2444. Count Subarrays With Fixed Bounds

Monotonic Queue

Sliding Window

Problem Description

Hard

Array

You are provided with an array of integers called nums, and two specific integers known as mink and maxk. The goal is to find and count the number of subarrays within nums that adhere to two criteria:

• The largest number in the subarray is exactly maxk.

The smallest number in the subarray is exactly mink.

To clarify, a subarray is defined as a contiguous segment of the array. It is essential to understand that any subarray meeting these conditions is considered a "fixed-bound subarray". The solution requires you to return the total count of such fixed-bound

- subarrays. Intuition

The problem is essentially asking us to isolate segments of the nums array where the range of values is strictly bounded between mink and maxk (both inclusive). To solve this, we need to keep track of certain positions within our array:

• The most recent positions where mink and maxk were found, since these will be the bounds of our subarrays. • The last position where a value outside the acceptable range (less than mink or greater than maxk) was encountered because this will help us determine the starting point of a new possible subarray.

- We iterate through the array while tracking these positions. For each new element:
- 1. If the current element is out of bounds (< mink or > maxk), we mark the current position as the starting point for future valid subarrays. 2. If the current element is equal to mink, we update the position tracking mink.

- The key insight here is that a new valid subarray can be formed at each step if the most recent mink and maxk are found after the
- last out-of-bounds value. The length of each new subarray added will be from the latest out-of-bound index until the minimum of the indices where mink and maxk were most recently found.

3. If the current element is equal to maxK, we update the position tracking maxK.

By performing these steps for each element, we are effectively counting all possible fixed-bound subarrays without having to explicitly list or generate them. This makes the solution efficient as it has a linear complexity with respect to the length of the input array.

Solution Approach The solution approach can be broken down into a few distinct steps that relate to the iteration over the nums array: **Initialization**: We start by initializing three pointers j1, j2, and k to -1.

• j2 will keep track of the most recent position where maxk has been found.

and v the value at each index.

ans as the final result.

• k signifies the most recent index before the current position where a number not within the mink and maxk range was discovered. Iteration and Tracking: The program iterates over the array nums using a for-loop, with the variable i representing the index,

If v is not within the range [minK, maxK] (v < minK or v > maxK), we update k to the current index i, since valid subarrays

o j1 will keep track of the most recent position where mink has been found.

- cannot extend beyond this point.
- If v equals mink, we update j1 to be the current index i. Similarly, if v equals maxk, we update j2. Key Algorithm: After updating the pointers, we calculate the additional number of valid subarrays that include the

element v at index i, by calculating max(0, min(j1, j2) - k). This effectively counts the number of new subarrays where

mink and maxk are the minimum and maximum values respectively, and which do not include any out-of-bound elements

before k. **Incremental Summation**: We accumulate this count in ans with ans $+= \max(0, \min(j1, j2) - k)$.

Result: After the loop concludes, ans holds the total number of fixed-bound subarrays that can be found in nums. We return

- solution efficiently leverages a sliding window technique to count subarrays without actually constructing them. The use of pointers (j1, j2, and k) to delineate bounds of subarrays is a common pattern in array processing problems and is a cornerstone
- **Example Walkthrough**

Let's illustrate the solution approach using a small example. Consider the following array nums, with mink set to 2 and maxk set to 5:

of this solution's efficiency since it means we only need to iterate through the array once, giving us a time complexity of O(n).

