1588. Sum of All Odd Length Subarrays

Easy Array Math Prefix Sum

Problem Description

The given problem entails finding the sum of all odd-length subarrays from a given array of positive integers. An odd-length subarray is a contiguous part of the original array that has an odd number of elements. For example, in the array `[1, 2, 3, 4]`, the odd-length subarrays include `[1]`, `[2]`, `[3]`, `[4]`, `[1, 2, 3]`, `[2, 3, 4]`, and the entire array itself `[1, 2, 3, 4]`. The goal is to calculate the sum of all the elements from all such subarrays and return this sum.

Intuition

The intuition behind the solution is to consider each element of the array as a potential starting point of an odd-length subarray and expand the subarray one element at a time as we iterate through the array. For each starting element, we add the elements to the subarray until we reach the end of the array. We check if the length of the current subarray is odd, and if it is, we add the sum of the current subarray to our answer.

As we iterate through the array `arr` of size `n`, we initialize a variable `ans` to store the sum of odd-length subarrays. For each element `arr[i]`, we initialize a subarray sum `s` to track the sum of elements starting from `arr[i]`. Then for each element `arr[j]` to the right of `arr[i]` (including `i`), we add to `s`. After each addition, we check if the subarray from `arr[i]` to `arr[j]` has an odd length by checking if the length `(j - i + 1)` is odd (using bitwise `& 1` to check for oddness). If it's odd, we add the current sum `s` to our overall answer `ans`.

By systematically expanding and checking each potential subarray based on its starting point, we ensure that we consider all possible odd-length subarrays. The iterative approach is straightforward and does not involve complex data structures or algorithms.

Solution Approach

In the implementation of the solution, the chosen approach is a brute force method where two nested loops are utilized to calculate the sum of all possible odd-length subarrays. Here's the step-by-step process:

1. Initialize an accumulator variable `ans` to `0` which will hold the final sum of all odd-length subarrays.
2. Determine the length `n` of the array `arr`.
3. Start the first loop with the variable `i` iterating from `0` up to `n-1`. This loop will help us select the starting element of each subarray.
4. Inside the first loop, initialize a variable `s` to `0`. This variable will keep track of the sum of the current subarray being considered.
5. Start the second nested loop with the variable `j` iterating from `i` up to `n-1`. This loop will extend the subarray one element at a time from the starting index `i`.
6. Inside the nested loop, add the current element `arr[j]` to the sum `s`.
7. Immediately after adding to `s`, check if the subarray from `i` to `j` is of odd length. This check is performed using the bitwise AND operation `(j - i + 1) & 1`. If `(j - i + 1) & 1` equals `1`, this means the length of the subarray (`j - i + 1`) is odd.
8. If the subarray length is found to be odd, increment `ans` by the current sum `s`.
9. The nested loop ends after reaching the final element, and all possible subarrays starting at index `i` have been considered.
10. Repeat the process for every starting index `i` until all starting positions have been accounted for.

While this solution is not the most optimized, it is easy to understand and implement. The time complexity for this approach is O(n^2), which is acceptable when `n` is not excessively large. It avoids the use of any additional complex data structures maintaining the simplicity of implementation. No patterns or algorithms other than straightforward conditional statements and arithmetic operations are used.

Example Walkthrough

Let's walk through the brute force solution approach using a small example array: `[1, 2, 3]`.

1. Initialize `ans` to `0`. This variable will store our final answer.
2. The length `n` of the array `[1, 2, 3]` is `3`.
3. Start the first loop with the variable `i`. For `i = 0`, we will look at subarrays starting with the element `1`.
4. Inside the loop for `i = 0`, initialize `s` to `0`. This is where we'll accumulate the sum of elements for our current subarray.
5. The second nested loop starts with `j = i`. For `j = 0` which means our subarray is `[1]`. We add `arr[j]`, which is `1` to `s`, resulting in `s = 1`.
6. We check whether the length of the subarray (`j - i + 1`) is odd, which is `1` in this case. Since `(1 & 1) == 1`, the condition is `true`.
7. Because our subarray length is odd, we increment `ans` by `s`, so `ans = ans + 1 = 1`.
8. We increment `j` to `1`, and now consider the subarray `[1, 2]`. Adding `arr[j] = 2` to `s` gives us `s = 3`. However, `(2 - 0 + 1) & 1 == 0`, the length `3` is not odd, thus we do nothing.
9. Increment `j` to `2` and our subarray becomes `[1, 2, 3]`. We add `arr[j] = 3` to `s` to get `s = 6`. The subarray length is `3` which is odd: `(3 - 0 + 1) & 1 == 1`. We add `6` to `ans`, making `ans = 1 + 6 = 7`.
10. We've considered all elements for the starting index `i = 0`. Now we increment `i` to `1` and repeat the process for the new starting element, `2`.

Through this repeated process, we would continue iterating over all starting indices, finding all possible odd-length subarrays, and summing their elements to `ans`. The final value of `ans` after considering all starting points would be the answer to our problem: the sum of all elements in all odd-length subarrays of the array `[1, 2, 3]`. Using this method with our example would yield an `ans` value of `12`, accounting for the sums of subarrays `[1]`, `[2]`, `[3]`, `[1, 2, 3]`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def sumOddLengthSubarrays(self, arr: List[int]) -> int:
        # Initialize the total_sum to accumulate the sum of all odd length subarrays
        total_sum = 0

        # Get the length of the array
        n = len(arr)

        # Loop over each element in the array as the starting point of the subarrays
        for start_index in range(n):
            # Initialize subarray_sum to hold the sum of the current subarray
            subarray_sum = 0

            # Loop over each element from the start_index to the end of the array
            for end_index in range(start_index, n):
                # Add the current element to subarray_sum
                subarray_sum += arr[end_index]

