<https://leetcode.com/problems/count-number-of-nice-subarrays/description/>

Given an array of integers nums and an integer k. A continuous subarray is called nice if there are k odd numbers on it.

Return the number of nice sub-arrays.

**Example 1:**

**Input:** nums = [1,1,2,1,1], k = 3

**Output:** 2

**Explanation:** The only sub-arrays with 3 odd numbers are [1,1,2,1] and [1,2,1,1].

**Example 2:**

**Input:** nums = [2,4,6], k = 1

**Output:** 0

**Explanation:** There are no odd numbers in the array.

**Example 3:**

**Input:** nums = [2,2,2,1,2,2,1,2,2,2], k = 2

**Output:** 16

**Constraints:**

1 <= nums.length <= 50000

1 <= nums[i] <= 10^5

1 <= k <= nums.length

**Attempt 1: 2025-02-02**

**Solution 1: Not fixed Sliding Window (10min)**

class Solution {

    public int numberOfSubarrays(int[] nums, int k) {

        return helper(nums, k) - helper(nums, k - 1);

    }

    private int helper(int[] nums, int k) {

        int result = 0;

        int oddCount = 0;

        int i = 0;

        for(int j = 0; j < nums.length; j++) {

            if(nums[j] % 2 == 1) {

                oddCount++;

            }

            while(oddCount > k && i <= j) {

                if(nums[i] % 2 == 1) {

                    oddCount--;

                }

                i++;

            }

            result += (j - i + 1);

        }

        return result;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to Deepseek**

Yes, **LeetCode 1248: Count Number of Nice Subarrays** can be solved using the **Sliding Window** approach. This problem is similar to **LeetCode 930: Binary Subarrays With Sum**, but instead of counting subarrays with a specific sum, we count subarrays with exactly k odd numbers.

**Problem Recap:**

Given an array of integers nums and an integer k, return the number of "nice" subarrays. A subarray is called "nice" if it contains exactly k odd numbers.

**Example:**

Input: nums = [1, 1, 2, 1, 1], k = 3

Output: 2

Explanation: The nice subarrays are:

[1, 1, 2, 1] and [1, 2, 1, 1].

**Sliding Window Approach:**

The idea is to use a sliding window to maintain a window [i, j] that contains exactly k odd numbers. We can use the same pattern as in **LeetCode 930**, where we calculate the number of subarrays with **at most** k odd numbers and subtract the number of subarrays with **at most** k - 1 odd numbers. This gives us the number of subarrays with **exactly** k odd numbers.

**Algorithm Steps:**

Define a helper function atMostK(nums, k):

This function counts the number of subarrays with \*\*at most\*\* k odd numbers.

Use a sliding window to maintain a window [i, j] where the number of odd numbers is less than or equal to k.

When the number of odd numbers exceeds k, shrink the window from the left (i++).

Count the number of valid subarrays ending at j.

Use the helper function to compute:

atMostK(nums, k): Number of subarrays with \*\*at most\*\* k odd numbers.

atMostK(nums, k - 1): Number of subarrays with \*\*at most\*\* k - 1 odd numbers.

The result is atMostK(nums, k) - atMostK(nums, k - 1).

**Code Implementation:**

class Solution {

public int numberOfSubarrays(int[] nums, int k) {

return atMostK(nums, k) - atMostK(nums, k - 1);

}

private int atMostK(int[] nums, int k) {

int i = 0; // Left pointer of the window

int count = 0; // Count of valid subarrays

int oddCount = 0; // Count of odd numbers in the current window

for (int j = 0; j < nums.length; j++) {

// If the current number is odd, increment oddCount

if (nums[j] % 2 != 0) {

oddCount++;

}

// Shrink the window if oddCount exceeds k

while (oddCount > k && i <= j) {

if (nums[i] % 2 != 0) {

oddCount--;

}

i++;

}

// Count the number of valid subarrays ending at j

count += j - i + 1;

}

return count;

}

}

**Explanation:**

**Helper Function atMostK(nums, k)**:

This function counts the number of subarrays with \*\*at most\*\* k odd numbers.

It uses a sliding window [i, j] to maintain a window where the number of odd numbers is less than or equal to k.

When the number of odd numbers exceeds k, the window is shrunk from the left (i++).

The number of valid subarrays ending at j is j - i + 1.

**Final Result**:

The number of subarrays with \*\*exactly\*\* k odd numbers is given by:

atMostK(nums, k) - atMostK(nums, k - 1)

**Example Walkthrough**:

For nums = [1, 1, 2, 1, 1] and k = 3:

atMostK(nums, 3) counts all subarrays with at most 3 odd numbers.

atMostK(nums, 2) counts all subarrays with at most 2 odd numbers.

