3034. Number of Subarrays That Match a Pattern I

Rolling Hash

Hash Function

Problem Description

<u>Array</u>

String Matching

Medium

In this problem, you're provided with two arrays, nums and pattern. The nums array is a basic integer array that begins with an index of 0 and has a size "n". The pattern array is also 0-indexed and contains a series of m integers that can only be −1, 0, or 1.

Your goal is to determine how many subarrays of nums match the pattern array. A subarray from nums is considered a match if for each element pattern[k] within the pattern array the relationship between the subsequent elements in the nums subarray follows the rule defined by the value of pattern[k]: If pattern[k] equals 1, then the element next in line in the nums subarray must be

greater than its predecessor. If it's ∅, they must be equal. And if it's -1, the next element must be less than its predecessor. The size of the subarray you are looking for in the nums array is always m + 1, which corresponds to the pattern length plus one, because you're comparing adjacent elements to be matched against the elements of pattern. The final answer should be the

total number of such subarrays found in nums. Intuition

To approach this problem, intuitively, you can think of sliding a window of size m + 1 across the nums array and compare each

the differences between adjacent elements in the nums subarray line up with the specified pattern in pattern. This involves a

slice of the array to the pattern. For every position in nums where you place the starting point of your sliding window, you check if

direct translation of the pattern's -1, 0, and 1 into "less than", "equal to" and "greater than" respectively, which can be done using straightforward comparisons. One way to facilitate this process is to define a helper function that encapsulates the comparison logic: it takes two numbers (adjacent elements from the nums array) and returns 0 if they're equal, 1 if the first is less than the second, or -1 if the first is greater than the second. This aligns perfectly with the elements of the pattern.

Solution Approach The solution provided is a straightforward brute force enumeration that involves iterating through the nums array and comparing slices of it against the pattern. To implement this approach, the solution does not depend on any special data structures or

to check against the pattern.

For the enumeration, the outer loop runs through the indices of the nums array, starting from 0 and going up to len(nums) len(pattern) to ensure that at each step there are enough elements left in the array to form a complete subarray of length m + 1. Within the outer loop, a comprehension is used with the all function, which returns True only if all elements in the iterable are True. This comprehension iterates over the pattern using enumerate, which provides both the index k and the value p(pattern

advanced algorithms. It relies on the use of a nested loop to access subarrays and the built-in comparison operators <, ==, and >

element) together. At each iteration of this comprehension, the helper function f is used which takes two arguments from the nums subarray at the positions i + k and i + k + 1 and returns a comparison result (-1, 0, or 1). If this result matches the current element p of the

pattern, the subarray can still be considered a match for the pattern. If any comparison fails, that subarray does not match the

pattern, and all will return False. If all does return True, meaning the current subarray does match the pattern, the ans variable is incremented by one. After the outer loop finishes, ans, which contains the count of all matching subarrays, is returned as the final answer.

This solution's time complexity is O(n * m), where n is the length of the nums array and m is the length of the pattern, as it entails

checking each subarray of size m + 1 in a linear manner against the pattern. No extra space is used outside of the input and

variables for counting, so the space complexity is O(1). **Example Walkthrough** Let's consider the nums array as [5, 3, 4, 2, 1, 3] and the pattern array as [1, -1, 0]. We need to find subarrays of length m

should be lesser, and 0 means the next number should be equal. The length of the pattern array m is 3. Therefore, the window size for comparison is 3 + 1 = 4. We will iterate through the nums array, taking sequences of 4 consecutive elements and comparing the changes between each to the pattern.

+ 1 = 4 that match the pattern where 1 in the pattern dictates the next number should be greater, -1 means the next number

The first window is [5, 3, 4, 2]. Here:

◦ 5 to 3 is a decrease (pattern[0] should be 1, but it is −1)

3 to 4 is an increase (pattern[1] should be −1, but it is 1)

3 to 4 is an increase (matches pattern[0] which is 1)

o 4 to 2 is a decrease (matches pattern[1] which is −1)

This window also does not match the pattern.

2 to 1 is a decrease (does not match pattern[2] which is 0)

 4 to 2 is a decrease (pattern[2] is 0, but it should be equal) This window does not match the pattern. Slide the window one position to the right to [3, 4, 2, 1]:

 4 to 2 is a decrease (does not match pattern[0] which is 1) ○ 2 to 1 is a decrease (matches pattern[1] which is -1)

Slide the window over one more to [4, 2, 1, 3]:

 1 to 3 is an increase (matches pattern[2] which is 0, but it should be equal) This window does not match the pattern either. Lastly, slide to the final window [2, 1, 3, 6] (assuming the array had a 6 at the end): 2 to 1 is a decrease (does not match pattern[0] which is 1)

This window, like the others, does not match the pattern.

3 to 6 is an increase (matches pattern[2] which is 0, but it should be equal)

1 to 3 is an increase (matches pattern[1] which is −1)

approach, this will return an answer of 0.

Solution Implementation

from typing import List

class Solution:

Python

method that was described in the solution approach.

def countMatchingSubarrays(self, nums: List[int], pattern: List[int]) -> int:

Iterate over nums to check for each subarray starting at index 'i'.

match_count += 1 # If it matches, increment the count.

// Method to count the number of subarrays matching a given pattern.

int numLength = nums.length; // Length of the input array.

int patternLength = pattern.length; // Length of the pattern array.