                # Calculate the length of the current subarray
                subarray_length = end_index - start_index + 1

                # Check if the length of the subarray is odd
                if subarray_length % 2 == 1:
                    # If it's odd, add the current subarray sum to total_sum
                    total_sum += subarray_sum

        # Return the total sum of all odd length subarrays
        return total_sum

# Example usage:
# solution = Solution()
# result = solution.sumOddLengthSubarrays([1,4,2,5,3])
# print(result)  # Output would be 58
```

Java

```java
class Solution {
    public int sumOddLengthSubarrays(int[] arr) {
        int n = arr.length; // The length of the input array
        int totalSum = 0; // This will hold the sum of all odd-length subarrays

        // We iterate over each element of the array as the start point for our subarrays
        for (int startIndex = 0; startIndex < n; ++startIndex) {
            int subarraySum = 0; // Holds the sum of the current subarray

            // We increase the end point of our subarray one element at a time
            for (int endIndex = startIndex; endIndex < n; ++endIndex) {
                // Add the current element to our subarray sum
                subarraySum += arr[endIndex];

                // Check if the length of the current subarray is odd
                if ((endIndex - startIndex + 1) % 2 == 1) {
                    // If it's odd, add the current subarray sum to the total sum
                    totalSum += subarraySum;
                }
            }
        }

        // Return the total sum of all odd-length subarrays
        return totalSum;
    }
}
```

C++

```cpp
#include <vector>

class Solution {
public:
    // Calculate the sum of all odd-length subarrays in the given array.
    int sumOddLengthSubarrays(std::vector<int>& arr) {
        int n = arr.size();     // Get the size of the array.
        int totalSum = 0;       // Initialize the total sum of odd-length subarrays.

        // Traverse the array starting from each element.
        for (int startIndex = 0; startIndex < n; ++startIndex) {
            int subarraySum = 0; // Initialize the sum for the current subarray.

            // Extend the subarray from the start index to the end of the array.
            for (int endIndex = startIndex; endIndex < n; ++endIndex) {
                // Add the current element to the subarray sum.
                subarraySum += arr[endIndex];

                // If the length of the current subarray (endIndex - startIndex + 1) is odd...
                if ((endIndex - startIndex + 1) % 2 == 1) {
                    // ...then add the current subarray sum to the total sum.
                    totalSum += subarraySum;
                }
            }
        }
        return totalSum;  // Return the accumulated sum of all odd-length subarrays.
    }
};
```

TypeScript

```typescript
function sumOddLengthSubarrays(arr: number[]): number {
    const arrLength = arr.length; // Store the length of the input array.
    let totalSum = 0; // Initialize total sum of all odd length subarrays.

    // Iterate through each element of the array.
    for (let startIndex = 0; startIndex < arrLength; ++startIndex) {
        let subarraySum = 0; // Initialize sum for the current subarray.

        // Form subarrays starting with the element at startIndex.
        for (let endIndex = startIndex; endIndex < arrLength; ++endIndex) {
            subarraySum += arr[endIndex]; // Add the current element to the subarray sum.

            // Check if the length of the current subarray is odd.
            if ((endIndex - startIndex + 1) % 2 === 1) {
                totalSum += subarraySum; // If it's odd, add the subarray's sum to the total sum.
            }
        }
    }

    return totalSum; // Return the total sum of all odd length subarrays.
}
```

```python
from typing import List

class Solution:
    def sumOddLengthSubarrays(self, arr: List[int]) -> int:
        # Initialize the total_sum to accumulate the sum of all odd length subarrays
        total_sum = 0

        # Get the length of the array
        n = len(arr)

        # Loop over each element in the array as the starting point of the subarrays
        for start_index in range(n):
            # Initialize subarray_sum to hold the sum of the current subarray
            subarray_sum = 0

            # Loop over each element from the start_index to the end of the array
            for end_index in range(start_index, n):
                # Add the current element to subarray_sum
                subarray_sum += arr[end_index]

                # Calculate the length of the current subarray
                subarray_length = end_index - start_index + 1

                # Check if the length of the subarray is odd
                if subarray_length % 2 == 1:
                    # If it's odd, add the current subarray sum to total_sum
                    total_sum += subarray_sum

        # Return the total sum of all odd length subarrays
        return total_sum

# Example usage:
# solution = Solution()
# result = solution.sumOddLengthSubarrays([1,4,2,5,3])
# print(result)  # Output would be 58
```

Time and Space Complexity

The provided Python function `sumOddLengthSubarrays` computes the sum of elements of all odd length subarrays of the given array `arr`. Let's analyze both time complexity and space complexity:

Time Complexity

The time complexity of the code is determined by the two nested loops. The outer loop runs from `0` to `n-1`, where `n` is the length of the array. The inner loop starts from the current index of the outer loop `i` and goes to `n-1`.

Since for every element in the array, we are iterating over all subsequent elements (progressively fewer as `i` increases), the number of operations can be approximated by the sum of an arithmetic series.

The total number of operations is close to `1 + 2 + ... + n`, which is given by the formula `(n*(n+1))/2`. But since we are only adding to the sum for odd indices, we can estimate roughly half of these operations are meaningful, giving us an estimate of `(n*(n+1))/4`. However, the presence of odd or even length subarrays does not change the fact that each element is considered in the sum precisely `(n*(n+1))/2` times.

Thus, the time complexity is `O(n^2)`.

Space Complexity

The space complexity is determined by the amount of additional memory used by the algorithm, which is independent of the size of the input array `arr`. Only a fixed number of individual integer variables `ans`, `n`, `s`, `i`, and `j` are used.

No additional data structures are dependent on the input size, hence the space complexity is `O(1)`, that is, constant.