2419. Longest Subarray With Maximum Bitwise AND

<u>Array</u>

```
Problem Description
```

**Bit Manipulation** 

Medium

In this LeetCode problem, you are presented with an integer array named nums, which contains a certain number of integers. Your objective is to find the maximum possible bitwise AND value from all possible non-empty subarrays of this array. A bitwise AND operates on two numbers bit by bit and only returns 1 for each bit position if both corresponding bits in the two numbers are also 1; otherwise, it returns 0 for that position.

Once you've determined this maximum bitwise AND value (let's call it k), you need to consider only the subarrays that produce

this maximum value when performing a bitwise AND on all their elements. Among these subarrays, your goal is to find the length of the longest one.

To sum up, you must:

2. Identify which subarrays yield this maximum value.

3. Determine the maximum length among those subar

3. Determine the maximum length among those subarrays.

1. Calculate the maximum bitwise AND value for all possible subarrays.

Brainteaser

A subarray is defined as a contiguous sequence of elements within the original array. So, the challenge essentially revolves around understanding bit manipulation and being able to efficiently navigate through the array to find the longest contiguous

sequence yielding the maximum bitwise AND value.

Intuition

The intuition behind the provided solution lies in understanding the properties of the bitwise AND operation. One of the key

insights is that if you perform a bitwise AND on any number with another number that is smaller, the result will always be less

## than or equal to the larger number. It means that the maximum bitwise AND value for any subarray in nums can only be obtained if every element in that subarray is at least as large as the maximum value in the entire array.

With this in mind, finding the solution does not require checking every possible subarray. Instead, you can follow a more straightforward approach by first finding the maximum value (mx) in nums. Then, you simply need to look for the longest contiguous sequence of mx within the array. This sequence will guarantee the maximum bitwise AND result because the AND of any number with itself is the number.

Here's how the solution approach is derived:
 Find the maximum value mx in nums.
 Initialize a counter (cnt) to track the length of the current sequence of mx elements, and another variable (ans) to keep track of the length of the longest sequence found so far.
 Iterate through the elements in nums, incrementing cnt if the current element is equal to mx. If an element is not equal to mx, reset cnt to 0

4. After each step, update ans with the greater of its current value or cnt, to ensure ans always represents the length of the longest sequence encountered.5. At the end of the iteration, ans holds the length of the longest subarray that gives the maximum bitwise AND value which is the solution to the

because the sequence is broken.

problem.

**Solution Approach** 

mx = max(nums)

ans = cnt = 0

**if** v == mx:

else:

return ans

cnt += 1

longest subarray found so far.

counting anew for the next potential sequence.

ans = max(ans, cnt)

The implementation of the solution uses a straightforward approach without the need for complex algorithms or additional data structures.

whose bitwise AND is to be maximized must contain mx according to the properties of the bitwise AND operation.

First, the maximum value (mx) in the array nums is identified using the max() function. This step is crucial because any subarray

2. Two variables are initialized: ans for storing the maximum subarray length found so far, and cnt for counting the length of the current sequence of maximum elements when iterating through nums.

Here is the step-by-step execution of the algorithm according to the provided Python code:

The function then iterates through every element (v) in nums. For each element, it checks if it equals the maximum value mx.
 for v in nums:
 If the current element is equal to mx, then cnt is incremented as we are currently in a subarray consisting of the maximum

element. The ans variable is updated with the larger of its current value or cnt to ensure that it always holds the length of the

cnt = 0

6. At the end of the iteration, the value of ans will be the length of the longest subarray where the bitwise AND is equal to the maximum possible value k. This value of ans is returned as the answer.

The code does not make use of any specific patterns or advanced data structures, relying instead on a simple linear scan of the

If the current element is not equal to mx, the cnt is reset to 0 because the sequence of mx is broken, and we need to start

```
input array and basic variables for counting. This type of pattern could be considered a two-pointer approach, where one pointer (or in this case a counter) keeps track of the current subarray's length and another (implicit pointer) moves through the array elements via the for-loop. The solution efficiently arrives at the answer in O(n) time, where n is the length of the nums array, since
```

**Example Walkthrough** 

mx = max(nums) # mx = 2

it requires only a single pass through the array.

nums = [2, 2, 1, 2, 2]

Using the solution approach, perform the following steps:

1. Identify the maximum value (mx) in nums:

Initialize the required variables to store the length of the current sequence (cnt) and the maximum subarray length found

Iterate through each element (v) in nums and compare it with mx, incrementing cnt if it's the same or resetting cnt if it's

## for v in nums: # Loop starts, iterating over the elements in nums.

else:

Solution Implementation

from typing import List

max\_value = max(nums)

for number in nums:

else:

class Solution {

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

longest\_length = current\_length = 0

current\_length += 1

current\_length = 0

public int longestSubarray(int[] nums) {

maxNum = Math.max(maxNum, num);

for (int num : nums) {

for (int num : nums) {

currentLength = 0;

// Return the length of the longest subarray

if number == max\_value:

# Find the maximum value in the array nums.

# Iterate through each element in the list.

# Initialize the longest length and the current length counter to zero.

# Update the longest length if the current length is greater.

# If the current element is equal to the maximum value...

longest\_length = max(longest\_length, current\_length)

int maxNum = 0; // variable to store the maximum value in the array

int maxLength = 0; // variable to store the length of the longest subarray

// Iterate through the array to find the length of the longest subarray

int currentLength = 0; // variable to track the length of the current subarray

// Reset the current length if the current element is not max

// Iterate through the array to find the maximum value

// where all elements are equal to the maximum value

# Increment the current length counter.

class Solution:

**Python** 

For the first element, v = 2:

if v == mx: # True, as 2 == 2

For the second element, v = 2:

if v == mx: # True, as 2 == 2

For the third element, v = 1:

cnt += 1 # cnt becomes 1

cnt += 1 # cnt becomes 2

different:

(ans):

ans = cnt = 0

ans = max(ans, cnt) # ans becomes max(0, 1) which is 1

ans = max(ans, cnt) # ans becomes max(1, 2) which is 2

cnt = 0 # cnt is reset since 1 is not equal to mx (2)

Let's consider the following array nums as an example to illustrate the solution approach:

```
    For the fourth element, v = 2:
    if v == mx: # True, as 2 == 2
        cnt += 1 # cnt becomes 1 again
        ans = max(ans, cnt) # ans remains 2, as max(2, 1) is 2
    For the fifth element, v = 2:
    if v == mx: # True, as 2 == 2
        cnt += 1 # cnt becomes 2 (as we had 1 from the previous step)
        ans = max(ans, cnt) # ans becomes max(2, 2) which is 2
    After completing the iteration, the ans variable contains the length of the longest subarray where the bitwise AND is equal to the maximum possible value k (which is 2 in this case), and ans is 2:
    return ans # Returns 2 as the answer.
    In this example, the longest continuous subarray with elements that have the maximum value mx (2) consists of 2 elements, so ans is 2. This is the length of the longest subarray that when bitwise AND-ed together would give the maximum possible value.
    The algorithm successfully finds this subarray length in one pass, which makes it a very efficient solution.
```

# After iterating through the list, return the longest length of the subarray with max values.
return longest\_length

Java

# If the current element is not equal to the max value, reset current length to 0.

```
if (num == maxNum) {
    // If the current element is the max, increment the current length
    currentLength++;
    // Update the maxLength if the current subarray is longer
    maxLength = Math.max(maxLength, currentLength);
```

} else {

return maxLength;

C++

```
#include <vector>
#include <algorithm>
class Solution {
public:
   // Method to find the length of the longest subarray consisting of the maximum element.
    int longestSubarray(std::vector<int>& nums) {
       // Get the maximum value in the array.
       int maxValue = *std::max_element(nums.begin(), nums.end());
       // Initialize answer (longest subarray length) and counter for current subarray length.
        int longestSubarrayLength = 0, currentSubarrayLength = 0;
       // Iterate over each element in the array.
        for (int value : nums) {
           // Check if the current element equals the maximum value.
            if (value == maxValue) {
                // Increment the current subarray length as it is part of a subarray containing max elements.
                ++currentSubarrayLength;
                // Update the answer with the maximum subarray length found so far.
                longestSubarrayLength = std::max(longestSubarrayLength, currentSubarrayLength);
            } else {
                // Reset current subarray length if the current element is not the maximum value.
                currentSubarrayLength = 0;
       // Return the length of the longest subarray found.
       return longestSubarrayLength;
TypeScript
// Importing the `max` function from Lodash for finding maximum element in array
import max from 'lodash/max';
// Function to find the length of the longest subarray consisting of the maximum element
function longestSubarray(nums: number[]): number {
```

// Get the maximum value in the array using Lodash max function

const maxValue: number = max(nums);

nums.forEach((value: number) => {

class Solution:

let longestSubarrayLength: number = 0;

let currentSubarrayLength: number = 0;

// Iterate over each element in the array

```
// If the current element equals the maximum value, it's part of a max-element subarray
          if (value === maxValue) {
              // Increment the current subarray length counter
              currentSubarrayLength++;
              // Update the longest subarray length if the current one is longer
              longestSubarrayLength = Math.max(longestSubarrayLength, currentSubarrayLength);
          } else {
              // If the current element is not the max value, reset current subarray length counter
              currentSubarrayLength = 0;
      });
      // Return the length of the longest subarray found
      return longestSubarrayLength;
  Please note that the method names haven't been changed as per the instruction. The given logic and functionality are equivalent 1
  Due to TypeScript's reliance on npm packages, you'd need to install lodash to use it:
  ```sh
  npm install lodash
from typing import List
```

// Initialize longestSubarrayLength for keeping track of the longest subarray length and currentSubarrayLength for the curren

```
Time and Space Complexity
```

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

longest\_length = current\_length = 0

if number == max\_value:

current\_length += 1

current\_length = 0

max\_value = max(nums)

for number in nums:

return longest\_length

else:

# Find the maximum value in the array nums.

# Iterate through each element in the list.

# Initialize the longest length and the current length counter to zero.

# Update the longest length if the current length is greater.

# If the current element is not equal to the max value, reset current length to 0.

# After iterating through the list, return the longest length of the subarray with max values.

# If the current element is equal to the maximum value...

longest\_length = max(longest\_length, current\_length)

# Increment the current length counter.

iterates through the list once with a single for-loop, performing constant-time operations within the loop.

The space complexity of the code is 0(1) as it uses a fixed amount of additional space (variables mx, ans, cnt, and v) that does not depend on the input size n.

The time complexity of the given code snippet is O(n) where n is the length of the input list nums. This is because the code

```
not depend on the input size n.
```