By employing pointers to keep track of recent occurrences of mink and maxk, and the cut-off point for valid subarrays (k), the

nums = [1, 2, 3, 4, 5, 1, 2, 5, 3]minK = 2maxK = 5Now let's walk through the procedure step-by-step: Initialization:

nums[i]

 \circ j1 = -1 (most recent mink position)

Action

out of range, update k to i

minK found, update j1 to i

maxK found, update j2 to i

out of range, update k to i

minK found, update j1 to i

within range, no pointer update

within range, no pointer update

Iteration and Tracking:

```
\circ j2 = -1 (most recent maxK position)
\circ k = -1 (last out-of-bound position)

 ans = 0 (accumulator for the count of valid subarrays)
```

-1

-1

-1

4

4

4

6

0

0

5

Valid Subarray length

max(0, min(-1, -1) - 0) = 0

max(0, min(1, 4) - 0) = 1

max(0, min(1, 4) - 5) = 0

max(0, min(6, 4) - 5) = 0

ans

0

0

2 3

4

5

2

2

0

3

4

5

6

```
5
                   maxK found, update j2 to i
                                                  6
                                                       7
                                                                max(0, min(6, 7) - 5) = 1
                                                                                          2
                                                            5
     8
                                                                max(0, min(6, 7) - 5) = 1
                   within range, no pointer update
                                                       7
                                                                                          3
         3
                                                  6
                                                            5
  Upon completion, ans = 3, which is the total count of fixed-bound subarrays within nums that have mink as their minimum and
  maxK as their maximum.
  The fixed-bound subarrays accounted for in this example are:
 • [2, 3, 4, 5] starting at index 1
 • [2, 5] starting at index 6
 • [2, 5, 3] starting at index 6
  Each time we encounter a valid subarray, we update our accumulator ans, which eventually gives us the count of all valid
  subarrays by the time we reach the end of the array.
Solution Implementation
```

def count_subarrays(self, nums: List[int], min_k: int, max_k: int) -> int:

Iterate through the list, checking each number against min_k and max_k

Invalidate the subarray if the value is out of the specified range

last min k = last max k = last out of range = -1

Initialize the counter for the valid subarrays

Update the last seen index for min_k, if found

from typing import List class Solution:

Initialize pointers for tracking the positions of min_k, max_k, and out-of-range elements

Add to the count the number of valid subarrays ending with the current element

valid_subarrays_count += max(0, min(last_min_k, last_max_k) - last_out_of_range)

which is determined by the smallest index among last_min_k and last_max_k after the

last_min_k = index # Update the last seen index for max_k, if found if value == max_k: $last_max_k = index$

valid_subarrays_count = 0

if value == min_k:

for index, value in enumerate(nums):

last out-of-range element

if value < min_k or value > max_k:

last_out_of_range = index

C++

public:

};

TypeScript

class Solution:

class Solution {

Python

```
# Return the total count of valid subarrays
       return valid_subarrays_count
Java
class Solution {
    public long countSubarrays(int[] nums, int minK, int maxK) {
        long totalCount = 0; // Variable to store the total count of subarrays
       int lastMinIndex = -1; // Index of the last occurrence of minK
       int lastMaxIndex = -1; // Index of the last occurrence of maxK
       int lastInvalidIndex = -1; // Index of the last element not in [minK, maxK]
       // Iterate over each element in the array
       for (int currentIndex = 0; currentIndex < nums.length; ++currentIndex) {</pre>
           // If the current element is outside of the [minK, maxK] range, update lastInvalidIndex
           if (nums[currentIndex] < minK || nums[currentIndex] > maxK) {
                lastInvalidIndex = currentIndex;
           // If the current element is equal to minK, update lastMinIndex
           if (nums[currentIndex] == minK) {
                lastMinIndex = currentIndex;
           // If the current element is equal to maxK, update lastMaxIndex
           if (nums[currentIndex] == maxK) {
                lastMaxIndex = currentIndex;
           // Calculate the number of valid subarrays ending at the current index
           // It is the distance between the last invalid index and the minimum of the last occurrences of minK and maxK
```

totalCount += Math.max(0, Math.min(lastMinIndex, lastMaxIndex) - lastInvalidIndex);

int lastIndexOutsideRange = -1; // Index of the last number that is outside the [minK, maxK] range

