2760. Longest Even Odd Subarray With Threshold

Sliding Window Easy

The problem presents an integer array nums and an integer threshold. The task is to determine the length of the longest

Problem Description

conditions: 1. The first element of the subarray, nums[1], must be even (nums[1] % 2 == 0).

subarray starting at index 1 and ending at index r, where $0 \ll 1 \ll r \ll nums$. Length. The subarray should satisfy these

- 2. Adjacent elements in the subarray must have different parity that is, one is even and the other is odd. For any two consecutive elements nums[i] and nums[i + 1] within the subarray, their mod 2 results must be different (nums[i] % 2 != nums[i + 1] % 2).
- 3. Every element within the subarray must be less than or equal to the <a hreshold (nums[i] <= threshold). The goal is to return the length of such the longest subarray meeting these criteria.

To find the longest subarray that satisfies the problem conditions, we can iterate through the array. For each potential starting

Intuition

extend the subarray by moving the right index r as far as possible while maintaining the alternating even-odd sequence and ensuring all elements are within the threshold. The process is as follows:

index 1 of a subarray, we check whether the starting element is even and less than or equal to the threshold. If it is, we try to

subarray must be non-empty.

We track the maximum length found throughout this process with the variable ans.

Here's a step-by-step breakdown of the algorithm based on the given solution approach:

while loop breaks, and the current window cannot be expanded further.

1. We iterate over each index 1 in the array nums as a potential start of the subarray.

3. We now extend the subarray by incrementing r as long as nums[r] alternates in parity with nums[r-1] and nums[r] is less than or equal to the threshold. 4. Once we can no longer extend r because the next element violates one of our conditions, we calculate the current subarray length as r - 1.

5. Continue the same process for each index 1 in the array and return ans, which will hold the maximum length of the longest subarray that meets

2. If nums [1] is even and less than or equal to the threshold, we have a potential subarray starting at 1. We initialize r as 1 + 1 because a

- all the criteria. By following this approach, we can ensure that we check all possible subarrays that start with an even number and have
- alternating parities until we either reach the end of the array or encounter elements that do not fulfill the conditions.
- The implementation uses a straightforward approach, which essentially follows the sliding window pattern. Sliding window is a common pattern used in array/string problems where a subarray or substring is processed, and then the window either expands or contracts to satisfy certain conditions.

Initialize a variable ans to store the maximum length found for any subarray that satisfies the problem conditions. Also,

Inside this loop, first check whether the element at index 1 is both even (nums[1] % 2 == 0) and less than or equal to the

Use a while loop to try expanding this window. The loop will continue as long as r is less than n, the array length, ensuring

In each iteration of the while loop, check two conditions: whether the parity at nums[r] is different from nums[r - 1]

As long as these conditions are satisfied, increment r to include the next element in the subarray. If either condition fails, the

determine the length n of the input array nums.

slide.

the final answer.

Solution Approach

given threshold. Only if these conditions are met, the element at 1 can be the starting element of a subarray. If the starting element is suitable, initialize the end pointer r for the subarray to l + 1. This is where the window starts to

Start the first for-loop to iterate over the array using the index 1, which represents the potential start of the subarray.

array bounds are not exceeded.

- (nums[r] % 2 != nums[r 1] % 2) and if nums[r] is less than or equal to threshold. This ensures the subarray keeps alternating between even and odd numbers with values within the threshold.
- Once the while loop ends, calculate the length of the current subarray by r 1 and update ans with the maximum of its current value and this newly found length.
- After processing the potential starting index 1, the for loop moves to the next possible start index and repeats the steps until the whole array is scanned. At the end of the for loop, ans contains the length of the longest subarray satisfying the given conditions, and it's returned as
- **Example Walkthrough**

each element in nums as a starting point, we might need to check up to n elements to find the end of the subarray.

 \circ r = 1: nums[1] = 3 which is different in parity from nums[0] and is also below the threshold. Increment r to 2.

