

# 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`.

- We start by fixing the end of the `left` subarray at index `i`.
- 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  $\text{sum}(\text{left}) \leq \text{sum}(\text{mid})$ .
- 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.
- The number of positions for `k` minus the number of positions for `j` gives us the count of good splits for a fixed `i`.
- We loop over all possible ends for the `left` subarray and sum up the counts to get the answer.
- 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:

- 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.
- 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.
- 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.
- 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.
- 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  $2 * s[i] \leq s[j]$ . 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  $s[k] \leq (s[-1] + s[i]) / 2$ . This ensures that  $\text{sum}(\text{mid}) \leq \text{sum}(\text{right})$ .
- 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.
- 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:

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

Following the steps of the solution:

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

```
1 s = [1, 3, 5, 7, 12, 12] // The cumulative sum of 'nums'
```
- Modular Arithmetic Constant:** Define `mod = 10**9 + 7` for later use.
- Initialization:** Initialize `ans = 0` and `n = 6` (length of `nums`).
- 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`:

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

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

- Finding the Lower and Upper Bounds for Mid Subarray Ends:**
  - For each `i`, find `j` and `k` using binary search:

```
1 For i = 0 (s[i] = 1):
2 - Search 'j' such that '2 * s[i] <= s[j]': 'j = 2'
3 - Search 'k' such that 's[k] <= (s[-1] + s[i]) / 2': 'k = 4'
4 - Good splits for i = 0: 'k - j' = '4 - 2' = 2
5
6 For i = 1 (s[i] = 3):
7 - 'j = 3'
8 - 'k = 5'
9 - Good splits for i = 1: 'k - j' = '5 - 3' = 2
10
11 For i = 2 (s[i] = 5):
12 - 'j = 4'
13 - 'k = 5'
14 - Good splits for i = 2: 'k - j' = '5 - 4' = 1
15
16 For i = 3 (s[i] = 7):
17 - 'j = 4'
18 - 'k = 5'
19 - Good splits for i = 3: 'k - j' = '5 - 4' = 1
```
- Counting Good Splits:** Sum up all the good splits:

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

```
1 from itertools import accumulate
2 from bisect import bisect_left, bisect_right
3 from typing import List
4
5 class Solution:
6     def ways_to_split(self, nums: List[int]) -> int:
7         # Constant for modulo operation to prevent overflow
8         MOD = 10**9 + 7
9
10        # Calculate the prefix sum of numbers to efficiently compute sums of subarrays
11        prefix_sums = list(accumulate(nums))
12
13        answer = 0 # Initialize answer which will hold the number of valid ways to split
14        num_elements = len(nums) # Total number of elements in nums
15
16        # Iterate through the array (except the last two elements, as those are needed for the second and third part)
17        for i in range(num_elements - 2):
18            # 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 part
19            j = bisect_left(prefix_sums, 2 * prefix_sums[i], i + 1, num_elements - 1)
20
21            # 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 part
22            k = bisect_right(prefix_sums, (prefix_sums[-1] + prefix_sums[i]) // 2, j, num_elements - 1)
23
24            # Increment the answer by the number of valid ways to split given the current first part
25            answer += k - j
26
27        # Return the total number of ways to split modulo MOD to handle large numbers
28        return answer % MOD
29
```

## Java Solution

```
1 class Solution {
2     private static final int MODULO = (int) 1e9 + 7;
3
4     public int waysToSplit(int[] nums) {
5         int length = nums.length;
6         // Create a prefix sum array
7         int[] prefixSum = new int[length];
8         prefixSum[0] = nums[0];
9         for (int i = 1; i < length; ++i) {
10             prefixSum[i] = prefixSum[i - 1] + nums[i];
11         }
12
13         int countWays = 0; // This will store the number of ways to split the array
14         // Loop through each possible index to split the array after
15         for (int i = 0; i < length - 2; ++i) {
16             // Find the earliest index to make the second part's sum at least equal to the first part
17             int earliest = binarySearchLeft(prefixSum, 2 * prefixSum[i], i + 1, length - 1);
18             // Find the latest index where the third part's sum would be at least as much as the second
19             int latest = binarySearchRight(prefixSum, ((prefixSum[length - 1] + prefixSum[i]) / 2) + 1, earliest, length - 1);
20             // Add the number of ways to split between these two indices
21             countWays = (countWays + latest - earliest) % MODULO;
22         }
23         return countWays;
24     }
25
26     private int binarySearchLeft(int[] prefixSum, int target, int leftIndex, int rightIndex) {
27         // Standard binary search to find the left-most index where prefixSum[index] >= target
28         while (leftIndex < rightIndex) {
29             int mid = (leftIndex + rightIndex) / 2;
30             if (prefixSum[mid] >= target) {
31                 rightIndex = mid;
32             } else {
33                 leftIndex = mid + 1;
34             }
35         }
36         return leftIndex;
37     }
38
39     private int binarySearchRight(int[] prefixSum, int target, int leftIndex, int rightIndex) {
40         // Binary search to find the right-most position to meet the constraints
41         while (leftIndex < rightIndex) {
42             int mid = (leftIndex + rightIndex) / 2;
43             if (prefixSum[mid] < target) {
44                 leftIndex = mid + 1;
45             } else {
46                 rightIndex = mid;
47             }
48         }
49         // We might overshoot by one, so we adjust if necessary
50         if (leftIndex == rightIndex && prefixSum[leftIndex] >= target) {
51             return leftIndex;
52         }
53         return leftIndex - 1;
54     }
55 }
56
```