Subtracting the two gives the number of subarrays with exactly 3 odd numbers.

**Time Complexity:**

(O(n)): We traverse the array twice (once for atMostK(nums, k) and once for atMostK(nums, k - 1)), and each traversal takes (O(n)) time.

**Space Complexity:**

(O(1)): We use only a few variables for the sliding window, so the space complexity is constant.

This approach is efficient and works well for this problem. Let me know if you have further questions!

**Solution 2: Prefix Sum + Hash Table (10min)**

class Solution {

    public int numberOfSubarrays(int[] nums, int k) {

        int result = 0;

        Map<Integer, Integer> freq = new HashMap<>();

        freq.put(0, 1);

        int count = 0;

        for(int num : nums) {

            count += (num % 2 == 1 ? 1 : 0);

            result += freq.getOrDefault(count - k, 0);

            freq.put(count, freq.getOrDefault(count, 0) + 1);

        }

        return result;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/count-number-of-nice-subarrays/editorial/>

**Approach 1: Hashing**

**Intuition**

Since we only need to find the number of subarrays that contain a certain count of odd elements, we can ignore the numerical values of the elements and replace all odd values with 1 and even values with 0.

Now, all we need to do is identify sequences of elements within the array whose sum equals the number of odd elements needed to make a nice array. Solutions that require sequences of elements to meet criteria often utilize prefix sums, also sometimes referred to as cumulative sums.

**Note:** If you aren't aware of this concept we recommend you first solve this problem [560. Subarray Sum Equals K](https://leetcode.com/problems/subarray-sum-equals-k/).

Utilizing prefix sums simplifies our approach and lets us avoid determining the sum of elements for every new subarray considered. Using the prefix sums approach, we can calculate the sum of elements between two indices, subtracting the prefix sum corresponding to the two indices to obtain the sum directly instead of iterating over the subarray to find the sum.

We'll use this approach to calculate how many odd numbers are between two indices in the array. Let's call the two indices start and end. If the number of odd numbers between start and end equals k, we have found a nice subarray. We will calculate this by finding the difference between the end and start indices.

Based on these thoughts, we use a hashmap to store the prefix sum of indices as keys and their frequency of occurrence as values. Instead of modifying nums, we can apply the modulo 2 operation when storing values in the hashmap.

We traverse the array nums to compute the prefix sum up to each element modulo 2. Each unique sum encountered is recorded in a hashmap. If a sum repeats, we increment its corresponding count in the hashmap. Also, for each sum encountered, we find the number of times sum - k has appeared before, as this count indicates how many subarrays with sum k exist up to the current index. We increase the count by that same amount.

**Algorithm**

Initialize integers currSum = 0,subarrays = 0 and a hashmap prefixSum.

**Initialize prefixSum[0] with 1 to account for the initial value of currSum.**

Iterate over all the elements of nums:

Compute currSum as currSum = currSum + nums[i] % 2.

If currSum - k exists in the hashmap:

Increment the value of subarrays with prefixSum[currSum - k].

Increment prefixSum[currSum] by 1.

Return subarrays.

**Implementation**

class Solution {

public int numberOfSubarrays(int[] nums, int k) {

int currSum = 0, subarrays = 0;

Map<Integer, Integer> prefixSum = new HashMap<>();

prefixSum.put(currSum, 1);

for (int i = 0; i < nums.length; i++) {

currSum += nums[i] % 2;

// Find subarrays with sum k ending at i

if (prefixSum.containsKey(currSum - k)) {

subarrays = subarrays + prefixSum.get(currSum - k);

}

// Increment the current prefix sum in hashmap

prefixSum.put(currSum, prefixSum.getOrDefault(currSum, 0) + 1);

}

return subarrays;

}

}

**Complexity Analysis**

Let *n* be the number of elements in nums.

Time complexity: *O*(*n*) We iterate through the array exactly once. In each iteration, we perform insertion and search operations in the hashmap that take *O*(1) time. Therefore, the time complexity can be stated as *O*(*n*).

Space complexity: *O*(*n*) In each iteration, we insert a key-value pair in the hashmap. The space complexity is *O*(*n*) because the size of the hashmap is proportional to the size of the list after *n* iterations.

**Refer to**

[L930.Binary Subarrays With Sum (Ref.L992,L1248)](note://WEB0c236239487de108cb6c215aee18f315)

[L992.Subarrays with K Different Integers](note://WEBed9f16531d534d22184e6335b24e4cff)