 \circ r = 2: nums [2] = 4, which fails the alternating parity condition. Break the while loop.

6. Calculate the current subarray length: r - l = 2 - 0 = 2. Update ans to max(ans, 2) = 2.

nums [1] = 3 is odd, so we skip this as it can't be the start of a subarray.

9. Move on to 1 = 3. Again, nums[3] = 6 is above the threshold, so we skip this index.

def longest_alternating_subarray(self, nums: List[int], threshold: int) -> int:

:return: The length of the longest alternating subarray.

Get the number of elements in the nums list.

Initialize the maximum length of the alternating subarray.

if nums[left] % 2 == 0 and nums[left] <= threshold:</pre>

nums[right] <= threshold):</pre>

max_length = max(max_length, right - left)

// given that each element should not exceed a certain threshold.

public int longestAlternatingSubarray(int[] nums, int threshold) {

int maxLen = 0; // Initialize maximum length of alternating subarray

Return the maximum length of the subarray found.

The solution does not use any additional data structures, and its space complexity is O(1), which represents constant space aside

from the input array. The time complexity is O(n^2) in the worst case, where n is the number of elements in nums, because for

1. Initialize ans to 0. The length of the array n is 5. 2. Start iterating over the array with index 1 ranging from 0 to n-1. 3. At 1 = 0, we find nums[0] = 2, which is even and less than the threshold, so this can be the start of a subarray. 4. Initiate r to l+1, which is 1.

8. Increment 1 to 2. No suitable subarray starting from here since nums [2] = 4 does not meet the parity alternation condition with any element afterward, and it's above the threshold.

7. Increment 1 to 1 and repeat the steps:

example. So the function returns 2.

Solution Implementation

from typing import List

max length = 0

num_elements = len(nums)

for left in range(num elements):

right += 1

Python

class Solution:

10. Lastly, l = 4. Since nums[4] = 7 is not even, we skip this index. After iterating through all elements, the maximum subarray length that satisfies the conditions is stored in ans, which is 2 in this

Let's walk through an example to illustrate the solution approach.

Consider an array nums = [2, 3, 4, 6, 7] and a threshold = 5.

Following the steps outlined in the solution approach:

5. We enter the while loop to expand the window starting at 1 = 0:

This illustrates the entire process of checking each potential starting index and trying to expand the window to find the longest subarray that satisfies the given criteria.

Finds the length of the longest subarray where adjacent elements have different parity

Iterate through the list to find all starting points of potential alternating subarrays.

Check if the current element satisfies the parity and threshold condition.

Update the maximum length if a longer subarray is found.

and each element does not exceed the given threshold. :param nums: List of integers to find the longest alternating subarray from. :param threshold: An integer representing the maximum allowed value for array elements.

Initialize the right pointer which will try to extend the subarray to the right. right = left + 1# Extend the subarray while the elements alternate in parity and are within the threshold. while (right < num elements and</pre> nums[right] % 2 != nums[right - 1] % 2 and

```
Java
class Solution {
    // Method to find the length of the longest alternating subarray
```

return max_length

```
int n = nums.length; // Store the length of the input array
       // Loop through each element in the array as the starting point
        for (int left = 0; left < n; ++left) {</pre>
            // Check if current starting element is even and lower than or equal to threshold
            if (nums[left] % 2 == 0 && nums[left] <= threshold) {</pre>
                int right = left + 1; // Initialize the pointer for the end of subarray
                // Extend the subarray towards the right as long as:
                // 1. The current element alternates in parity with previous (even/odd)
                // 2. The current element is below or equal to the threshold
                while (right < n && nums[right] % 2 != nums[right - 1] % 2 && nums[right] <= threshold) {</pre>
                    ++right; // Move to the next element
                // Update the maximum length if a longer alternating subarray is found
                maxLen = Math.max(maxLen, right - left);
       // Return the length of the longest alternating subarray found
       return maxLen;
#include <vector>
#include <algorithm> // For using the max() function
class Solution {
public:
   // Function to find the length of the longest subarray with alternating even and odd numbers
   // not exceeding a given threshold
   int longestAlternatingSubarray(vector<int>& nums, int threshold) {
        int longestLength = 0; // This will hold the maximum length found
        int size = nums.size(); // Store the size of nums to avoid multiple size() calls
       // Iterate over each element in the nums array
        for (int left = 0; left < size; ++left) {</pre>
            // Check if the current element is even and within the threshold
            if (nums[left] % 2 == 0 && nums[left] <= threshold) {</pre>
```

