1248. Count Number of Nice Subarrays

Medium | Array | Hash Table | Math | Sliding Window

Problem Description

The problem provides us with an array `nums` of integers and an integer `k`. We need to count how many continuous subarrays (a sequence of adjacent elements in the array) have exactly `k` odd numbers. These subarrays are referred to as **nice** subarrays.

The main challenge is to do this efficiently for possibly large arrays.

Intuition

To solve this problem, we can apply the concept of **prefix sums**; however, instead of keeping a running total of the elements, we'll keep a running count of how many odd numbers we've encountered so far. The key idea is to use a map (`Counter` in Python) to keep track of how many times each count of odd numbers has occurred in our running total (prefix).

Let's say we're iterating through the array and at some point we have encountered `x` odd numbers. Now, if at an earlier point in the array we had encountered `x - k` odd numbers, any subarray starting from that earlier point to our current position will have exactly `k` odd numbers. Why? Because `x - (x - k) = k`.

So, we use a counter to keep track of all previously seen counts of odd numbers. Every time we hit a new odd number, we increment our current count (`t` in the code). We then check in our counter to see how many times we've previously seen a count of `t - k`. The value we get tells us the number of subarrays ending at the current position with exactly `k` odd numbers.

We add this count to our answer (`ans` in the code) and move to the next element. Since the subarrays are continuous, we increment our counter for the current count of odd numbers to reflect that we found a new subarray starting point that ends at the current index.

The `Counter` is initialized with `{0: 1}` to handle the case where the first `k` elements form a **nice** subarray. The count is zero (no odd numbers yet), but we consider it as a valid starting point.

The solution iteratively applies the above logic, using bitwise operations to check if a number is odd (`v & 1`) and updating the count and answer accordingly.

Solution Approach

The solution uses a `Counter` dictionary to store the frequency of counts of odd numbers encountered as a prefix sum. This `Counter` represents how many times each count has been seen so far in the array, which is essential for determining the number of **nice** subarrays.

Here's a step-by-step breakdown of the implementation:

1. Initialize a `Counter` with `{0: 1}`. This accounts for the prefix sum before we start traversing the array. The value `1` indicates that there's one way to have zero odd numbers so far (essentially, the subarray of length zero).
2. Initialize two variables, `ans` and `t` to `0`. `ans` will hold the final count of **nice** subarrays, while `t` will keep the running count of odd numbers seen.
3. Loop through each number `v` in the array `nums`.

   a. Use the bitwise AND operation (`v & 1`) to determine if `v` is odd. If so, add `1` to `t`. The operation `v & 1` evaluates to `1` if `v` is odd, and `0` if `v` is even.

   b. Calculate `t - k`, which represents the prefix sum we'd have if we subtracted `k` odd numbers from the current count. If this value has been seen before (which means there's a subarray ending at an earlier position with `t - k` odd numbers), we can form a **nice** subarray ending at the current index. Add the count of `t - k` from the `Counter` to `ans`.

   c. Increment the count of `t` in the `Counter` by `1` to indicate that we have encountered another subarray ending at the current index with a prefix sum of `t` odd numbers.
4. After the loop completes, `ans` holds the total count of **nice** subarrays, and this is what is returned.

Here is how the algorithm works in a nutshell: By maintaining a running count of odd numbers, and having a `Counter` to record how many times each count has occurred, we can efficiently compute how many subarrays end at a particular index with exactly `k` odd numbers. This lets us add up the counts for all subarrays in the array just by iterating through it once.

Example Walkthrough

Let's consider an example where `nums = [1, 2, 3, 4, 5]` and `k = 2`. We want to find out how many continuous subarrays have exactly 2 odd numbers.

Following the solution approach:

1. We initialize a `Counter` with `{0: 1}` to count the number of times we have encountered 0 odd numbers so far (which is once for the empty subarray).
2. We set `ans` and `t` to `0`. `ans` will count our **nice** subarrays, and `t` will hold our running tally of odd numbers seen.
3. We start iterating through each number `v` in `nums`.
   - For `v = 1`: It's odd (since `1 & 1` is `1`), so we increment `t` to `1`. We then look for `t - k`, which is `-1`. Since `-1` is not in our counter, there are `0` subarrays ending here with 2 odd numbers. We update the counter to `{0: 1, 1: 1}`.
   - For `v = 2`: It's even (since `2 & 1` is `0`), so `t` remains `1`. Looking for `t - k` (`1-2=-1`) yields `0` subarrays. Our counter doesn't change.
   - For `v = 3`: It's odd, `t` becomes `2`. Now, `t - k` is `0`, and since our counter shows `{0: 1}`, there is `1` subarray ending here with exactly 2 odd numbers. We increment `ans` to `1` and update the counter to `{0: 1, 1: 1, 2: 1}`.
   - For `v = 4`: It's even, so `t` stays `2`. We look for `t - k` (`2-2=0`) in the counter, find `1` occurrence, so there's `1` more subarray that meets the criteria. Increment `ans` to `2`, and the counter remains the same.
   - For `v = 5`: Again, it's odd, `t` increases to `3`. Looking for `t - k` (`3-2=1`), we find `1` in the counter. This gives us `1` subarray. `ans` goes up to `3`, and we update the counter to `{0: 1, 1: 1, 2: 1, 3: 1}`.
4. After iterating through the array, we have `ans` as `3`, meaning there are three subarrays within `nums` that contain exactly 2 odd numbers. These subarrays are `[1, 2, 3]`, `[3, 4]`, and `[5]`.

