1524. Number of Sub-arrays With Odd Sum

Dynamic Programming

problems to avoid issues with large number handling.

The task is to count how many subarrays (continuous parts of the array) of a given array of integers have a sum that is odd. It's

<u>Array</u>

Problem Description

Math

Medium

then [1, 2] and [2, 3] are subarrays, but [1, 3] is not a subarray because it doesn't preserve the order of elements. Since the number of possible subarrays for even a moderate-sized array is quite large, and therefore the result could be a very large number, the problem asks to return the count of such subarrays modulo 10^9 + 7. This is a common requirement for such

Prefix Sum

important to note that the array consists of integers which may be positive, negative, or zero.

In short, we need to find the total count of subarrays with odd sums and then report this count modulo $10^9 + 7$.

Subarrays are defined as slices of the original array that maintain the order of elements. For example, if the array is [1, 2, 3],

When we want to solve a problem related to subarrays and their sums, a common approach is to think in terms of prefix sums. A prefix sum is the cumulative sum of elements from the start of the array up to a given point. It allows us to quickly calculate the

sum of any subarray by subtracting the prefix sum up to the start of the subarray from the prefix sum up to the end of the

(which is the zero-sum before we start the sum, equivalent to an empty subarray), and no odd sums.

To solve this problem, we initialize a count array cnt of size 2, where cnt[0] keeps track of the number of even prefix sums

arrays.

Solution Approach

subarray.

even = odd and even - odd = odd to determine if the current prefix sum would lead to a subarray with an odd sum, given previous prefix sums. The expression (s & 1) gives us 1 if s (the current prefix sum) is odd, and 0 if it's even. We use this to update our cnt array to count the occurrences of even and odd prefix sums.

encountered so far and cnt[1] keeps track of the odd ones. Initially, cnt is set to [1, 0], meaning we have one even prefix sum

We iterate through the input array arr and update our current running sum s. At each element, we use the property that odd -

For each new element x added to our running sum s, we look at "s & 1 ^ 1", which essentially flips the parity; if s is even, we look at odd counts (cnt[1]), if s is odd, we look at even counts (cnt[0]). We add this to our answer ans, which keeps track of the total count of subarrays with odd sum encounters so far. We then update the cnt array to reflect the new counts after

considering the current element. Lastly, we return ans, the total count, modulo 10^9 + 7 to keep the number within bounds per the problem statement's requirements. This solution takes O(n) time, where n is the number of elements in arr, and O(1) extra space, making it efficient for even large

The solution uses a single pass through the array and leverages bitwise operations and modular arithmetic to maintain and calculate the count of subarrays with odd sums efficiently. Here's a step-by-step walkthrough:

Initialize mod to 10^9 + 7. This will be used to perform modular arithmetic to ensure that the result remains within integer limits. Declare a list cnt with [1, 0] which represents the count of prefix sums that are even and odd. cnt [0] is initially 1 because

Initialize ans (the variable to store the final count of subarrays with odd sums) to 0 and s (the running prefix sum) to 0.

a. Add x to the running prefix sum s. b. Calculate (s & 1) to get the parity of the prefix sum s. s & 1 will be 1 for odd and 0 for even. This is a standard way to

vice versa. d. Update ans by adding the count of prefix sums of the opposite parity (cnt[s & 1 ^ 1]). It's important to note that if we

have an odd <u>prefix sum</u> now and an even prefix sum previously, the subarray between the two will have an odd sum.

The code uses a for loop that iterates through every element in the array, maintaining a running prefix sum which allows us to

determine, at each step, how many subarrays ended at this element have an odd sum. The use of the prefix sum pattern is crucial

In terms of data structures, the solution keeps a small array cnt to count even and odd sums, rather than a prefix sum array,

which would normally be used in prefix sum problems. This is an optimization that takes advantage of the problem's specifics

The code is concise and efficient, running in O(n) time complexity with constant space usage, which is optimal for this problem.

