements - 1)

            # Find the right boundary 'k' for the second part where the sum of the third part is not less than the sum of the second
            k = bisect_right(prefix_sums, (prefix_sums[-1] + prefix_sums[i]) // 2, j, num_elements - 1)

            # Increment the answer by the number of valid ways to split given the current first part
            answer += k - j

        # Return the total number of ways to split modulo MOD to handle large numbers
        return answer % MOD

```

## Java Solution

```java
class Solution {
    private static final int MODULO = (int) 1e9 + 7;

    public int waysToSplit(int[] nums) {
        int length = nums.length;
        // Create a prefix sum array
        int[] prefixSum = new int[length];
        prefixSum[0] = nums[0];
        for (int i = 1; i < length; ++i) {
            prefixSum[i] = prefixSum[i - 1] + nums[i];
        }

        int countWays = 0; // This will store the number of ways to split the array
        // Loop through each possible index to split the array after
        for (int i = 0; i < length - 2; ++i) {
            // Find the earliest index to make the second part's sum at least equal to the first part
            int earliest = binarySearchLeft(prefixSum, 2 * prefixSum[i], i + 1, length - 1);
            // Find the latest index where the third part's sum would be at least as much as the second
            int latest = binarySearchRight(prefixSum, ((prefixSum[length - 1] + prefixSum[i]) / 2) + 1, earliest, length - 1);
            // Add the number of ways to split between these two indices
            countWays = (countWays + latest - earliest) % MODULO;
        }
        return countWays;
    }

    private int binarySearchLeft(int[] prefixSum, int target, int leftIndex, int rightIndex) {
        // Standard binary search to find the left-most index where prefixSum[index] >= target
        while (leftIndex < rightIndex) {
            int mid = (leftIndex + rightIndex) / 2;
            if (prefixSum[mid] >= target) {
                rightIndex = mid;
            } else {
                leftIndex = mid + 1;
            }
        }
        return leftIndex;
    }

    private int binarySearchRight(int[] prefixSum, int target, int leftIndex, int rightIndex) {
        // Binary search to find the right-most position to meet the constraints
        while (leftIndex < rightIndex) {
            int mid = (leftIndex + rightIndex) / 2;
            if (prefixSum[mid] < target) {
                leftIndex = mid + 1;
            } else {
                rightIndex = mid;
            }
        }
        // We might overshoot by one, so we adjust if necessary
        if (leftIndex == rightIndex && prefixSum[leftIndex] >= target) {
            return leftIndex;
        }
        return leftIndex - 1;
    }
}

```

## C++ Solution

```cpp
class Solution {
public:
    const int MOD = 1e9 + 7; // Defining the modulo constant for operations

    int waysToSplit(vector<int>& nums) {
        int numCount = nums.size(); // n is the total number of elements in nums
        // Create a prefix sum array where each element s[i] is the sum of nums[0] to nums[i]
        vector<int> prefixSum(numCount, nums[0]);
        for (int i = 1; i < numCount; ++i) {
            prefixSum[i] = prefixSum[i - 1] + nums[i];
        }
        int answer = 0; // To store the final count of ways to split the vector
        // Iterate through each element to find valid splits
        for (int i = 0; i < numCount - 2; ++i) {
            // Use binary search to find the smallest j where the sum of the first part is less than or equal to the sum of the secor
            int j = lower_bound(prefixSum.begin() + i + 1, prefixSum.begin() + numCount - 1, 2 * prefixSum[i]) - prefixSum.begin();
            // Use binary search to find the largest k where the sum of the second part is less than or equal to the sum of the thirc
            int k = upper_bound(prefixSum.begin() + j, prefixSum.begin() + numCount - 1, (prefixSum[numCount - 1] + prefixSum[i]) / 2
            // Add the count of ways between j and k to the answer
            answer = (answer + k - j) % MOD;
        }
        return answer; // Return the final answer
    }
};

```

## Typescript Solution

```typescript
/**
 * Calculate the number of ways to split an array into three non-empty contiguous subarrays
 * where the sum of the first subarray is less than or equal to the sum of the second subarray,
 * which is less than or equal to the sum of the third subarray.
 *
 * @param {number[]} nums - The input array
 * @return {number} - Number of ways to split the array
 */
function waysToSplit(nums: number[]): number {
    const mod: number = 1e9 + 7;
    const n: number = nums.length;
    const prefixSums: number[] = new Array(n).fill(nums[0]);

    // Compute the prefix sum of the array
    for (let i: number = 1; i < n; ++i) {
        prefixSums[i] = prefixSums[i - 1] + nums[i];
    }

    // Binary search function to find the leftmost (or rightmost) index
    function binarySearch(prefixSums: number[], target: number, left: number, right: number): number {
        while (left < right) {
            // Find the middle index
            const mid: number = (left + right) >> 1;

            // Narrow the search range based on the sum comparison
            if (prefixSums[mid] >= target) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }

    // Initialize the answer variable
    let answer: number = 0;

    // Iterate through the array and count valid splits
    for (let i: number = 0; i < n - 2; ++i) {
        // Binary searches to find the eligible split indices j and k
        const j = binarySearch(prefixSums, prefixSums[i] * 2, i + 1, n - 1);
        const k = binarySearch(prefixSums, Math.floor((prefixSums[n - 1] + prefixSums[i]) / 2) + 1, j, n - 1);

        // Update the number of ways to split
        answer = (answer + k - j) % mod;
    }

    // Return the final answer
    return answer;
}

```

## Time and Space Complexity

### Time Complexity

The time complexity of the `waysToSplit` method can be broken down into the following parts:

1. Calculating the prefix sum of the `nums` list using `accumulate`. This takes `O(n)` time, where `n` is the length of `nums`.

2. The outer for-loop runs from `0` to `n - 3` inclusive, which gives us `O(n)` iterations.

3. Inside the for-loop, it uses `bisect_left` and `bisect_right` methods to find indices `j` and `k`. Both of these binary search operations run in `O(log n)` time.

Therefore, with every iteration of the for-loop, we perform two `O(log n)` operations. Since we iterate `O(n)` times, this leads to a compound time complexity of `O(n log n)` for this part of the function.

The total time complexity of the function is therefore `O(n) + O(n log n)` = `O(n log n)`.

### Space Complexity

The space complexity is determined by the additional space used by the algorithm, which includes:

1. The space used by the prefix sum array `s`. This array is the same length as `nums`, so it requires `O(n)` space.

2. Variables used for iteration and indexing, such as `i`, `j`, `k`, `ans`, `mod`, and `n`, which use `O(1)` space.

The dominant term here is the space used by the prefix sum array, so the total space complexity is `O(n)`.