// Counts and returns the number of subarrays where the minimum value is at least minK and the maximum value is at most maxK.

```
if (nums[i] == minK) lastMinIndex = i;
   // If current element equals maxK, update the index of the last occurrence of maxK
   if (nums[i] == maxK) lastMaxIndex = i;
    // Count subarrays ending at index i which have minK and maxK within them
    answer += max(0, min(lastMinIndex, lastMaxIndex) - lastIndexOutsideRange);
return answer; // Return the total count of valid subarrays
```

function countSubarrays(nums: number[], minK: number, maxK: number): number {

let minIndex = -1; // Stores the latest index of the element equal to minK

let maxIndex = -1; // Stores the latest index of the element equal to maxK

let result = 0; // This will hold the final count of subarrays

nums.forEach((number, index) => {

if (number < minK || number > maxK) {

if (number === minK) {

if (number === maxK) {

valid_subarrays_count = 0

for index, value in enumerate(nums):

last out-of-range element

return valid_subarrays_count

Time and Space Complexity

Return the total count of valid subarrays

if value < min_k or value > max_k:

last_out_of_range = index

return totalCount; // Return the total count of valid subarrays

long long countSubarrays(vector<int>& nums, int minK, int maxK) {

// Iterate through the array to count valid subarrays

for (int i = 0; i < nums.size(); ++i) {</pre>

int lastMinIndex = -1; // Index of the last occurrence of minK

int lastMaxIndex = -1; // Index of the last occurrence of maxK

long long answer = 0; // Variable to store the final count of subarrays

if (nums[i] < minK || nums[i] > maxK) lastIndexOutsideRange = i;

// If current element is outside the [minK, maxK] range, update the index

// If current element equals minK, update the index of the last occurrence of minK

let invalidIndex = -1; // Stores the latest index of the element outside of the [minK, maxK] range

minIndex = index; // Update minIndex when we find an element equal to minK

maxIndex = index; // Update maxIndex when we find an element equal to maxK

invalidIndex = index; // Update invalidIndex for numbers outside the range

Initialize pointers for tracking the positions of min_k, max_k, and out-of-range elements

```
// Calculate the number of valid subarrays that end at the current index
          result += Math.max(Math.min(minIndex, maxIndex) - invalidIndex, 0);
      });
      return result; // Return the total count of valid subarrays
from typing import List
```

 $last_min_k = last_max_k = last_out_of_range = -1$

Initialize the counter for the valid subarrays

def count_subarrays(self, nums: List[int], min_k: int, max_k: int) -> int:

Iterate through the list, checking each number against min_k and max_k

Invalidate the subarray if the value is out of the specified range

```
if value == min_k:
    last_min_k = index
# Update the last seen index for max_k, if found
if value == max_k:
    last_max_k = index
```

Update the last seen index for min_k, if found

The provided code snippet is designed to count the number of subarrays within an array nums where the minimum element is mink and the maximum element is maxk. To analyze the computational complexity, we will examine the time taken by each component of the code and then aggregate these components to get the final complexity. **Time Complexity:**

Add to the count the number of valid subarrays ending with the current element

valid_subarrays_count += max(0, min(last_min_k, last_max_k) - last_out_of_range)

which is determined by the smallest index among last_min_k and last_max_k after the

nums.

Space Complexity:

nums list.

through the list of elements.

• Inside the for loop, the code performs constant-time checks and assignments (such as comparison, assignment, and max operations), and these do not depend on the size of the input. Because there are no nested loops or recursive calls, the time complexity is directly proportional to the number of elements in the

The time complexity of the code can be determined by analyzing the for loop since it is the only part of the code that iterates

• The for loop iterates through each element of nums exactly once, meaning the loop runs for n iterations, where n is the number of elements in

- Therefore, the **time complexity** of the code is O(n).
- For space complexity, we look at the extra space required by the algorithm, not including the input and output: • The code uses a fixed number of variables (j1, j2, k, ans, and i), and no additional data structures that grow with the input size are used. The space required for these variables is constant and does not scale with the size of the input list nums.

As a result, the space complexity of the code is 0(1), meaning it requires a constant amount of extra space.