## C++ Solution

```
1 class Solution {
2 public:
3     const int MOD = 1e9 + 7; // Defining the modulo constant for operations
4
5     int waysToSplit(vector<int& nums) {
6         int numCount = nums.size(); // n is the total number of elements in nums
7         // Create a prefix sum array where each element s[i] is the sum of nums[0] to nums[i]
8         vector<int> prefixSum(numCount, nums[0]);
9         for (int i = 1; i < numCount; ++i) {
10             prefixSum[i] = prefixSum[i - 1] + nums[i];
11         }
12         int answer = 0; // To store the final count of ways to split the vector
13         // Iterate through each element to find valid splits
14         for (int i = 0; i < numCount - 2; ++i) {
15             // 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 second part
16             int j = lower_bound(prefixSum.begin() + i + 1, prefixSum.begin() + numCount - 1, 2 * prefixSum[i]) - prefixSum.begin();
17             // 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 third part
18             int k = upper_bound(prefixSum.begin() + j, prefixSum.begin() + numCount - 1, (prefixSum[numCount - 1] + prefixSum[i]) / 2) - prefixSum.begin();
19             // Add the count of ways between j and k to the answer
20             answer = (answer + k - j) % MOD;
21         }
22         return answer; // Return the final answer
23     }
24 };
25
```

## Typescript Solution

```
1 /**
2  * Calculate the number of ways to split an array into three non-empty contiguous subarrays
3  * where the sum of the first subarray is less than or equal to the sum of the second subarray,
4  * and the sum of the second subarray is less than or equal to the sum of the third subarray.
5  *
6  * @param {number[]} nums - The input array
7  * @return {number} - Number of ways to split the array
8  */
9 function waysToSplit(nums: number[]): number {
10     const mod: number = 1e9 + 7;
11     const n: number = nums.length;
12     const prefixSums: number[] = new Array(n).fill(nums[0]);
13
14     // Compute the prefix sum of the array
15     for (let i: number = 1; i < n; ++i) {
16         prefixSums[i] = prefixSums[i - 1] + nums[i];
17     }
18
19     // Binary search function to find the leftmost (or rightmost) index
20     function binarySearch(prefixSums: number[], target: number, left: number, right: number): number {
21         while (left < right) {
22             // Find the middle index
23             const mid: number = (left + right) >> 1;
24
25             // Narrow the search range based on the sum comparison
26             if (prefixSums[mid] >= target) {
27                 right = mid;
28             } else {
29                 left = mid + 1;
30             }
31         }
32         return left;
33     }
34
35     // Initialize the answer variable
36     let answer: number = 0;
37
38     // Iterate through the array and count valid splits
39     for (let i: number = 0; i < n - 2; ++i) {
40         // Binary searches to find the eligible split indices j and k
41         const j = binarySearch(prefixSums, prefixSums[i] * 2, i + 1, n - 1);
42         const k = binarySearch(prefixSums, Math.floor((prefixSums[n - 1] + prefixSums[i]) / 2) + 1, j, n - 1);
43
44         // Update the number of ways to split
45         answer = (answer + k - j) % mod;
46     }
47
48     // Return the final answer
49     return answer;
50 }
51
```

## Time and Space Complexity

### Time Complexity

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

- Calculating the prefix sum of the `nums` list using `accumulate`. This takes  $O(n)$  time, where `n` is the length of `nums`.
- The outer for-loop runs from 0 to `n - 3` inclusive, which gives us  $O(n)$  iterations.
- 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:

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