Let's illustrate the solution approach using a small example array: arr = [1, 2, 3, 4]. We're tasked with finding the count of

c. Calculate s & 1 ^ 1 to flip the current parity. We use bitwise XOR here to flip 0 to 1 and 1 to 0. This is because if the

current prefix sum s is even, we are interested in the count of previously seen odd prefix sums to form an odd subarray and

- f. Increment the count of the appropriate parity in cnt by 1 (cnt[s & 1]).
- here because it helps in determining the sum of subarrays in constant time without the need to recalculate sums for each potential subarray.

(only the parity of the sums is relevant).

subarrays that have an odd sum.

a. For x = 1: s becomes 1.

e. Apply a modulo to ans if needed.

For x = 2:

a. s becomes 3.

a. s becomes 6.

is the new odd subarray).

Initialize mod to $10^9 + 7$ for modular arithmetic.

Start iterating through each element x in arr.

ans is set to 0 and s (the running prefix sum) is also 0.

odd sum subarray (in this case, the subarray is just [1]).

After the first iteration: s = 1, cnt = [1, 1], ans = 1.

b. (s & 1) gives us 1, so the current prefix sum is odd.

f. Increment cnt[1] since we now have an odd prefix sum (1).

e. Apply the modulo operation to ensure that ans does not overflow integer limits.

After the loop, return ans, which now contains the moduloed count of subarrays with odd sums.

a <u>prefix sum</u> of (before processing any elements) is even.

check for even or odd numbers using bitwise AND operation.

Iterate through each element x in the array arr.

Example Walkthrough

cnt starts as [1, 0], indicating one even prefix sum (0 from no elements) and no odd prefix sums.

b. (s & 1) gives us 1, indicating the current prefix sum is odd. c. s & 1 ^ 1 yields 0, so we look at the even counts from cnt. d. ans is updated with cnt[0], which is 1, because adding an odd number (1) to any even prefix sums would result in an

d. ans is updated with cnt[0], which is 1, because an even number added to an odd prefix sum keeps the sum odd ([1, 2]

d. ans is updated with cnt[1], which is 2. The addition of an odd number (3) to the previous odd prefix sums creates new

d. ans is updated with cnt[1], which is 2. Even prefix sums followed by an even number do not change the parity ([2, 3,

Return ans, which is 6. This means there are 6 subarrays with an odd sum in arr = [1, 2, 3, 4]. The result is given modulo

- c. s & 1 ^ 1 yields 0, so we look at the even counts from cnt.
- e. Apply a modulo to ans.

f. Increment cnt[1] since our running sum is still odd.

- After the second iteration: s = 3, cnt = [1, 2], ans = 2. For x = 3:
- b. (s & 1) gives us 0, so the current prefix sum is even. c. s & 1 ^ 1 yields 1, now we look at the odd counts from cnt.

odd sum subarrays ([1, 2, 3] and [3]).

f. Increment cnt[0] because s is now even.

e. Apply a modulo to ans.

For x = 4:

 $10^9 + 7$.

Python

class Solution:

Solution Implementation

mod = 10**9 + 7

answer = 0

prefix_sum = 0

for num in arr:

a. s becomes 10. b. (s & 1) gives us 0, meaning an even prefix sum.

c. s & 1 ^ 1 flips to 1, looking at odd counts in cnt.

4] and [4] are the new subarrays).

f. Increment cnt[0] as s remains even.

Final state: s = 10, cnt = [3, 2], ans = 6.

def numOfSubarrays(self, arr: List[int]) -> int:

Initialize answer and prefix sum variables

Iterate over the array elements

count[prefix_sum % 2] += 1

Update the prefix sum

prefix_sum += num

Return the final answer

public int numOfSubarrays(int[] arr) {

// Variable to store the final answer

// Variable to store the current sum

// Iterate through each element in the array

final int MOD = 1000000007;

int[] count = {1, 0};

for (int num : arr) {

++count[sum & 1];

// Return the final answer

for (int number : arr) {

++count[sum % 2];

sum += num;

int answer = 0;

int sum = 0;

e. Apply a modulo to ans.

After the third iteration: s = 6, cnt = [2, 2], ans = 4.

