

# 713. Subarray Product Less Than K

Medium   Array   Sliding Window

[Leetcode Link](#)

## Problem Description

Given an array of integers `nums` and an integer `k`, the task is to find out how many contiguous subarrays (a subarray is a sequence of adjacent elements in the array) have a product of all its elements that is less than `k`. The integers in the array can be any integer and the order of the array must not be changed. The output should be the count of such subarrays.

## Intuition

To arrive at the solution, we can think about a sliding window approach. We keep expanding our window by adding elements from the right and multiplying them until the product of all elements in the window is less than `k`. When the product becomes equal or greater than `k`, we'll shrink the window from the left until the product is less than `k` again.

As we slide the window to the right across the array, we can continuously make the window as large as possible such that its product is less than `k`. For each position `i`, if the product of the window `s` of elements to the left (up to `j`) is less than `k`, then adding the element at `i` can form `i - j + 1` new subarrays with products less than `k` (each subarray ending at `i` and starting at any of the elements between `i` and `j`, inclusive).

The intuition behind while calculating `i - j + 1`, is that for each new element added to the window (when `i` is increased), we add all the possible subarrays that end with that element. For example, if our window includes elements `nums[3]`, `nums[4]`, `nums[5]`, and we now include `nums[6]`, then we can form the subarrays `[nums[6]]`, `[nums[5], nums[6]]`, `[nums[4], nums[5], nums[6]]`, and `[nums[3], nums[4], nums[5], nums[6]]`.

By using this approach, we can avoid recalculating the product of subarrays that have already been considered, therefore optimizing the entire process.

## Solution Approach

The solution approach uses a sliding window technique to find contiguous subarrays that meet the product requirement. Here's how it breaks down:

- Initialization:** We start with initializing three variables - `ans`, `s`, and `j`.
  - `ans` will hold the final count of the number of subarrays whose product is less than `k` and is initialized to 0.
  - `s` is used to maintain the product of the elements within the current window and is initialized to 1.
  - `j` represents the start of the sliding window, and it's initialized to 0.
- Traversing the Array:** We traverse the array using variable `i`, which acts as the end of the sliding window. For each element, we do the following:
  - Multiply `s` by the current element `v` (the product of the window).
- Adjusting the Window:** If the product `s` becomes greater than or equal to `k`, we need to shrink our window from the left. We do this in a `while` loop, which continues until the product `s` is smaller than `k`:
  - Divide `s` by the value at the start of the window, `nums[j]`, to remove it from the product.
  - Move the start of the window to the right by incrementing `j`.
- Counting Subarrays:** After adjusting the window such that the product `s` is less than `k`, we count the number of new subarrays that can be formed with the current end-point `i`. This is done by calculating `i - j + 1`:
  - The reason behind `i - j + 1` is that each time the right end of the window moves (when `i` increases), all possible subarrays that end with the new element must be counted. These subarrays are all unique for this particular position of `i`.
- Accumulating the Answer:** The number calculated from the step above is added to `ans`, accumulating the count of all valid subarrays that meet the condition.
- Returning the Result:** Finally, after the loop has traversed the whole array, the accumulated value in `ans` represents the total number of contiguous subarrays where the product of all the elements is strictly less than `k`, and we return this value.

This approach ensures that each element is added and removed from the product at most once, leading to a time complexity of  $O(N)$ , where  $N$  is the number of elements in the array. The sliding window efficiently keeps track of the product of elements within it while keeping the product under `k`.

## Example Walkthrough

Let's walkthrough a small example to illustrate the solution approach.

Suppose we have `nums = [10, 5, 2, 6]` and `k = 100`.

- Initialization:** We start with `ans = 0`, `s = 1`, and `j = 0`.
- Traversing the Array:**
  - `i = 0`: We have the element `10` and `s = s * 10 = 10` which is less than `k`. No need to adjust the window.
    - Counting Subarrays:** `i - j + 1` gives us `1 - 0 + 1 = 2`, but the only valid subarray is `[10]`. So, `ans = ans + 1 = 1`.
  - `i = 1`: We have the element `5` and `s = s * 5 = 50` which is still less than `k`.
    - Counting Subarrays:** `i - j + 1` gives us `2 - 0 + 1 = 3`. The new valid subarrays are `[5]`, `[10, 5]`. So, `ans = ans + 2 = 3`.
  - `i = 2`: We have the element `2` and `s = s * 2 = 100` which is not less than `k`. Time to adjust the window.
    - Adjusting the Window:** Since `s >= k`, we divide `s` by `nums[j]`. So, `s = s / 10 = 10`, and `j = j + 1 = 1`.
    - We multiply `s` by `2` again as our window had shrunk. Now `s = 20`.
    - Counting Subarrays:** `i - j + 1` gives us `3 - 1 + 1 = 3`. The new valid subarrays are `[2]`, `[5, 2]`. So `ans = ans + 2 = 5`.
  - `i = 3`: We have the element `6` and `s = s * 6 = 120` which is not less than `k`.
    - Adjusting the Window:** We continue to shrink the window. `s = s / 5 = 24`, and `j = j + 1 = 2`.
    - Counting Subarrays:** `i - j + 1` gives us `4 - 2 + 1 = 3`. The new valid subarrays are `[6]`, `[2, 6]`. So, `ans = ans + 2 = 7`.
- Returning the Result:** After traversing the whole array, `ans = 7`, which is the count of contiguous subarrays with a product less than `100`.

