<https://leetcode.com/problems/subarray-product-less-than-k/>

Given an array of integers nums and an integer k, return *the number of contiguous subarrays where the product of all the elements in the subarray is strictly less than* k.

**Example 1:**

Input: nums = [10,5,2,6], k = 100

Output: 8

Explanation: The 8 subarrays that have product less than 100 are:

[10], [5], [2], [6], [10, 5], [5, 2], [2, 6], [5, 2, 6]

Note that [10, 5, 2] is not included as the product of 100 is not strictly less than k.

**Example 2:**

Input: nums = [1,2,3], k = 0

Output: 0

**Constraints:**

1 <= nums.length <= 3 \* 10^4

1 <= nums[i] <= 1000

0 <= k <= 10^6

**Attempt 1: 2022-09-09 (30min, spend time to figure out condition avoid forever loop)**

class Solution {

// We can use Sliding Window to expand on right and shrink on left is because the

// given nums[i] is positive only as 1 <= nums[i] <= 1000, which means if multiple

// any new element will monopoly increasing, and divide any old element will monopoly

// decreasing, which match the usage scenario of Sliding Window.

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

        int product = 1;

        int count = 0;

        int len = nums.length;

        int i = 0;

        // e.g nums = {10,5,2,6}, k = 100

        // i = 0, j = 0 -> product = 10 < 100, count += (0 - 0 + 1) = 1

        // {10}

        // i = 0, j = 1 -> product = 10 \* 5 < 100, count += (1 - 0 + 1) = 3

        // {10} || new add {10,5}, {5}

        // i = 0, j = 2 -> product = 10 \* 5 \* 2 >= 100, product = 100 / 10 = 10, i = 1

        // count += (2 - 1 + 1) = 5

        // {10}, {10,5}, {5} || new add {2}, {5,2}

        // i = 1, j = 3 -> product = 5 \* 2 \* 6 < 100, count += (3 - 1 + 1) = 8

        // {10}, {10,5}, {5}, {2}, {5,2} || new add {5,2,6}, {2,6}, {6}

        // ------------------------------------------------------------

        // e.g nums = {1,2,3}, k = 0

        // i = 0, j = 0 -> product = 1 >= 0, product = 1 / 1 = 1, i = 1

        // count += (0 - 1 + 1) = 0

        // i = 1, j = 1 -> product = 2 >= 0, product = 2 / 2 = 1, i = 2

        // count += (1 - 2 + 1) = 0

        // i = 2, j = 2 -> product = 3 >= 0, product = 3 / 3 = 1, i = 3

        // count += (2 - 3 + 1) = 0

        for(int j = 0; j < len; j++) {

            product \*= nums[j];

            // Must include i <= j, test out by nums = {1,2,3}, k = 0

            // if no i <= j, for product /= nums[i], i will increase to 3

            // and index out of boundary as product >= k always satisfied

            // and loop forever result into i keep increasing

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

                product /= nums[i];

                i++;

            }

            // The distance between left and right end pointer equal to newly

            // add number of subarrays

            count += (j - i + 1);

        }

        return count;

    }

}

Space Complexity: O(1)

Time Complexity: O(n)

**We can use Sliding Window to expand on right and shrink on left is because the given nums[i] is positive only as 1 <= nums[i] <= 1000, which means if multiple any new element will monopoly increasing, and divide any old element will monopoly decreasing, which match the usage scenario of Sliding Window.**

**Refer to chatGPT**

**Key Idea: Sliding Window (Two Pointers)**

Use the **sliding window** technique to maintain a window of contiguous elements where the product is less than k.

Expand the window by moving the **right pointer**.

If the product exceeds or equals k, shrink the window by moving the **left pointer**.

For each valid window, all subarrays ending at the current right pointer are valid:

Count = right - left + 1

class Solution {

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

if (k <= 1) return 0; // If k <= 1, no product can be less than k

int count = 0;

int left = 0;

long product = 1;

// Sliding window

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

product \*= nums[right];

// Shrink the window if product >= k

while (product >= k) {

product /= nums[left];

left++;

}

// Count all valid subarrays ending at 'right'

count += (right - left + 1);

}

return count;

}

}

**Explanation:**

Initialization:

left pointer starts at the beginning of the array.

product keeps track of the product of elements in the current window.

Expand the Window:

Multiply the current nums[right] to the product.

Shrink the Window:

If product >= k, move the left pointer to the right (shrink the window) and divide out nums[left] from the product.

Count Valid Subarrays:

At each step, the number of valid subarrays ending at index right is:
right - left + 1

This is because all subarrays starting from left to right (inclusive) are valid.

Return Result:

Accumulate the count for all windows.

**Example:**

**Input:**

nums = [10, 5, 2, 6]

k = 100

**Execution:**

Initialize: left = 0, product = 1, count = 0

Step-by-Step:

right = 0, nums[0] = 10 → product = 10, count += 1 → count = 1

right = 1, nums[1] = 5 → product = 50, count += 2 → count = 3

right = 2, nums[2] = 2 → product = 100, shrink left → product = 10, count += 3 → count = 6

right = 3, nums[3] = 6 → product = 60, count += 4 → count = 10

**Output:**

10

**Edge Cases:**

k = 0 or 1: Return 0 since no subarray can have a product less than kkk.

Single Element: Handle edge cases where the product is already less than kkk.

**Refer to**

[L325.Maximum Size Subarray Sum Equals k (Ref.L560)](note://9A4A0D452DBF4E8AB5637DE9850FE4B6)

[L560.Subarray Sum Equals K](note://5CE29DE019904808986311AFAC74C85C)

[L2302.Count Subarrays With Score Less Than K](note://WEB7e8bf2699e4e216a31dae44a8cebc592)