The solution approach effectively identifies all possible subarrays with odd sums by keeping track of prefix sums and their parity, using a simple count array and adding the counts of prefix sums with the opposite parity when encountering a new element.

Hence, we achieve an O(n) time complexity solution with constant space utilization.

Define a large number for modulo operation to prevent integer overflow

Increment the answer by the number of subarrays encountered so far

that when added to the current element yields an even sum

answer = (answer + count[prefix_sum % 2 ^ 1]) % mod

// Method to calculate the number of subarrays with odd sum

// Initialize the modulus value as per the problem statement

// Add the current element's value to the cumulative sum

// Increment the count of current parity (even/odd) of sum

sum += number; // Add the current number to the cumulative sum

Define a large number for modulo operation to prevent integer overflow

Increment the answer by the number of subarrays encountered so far

that when added to the current element yields an even sum

answer = (answer + count[prefix_sum % 2 ^ 1]) % mod

Update the count of subarrays with current sum parity

count = [1, 0] # Even sums are initialized to 1 due to the virtual prefix sum of 0 at the start

Initialize the count of subarrays with even and odd sum

// Increment the count of the current sum's parity (1 for odd, 0 for even)

// Then, take the modulo to handle large numbers

answer = (answer + count[1 - (sum & 1)]) % MOD;

// Iterate over each element in the input array

def numOfSubarrays(self, arr: List[int]) -> int:

Initialize answer and prefix sum variables

Iterate over the array elements

count[prefix_sum % 2] += 1

Update the prefix sum

prefix_sum += num

Return the final answer

Time and Space Complexity

// This is because an odd sum — even sum = odd

answer = (answer + count[sum % 2 ^ 1]) % MOD;

Update the count of subarrays with current sum parity

count = [1, 0] # Even sums are initialized to 1 due to the virtual prefix sum of 0 at the start

// Counter array to track even and odd sums, where index 0 is for even and index 1 is for odd

// If the cumulative sum is odd, add the count of previous even sums to the answer.

// If the cumulative sum is even, add the count of previous odd sums to the answer.

Initialize the count of subarravs with even and odd sum

return answer Java class Solution {

int numOfSubarrays(vector<int>& arr) { const int MOD = 1000000007; // The modulus value for avoiding integer overflow int count[2] = $\{1, 0\}$; // Initialize count array to keep track of even (count[0]) and odd (count[1]) sums int answer = 0; // This will store the final result, the number of subarrays with an odd sum int sum = 0; // This variable will store the cumulative sum of the elements

C++

public:

class Solution {

return answer;

return answer; // Return the total count of subarrays with an odd sum **}**; **TypeScript** function numOfSubarrays(arr: number[]): number { let answer = 0; // Initializes the answer to 0. let cumulativeSum = 0; // Tracks the cumulative sum of the elements. const count: number[] = [1, 0]; // count[0] represents count of even cumulative sums, count[1] represents count of odd cumulativ const MOD = 1e9 + 7; // Define the modulo value for the answer. // Iterate through the given array. for (const element of arr) { cumulativeSum += element; // Update the cumulative sum with the current element. // Increment the answer with the amount of even cumulative sums if the current cumulative sum is odd // or the amount of odd cumulative sums if the current cumulative sum is even, then take modulo. answer = (answer + count[(cumulativeSum & 1) ^ 1]) % MOD; // Increment the count of even/odd cumulative sums as appropriate. count[cumulativeSum & 1]++; // Return the final answer. return answer;

// Increment the answer by the count of the opposite parity (even if sum is odd, odd if sum is even)

Time Complexity

class Solution:

mod = 10**9 + 7

answer = 0

prefix_sum = 0

for num in arr:

return answer

that iterates through the array once, and the operations within the loop (calculating the sum s, updating ans, and managing cnt array) have constant time complexity. **Space Complexity**

The time complexity of the code is O(n), where n is the length of the input array arr. This is because there is a single for-loop

The space complexity of the code is 0(1). The additional space used by the algorithm is limited to a few variables (mod, cnt, ans, s) and does not depend on the size of the input array. The cnt array is of fixed size 2, which stores the count of cumulative sums that are even and odd.