// Iterating through the elements of the subarray and the pattern.

for (int k = 0; k < patternLength && matchesPattern == 1; ++k) {</pre>

if (f(nums[i + k], nums[i + k + 1]) != pattern[k]) {

public int countMatchingSubarrays(int[] nums, int[] pattern) {

for (int i = 0; i <= numLength - patternLength; ++i) {</pre>

// Loop through the pattern

if (isMatch) {

return count;

if (a === b) {

} else {

return 0;

return a < b ? 1 : -1;

count++;

for (int k = 0; $k < m && isMatch; ++k) {$

// Return the total count of matching subarrays

const compare = (a: number, b: number): number => {

if (compare(nums[i + k], nums[i + k + 1]) != pattern[k]) {

// If the subarray matches the pattern, increment the count

function countMatchingSubarrays(nums: number[], pattern: number[]): number {

// Store the length of the 'nums' array and the 'pattern' array.

// A comparator function to compare two numbers based on the problem's criteria.

Check if the pattern of comparisons matches for the subarray starting at 'i'.

// Iterate over the input array up to the point where comparison with the pattern makes sense.

// Compare the result of the function `f` with the current element of the pattern.

matchesPattern = 0; // If they do not match, set the flag to false (0).

int matchesPattern = 1; // Flag to check if the current subarray matches the pattern (1 for true).

Initialize the count of matching subarrays.

for i in range(len(nums) - len(pattern) + 1):

Return the total count of matching subarrays.

Helper function to compare two numbers and return # 0 if equal, 1 if first is less, and -1 if first is greater. def compare(a: int, b: int) -> int: **if** a == b: return 0 else: return 1 if a < b else -1

if all(compare(nums[i + k], nums[i + k + 1]) == pattern_val for k, pattern_val in enumerate(pattern)):

// Variable to hold the count of matching subarrays.

None of the sliding windows for our chosen nums and pattern result in a match. Therefore, according to the above solution

Through the iteration, we perform the comparisons and increase our answer count when a subarray matches the pattern;

however, in this example, since we didn't find any matching subarrays, the answer remains 0. This illustrates the brute force

```
return match_count
Java
```

int count = 0;

class Solution {

match_count = 0

```
// If the subarray matches the pattern, increment the count.
            count += matchesPattern;
       // Return the total count of matching subarrays.
        return count;
    // Helper function to compare two integers and return -1, 0, or 1 based on their relationship.
    private int f(int a, int b) {
       if (a == b) {
            return 0;
                          // If both integers are equal, return 0.
       } else if (a < b) {
                         // If the first integer is less than the second, return 1.
            return 1;
       } else {
                         // If the first integer is greater than the second, return -1.
            return -1;
C++
class Solution {
public:
    // Function to count the number of subarrays in 'nums' that match the comparison pattern described in 'pattern'
    int countMatchingSubarrays(vector<int>& nums, vector<int>& pattern) {
        int n = nums.size(); // Size of the input array
        int m = pattern.size(); // Size of the pattern array
       int count = 0; // Initialize the count of matching subarrays
       // Helper lambda function to compare two numbers and output -1, 0, or 1 based on their relation
       auto compare = [](int a, int b) {
            if (a == b) {
                return 0;
            } else if (a < b) {</pre>
                return 1;
            } else {
                return -1;
        };
       // Loop through the main array to check every subarray of size equal to the pattern size
        for (int i = 0; i \le n - m; ++i) { // Fix: Updated the loop condition with '=' to include last subarray
            bool isMatch = true; // Flag to check if the current subarray matches the pattern
```

// Compare adjacent elements in 'nums' using the 'compare' function, and check against the pattern

isMatch = false; // If any comparison does not match the pattern, set the flag to false

};

};

TypeScript

```
const numsLength = nums.length;
      const patternLength = pattern.length;
      // Initialize the count of matching subarrays to 0.
      let matchingSubarraysCount = 0;
      // Iterate over the 'nums' array, checking each possible subarray of length equal to 'pattern'.
      for (let i = 0; i <= numsLength - patternLength; ++i) {</pre>
          // Initialize a flag indicating if the current subarray matches the 'pattern'.
          let isMatching = true;
          // Compare each element of the subarray with the next one and check against the 'pattern'.
          for (let k = 0; k < patternLength && isMatching; ++k) {</pre>
              // If the comparison does not match the pattern, set the flag to false.
              if (compare(nums[i + k], nums[i + k + 1]) !== pattern[k]) {
                  isMatching = false;
          // If the flag is still true after the comparison, increment the count.
          matchingSubarraysCount += isMatching ? 1 : 0;
      // Return the total count of matching subarrays.
      return matchingSubarraysCount;
from typing import List
class Solution:
   def countMatchingSubarrays(self, nums: List[int], pattern: List[int]) -> int:
       # Helper function to compare two numbers and return
       # 0 if equal, 1 if first is less, and -1 if first is greater.
       def compare(a: int, b: int) -> int:
           if a == b:
               return 0
           else:
               return 1 if a < b else -1
       # Initialize the count of matching subarrays.
       match_count = 0
       # Iterate over nums to check for each subarray starting at index 'i'.
        for i in range(len(nums) - len(pattern) + 1):
           # Check if the pattern of comparisons matches for the subarray starting at 'i'.
            if all(compare(nums[i + k], nums[i + k + 1]) == pattern_val for k, pattern_val in enumerate(pattern)):
               match_count += 1 # If it matches, increment the count.
       # Return the total count of matching subarrays.
        return match_count
Time and Space Complexity
```

array. This is because for each starting index in nums, the code checks whether the subarray beginning there matches the pattern by comparing each subsequent element until the end of pattern. There are n - m + 1 possible starting points in nums for

a matching subarray, and for each starting point, it potentially checks melements (the length of pattern). This results in O((n - m + 1) * m), which simplifies to 0(n * m).

The time complexity of the provided code is 0(n * m) where n is the length of the nums array and m is the length of the pattern

The space complexity of the code is 0(1) since the algorithm only uses a few extra variables (ans, i, and k) and does not allocate any additional space that grows with the input size; the space used is constant regardless of the input size.