So the function would return `3` as the result. The efficiency of the solution stems from the fact that we only needed to traverse the array once and keep a running tally of our counts in the `Counter`, which gives us the ability to find the number of nice subarrays in constant time for each element of the array.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def numberOfSubarrays(self, nums: List[int], k: int) -> int:
        # Initialize counter for odd counts with 0 count as 1 (base case)
        odd_count = Counter({0: 1})

        # Initialize variables for the answer and the temporary count of odds
        answer = temp_odd_count = 0

        # Iterate over each value in the list
        for value in nums:
            # Increment temp_odd_count if value is odd
            temp_odd_count += value & 1  # value & 1 is 1 if value is odd, 0 otherwise

            # If there are at least k odd numbers, add the count to answer
            # This checks if a valid subarray ending at the current index exists
            answer += odd_count[temp_odd_count - k]

            # Increment the count of the current number of odd integers seen so far
            odd_count[temp_odd_count] += 1

        # Return the total number of valid subarrays
        return answer
```

Java

```java
class Solution {
    public int numberOfSubarrays(int[] nums, int k) {
        int n = nums.length; // Length of the input array
        int[] prefixOddCount = new int[n + 1]; // Array to keep track of the prefix sums of odd numbers
        prefixOddCount[0] = 1; // There's one way to have zero odd numbers - by taking no elements
        int result = 0; // Initialize the result count to 0
        int currentOddCount = 0; // Tracks the current number of odd elements encountered

        // Iterate over each number in the input array
        for (int num : nums) {
            // If 'num' is odd, increment the count of odd numbers encountered so far
            currentOddCount += num & 1;

            // If we have found at least 'k' odd numbers so far
            if (currentOddCount - k >= 0) {
                // Add to 'result' the number of subarrays that have 'k' odd numbers up to this point
                result += prefixOddCount[currentOddCount - k];
            }

            // Increment the count in our prefix sum array for the current odd count
            prefixOddCount[currentOddCount]++;
        }
        return result; // Return the total count of subarrays with exactly 'k' odd numbers
    }
}
```

C++

```cpp
#include <vector>

class Solution {
public:
    int numberOfSubarrays(std::vector<int>& nums, int k) {
        int size = nums.size(); // Store the size of the input vector nums
        std::vector<int> count(size + 1, 0); // Vector to store the count of odd numbers
        count[0] = 1; // There's one way to have zero odd numbers - empty subarray

        int answer = 0; // Initialize the count of valid subarrays
        int oddCount = 0; // Counter for the number of odd numbers in the current sequence

        // Iterate through the input numbers
        for (int num : nums) {
            oddCount += num & 1; // Increment the odd count if num is odd
            // If there have been at least k odd numbers so far, update the answer
            if (oddCount >= k) {
                answer += count[oddCount - k];
            }
            count[oddCount]++; // Increment the count for this number of odd numbers
        }

        return answer; // Return the total number of valid subarrays
    }
};
```

TypeScript

```typescript
function numberOfSubarrays(nums: number[], k: number): number {
    // The length of the input array.
    const arrayLength = nums.length;

    // An array to store the count of subarrays with odd numbers sum.
    const countOfSubarrays = new Array(arrayLength + 1).fill(0);

    // Base condition: There is always 1 subarray with 0 odd numbers (the empty subarray).
    countOfSubarrays[0] = 1;

    // The answer to return that accumulates the number of valid subarrays.
    let numberOfValidSubarrays = 0;

    // Tracks the total number of odd numbers encountered so far.
    let totalOdds = 0;

    // Iterate through all elements in the array.
    for (const value of nums) {
        // If value is odd, increment the count of totalOdds.
        totalOdds += value & 1;

        // If there are enough previous odds to form a valid subarray, increment the count.
        if (totalOdds - k >= 0) {
            numberOfValidSubarrays += countOfSubarrays[totalOdds - k];
        }

        // Increment the count of subarrays we've seen with the current totalOdds.
        countOfSubarrays[totalOdds] += 1;
    }

    // Return the total number of valid subarrays found.
    return numberOfValidSubarrays;
}
```


```python
from collections import Counter

class Solution:
    def numberOfSubarrays(self, nums: List[int], k: int) -> int:
        # Initialize counter for odd counts with 0 count as 1 (base case)
        odd_count = Counter({0: 1})

        # Initialize variables for the answer and the temporary count of odds
        answer = temp_odd_count = 0

        # Iterate over each value in the list
        for value in nums:
            # Increment temp_odd_count if value is odd
            temp_odd_count += value & 1  # value & 1 is 1 if value is odd, 0 otherwise

            # If there are at least k odd numbers, add the count to answer
            # This checks if a valid subarray ending at the current index exists
            answer += odd_count[temp_odd_count - k]

            # Increment the count of the current number of odd integers seen so far
            odd_count[temp_odd_count] += 1

        # Return the total number of valid subarrays
        return answer
```


Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the number of elements in the `nums` list. This is because the code iterates through each element of the array exactly once.

During this iteration, the code performs constant-time operations: using bitwise AND to determine the parity of the number, updating the count hashtable, and incrementing the result. The lookup and update of the counter `cnt` are also expected to be constant time because it is a hashtable.

The space complexity of the code is `O(n)`. In the worst case, if all elements in `nums` are odd, then the counter `cnt` can potentially have as many entries as there are elements in `nums`. Therefore, the size of the counter scales linearly with the size of the input.