int right = left + 1; // Start a pointer to expand the subarray to the right

// the subarray satisfies the conditions (alternating and within the threshold)

longestLength = max(longestLength, right - left); // Update the maximum length found

while (right < size && nums[right] % 2 != nums[right - 1] % 2 && nums[right] <= threshold) {</pre>

// Continue while right pointer is within array bounds and

// Return the length of the longest subarray found

// Iterate over the array starting from each index 'left'

if (nums[left] % 2 === 0 && nums[left] <= threshold) {</pre>

++right; // Move to the next element

let right = left + 1; // Start from the next element

// 'n' is the length of the input array 'nums'

for (let left = 0; left < n; ++left) {</pre>

function longestAlternatingSubarray(nums: number[], threshold: number): number {

let maxLength = 0; // Stores the maximum length of the alternating subarray

// and ensuring the values are below or equal to the threshold

Update the maximum length if a longer subarray is found.

max_length = max(max_length, right - left)

Return the maximum length of the subarray found.

// Check if the current element is even and lesser than or equal to the threshold

// Proceed to the right in the array while alternating between odd and even

while (right < n && nums[right] % 2 !== nums[right - 1] % 2 && nums[right] <= threshold) {</pre>

return longestLength;

const n = nums.length;

++right; // Move the right pointer to the next element

};

TypeScript

```
// Calculate the length of the current alternating subarray and update the max length
            maxLength = Math.max(maxLength, right - left);
    // Return the length of the longest alternating subarray found
    return maxLength;
from typing import List
class Solution:
    def longest_alternating_subarray(self, nums: List[int], threshold: int) -> int:
       Finds the length of the longest subarray where adiacent elements have different parity
        and each element does not exceed the given threshold.
        :param nums: List of integers to find the longest alternating subarray from.
        :param threshold: An integer representing the maximum allowed value for array elements.
        :return: The length of the longest alternating subarray.
       # Initialize the maximum length of the alternating subarray.
       max length = 0
       # Get the number of elements in the nums list.
       num_elements = len(nums)
       # Iterate through the list to find all starting points of potential alternating subarrays.
        for left in range(num elements):
           # Check if the current element satisfies the parity and threshold condition.
           if nums[left] % 2 == 0 and nums[left] <= threshold:</pre>
                # Initialize the right pointer which will try to extend the subarray to the right.
                right = left + 1
                # Extend the subarray while the elements alternate in parity and are within the threshold.
                while (right < num elements and</pre>
                       nums[right] % 2 != nums[right - 1] % 2 and
                       nums[right] <= threshold):</pre>
                    right += 1
```

return max_length Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed by considering that there are two nested loops: an outer loop that runs from 1 from 0 to n - 1, and an inner while loop that potentially runs from 1 + 1 to n in the worst case when all the elements conform to the specified condition (alternating even and odd values under the threshold).

In the worst-case scenario, the inner loop can iterate n times for the first run, then n-1 times, n-2 times, and so forth until 1

time. This forms an arithmetic progression that sums up to (n * (n + 1)) / 2 iterations in total. Hence, the worst-case time complexity of the function is $0(n^2)$.

Space Complexity

The space complexity of the given code is constant, 0(1), as it only uses a fixed number of variables (ans, n, 1, r) and does not allocate any additional space that grows with the input size.