So, by following the algorithm's steps, we can determine that there are seven contiguous subarrays within `nums` that have a product less than `k = 100`.

## Python Solution

```
1 from typing import List
2
3 class Solution:
4     def numSubarrayProductLessThanK(self, nums: List[int], k: int) -> int:
5         # Initialize count of subarrays, product of elements, and the start index
6         count_subarrays = 0
7         product = 1
8         start_index = 0
9
10        # Iterate over the list using index and value
11        for end_index, value in enumerate(nums):
12            product *= value # Update the product with the current value
13
14            # When the product is equal or greater than k, divide it by the
15            # starting value and increment the start_index to reduce the window size
16            while start_index <= end_index and product >= k:
17                product //= nums[start_index]
18                start_index += 1
19
20            # Calculate the new subarrays with the current element
21            # Here, (end_index - start_index + 1) gives the count of subarrays ending with nums[end_index]
22            count_subarrays += end_index - start_index + 1
23
24        # The final result is the count of subarrays satisfying the condition
25        return count_subarrays
26
```

## Java Solution

```
1 class Solution {
2     public int numSubarrayProductLessThanK(int[] nums, int k) {
3         int count = 0; // Initialize the count of subarrays
4         // Initialize 'start' and 'end' as the current window's start and end indices,
5         // and 'product' as the product of the elements within the window.
6         for (int start = 0, end = 0, product = 1; end < nums.length; end++) {
7             // Multiply the product by the current element.
8             product *= nums[end];
9             // While the product is not less than 'k' and 'start' is not beyond 'end'...
10            while (start <= end && product >= k) {
11                // ...divide the product by the 'start' element and move the start forward.
12                product /= nums[start++];
13            }
14            // Add to count the number of subarray combinations with the current 'end'.
15            // This is found by subtracting 'start' from 'end' and adding 1.
16            count += end - start + 1;
17        }
18        // Return the total count of subarrays where the product is less than 'k'.
19        return count;
20    }
21 }
22
```

## C++ Solution

```
1 class Solution {
2 public:
3     // Function to count the number of continuous subarrays
4     // where the product of all elements in the subarray is less than k
5     int numSubarrayProductLessThanK(vector<int>& nums, int k) {
6         // Initialize the answer to 0
7         int count = 0;
8
9         // Initialize two pointers defining the sliding window
10        // i is the right pointer, j is the left pointer,
11        // and product keeps the product of all elements within the window
12        for (int right = 0, left = 0, product = 1; right < nums.size(); ++right) {
13            // Update the product by multiplying the rightmost element
14            product *= nums[right];
15
16            // While the product is equal to or larger than k,
17            // increment the left pointer to reduce the product
18            // and contract the window size from the left
19            while (left <= right && product >= k) {
20                product /= nums[left++];
21            }
22
23            // The number of subarrays with a product less than k
24            // that end with nums[right] is equal to the size of the current window
25            count += right - left + 1;
26        }
27
28        // Return the total count
29        return count;
30    }
31 };
32
```

## Typescript Solution

```
1 function numSubarrayProductLessThanK(nums: number[], k: number): number {
2     let count = 0; // Initialize the count of subarrays
3     let product = 1; // Initialize the product of the current subarray
4     let left = 0; // Left index of the sliding window
5
6     // Process all numbers in the input array
7     for (let right = 0; right < nums.length; ++right) {
8         // Accumulate the product of the elements within the sliding window
9         product *= nums[right];
10
11        // If the product is greater than or equal to k,
12        // shrink the window from the left
13        while (left <= right && product >= k) {
14            product /= nums[left++];
15        }
16
17        // Add the number of subarrays ending at 'right' that have a product less than k
18        // This step utilizes the idea that if 'right' is the end of a valid subarray,
19        // then there are 'right - left + 1' possible starting positions for subarrays
20        count += right - left + 1;
21    }
22
23    // Return the total count of subarrays
24    return count;
25 }
26
```

## Time and Space Complexity

The provided code implements a two-pointer approach to find the number of contiguous subarrays where the product of all elements in the subarray is less than `k`.

### Time Complexity

The time complexity of the code is  $O(n)$ , where  $n$  is the number of elements in the input list `nums`. This is because there is a single loop over the elements, and within this loop, there is a while loop. However, the inner while loop does not result in a time complexity greater than  $O(n)$  since each element is visited at most twice by the pointers — once by the outer loop and at most once by the `j` pointer moving forward. Every element is processed by the inner loop only once when the product exceeds `k`, and after it's processed, the `j` pointer does not go back.

### Space Complexity

The space complexity of the code is  $O(1)$ . This is because the space used does not grow with the size of the input `nums`. Instead, only a fixed number of integers (`ans`, `s`, `j`, `i`, `v`) are used to keep the state of the algorithm, which occupies constant space.