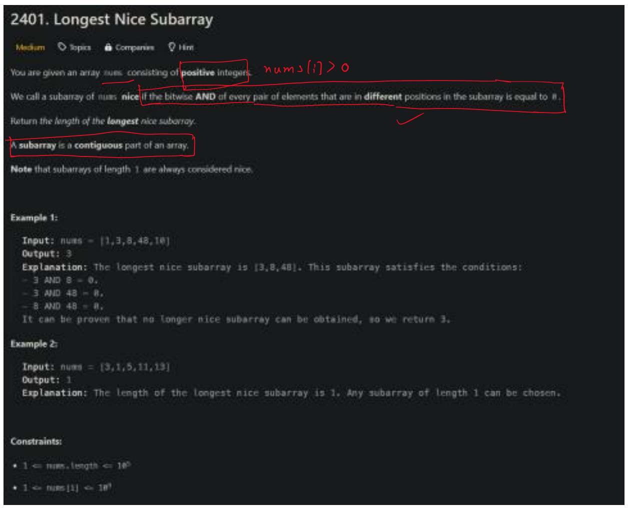
Longest Nice Subarray



2401. Longest Nice Subarray

Medium Topics Companies Hint

You are given an array `nums` consisting of **positive** integers. nums[i] > 0

We call a subarray of `nums` **nice** if the bitwise **AND** of every pair of elements that are in **different** positions in the subarray is equal to `0`.

*Return the length of the **longest** nice subarray.*

A **subarray** is a **contiguous** part of an array.

Note that subarrays of length `1` are always considered nice.

Example 1:

```
Input: nums = [1,3,8,48,10]
Output: 3
Explanation: The longest nice subarray is [3,8,48]. This subarray satisfies the conditions:
- 3 AND 8 = 0.
- 3 AND 48 = 0.
- 8 AND 48 = 0.
It can be proven that no longer nice subarray can be obtained, so we return 3.
```

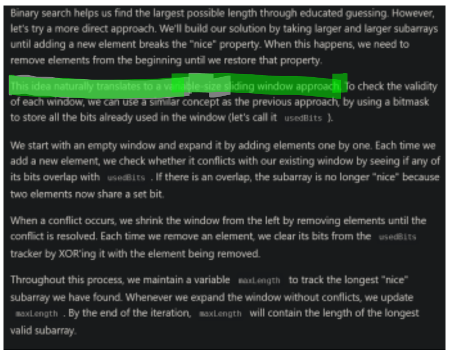
Example 2:

```
Input: nums = [3,1,5,11,13]
Output: 1
Explanation: The length of the longest nice subarray is 1. Any subarray of length 1 can be chosen.
```

Constraints:

- `1 <= nums.length <= $10^5$`
- `1 <= nums[i] <= $10^9$`

Approach 1: Sliding Window :- (Variable - sized)



Binary search helps us find the largest possible length through educated guessing. However, let's try a more direct approach. We'll build our solution by taking larger and larger subarrays until adding a new element breaks the "nice" property. When this happens, we need to remove elements from the beginning until we restore that property.

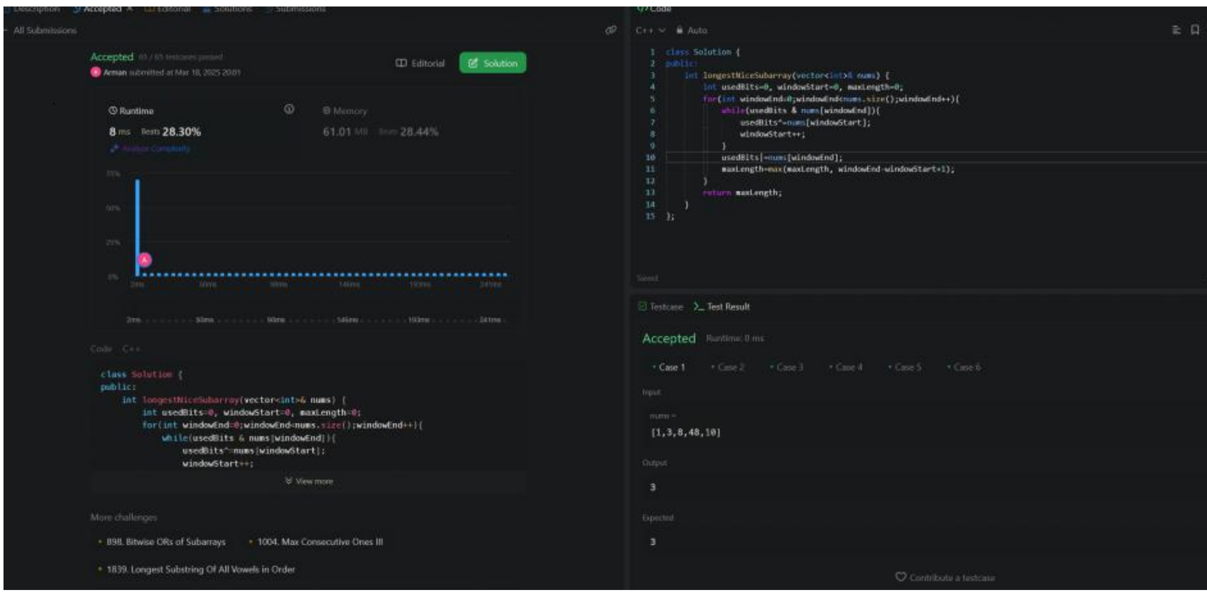
This idea naturally translates to a variable-size sliding window approach. To check the validity of each window, we can use a similar concept as the previous approach, by using a bitmask to store all the bits already used in the window (let's call it `usedBits`).

We start with an empty window and expand it by adding elements one by one. Each time we add a new element, we check whether it conflicts with our existing window by seeing if any of its bits overlap with `usedBits`. If there is an overlap, the subarray is no longer "nice" because two elements now share a set bit.

When a conflict occurs, we shrink the window from the left by removing elements until the conflict is resolved. Each time we remove an element, we clear its bits from the `usedBits` tracker by XOR'ing it with the element being removed.

Throughout this process, we maintain a variable `maxLength` to track the longest "nice" subarray we have found. Whenever we expand the window without conflicts, we update `maxLength`. By the end of the iteration, `maxLength` will contain the length of the longest valid subarray.

Solution Code :-



Accepted 65 / 65 testcases passed
Arman submitted at Mar 18, 2025 20:01

Runtime 8 ms Beats 28.30%
Memory 61.01 MB Beats 28.44%

```cpp
class Solution {
public:
    int longestNiceSubarray(vector<int>& nums) {
        int usedBits=0, windowStart=0, maxLength=0;
        for(int windowEnd=0;windowEnd<nums.size();windowEnd++){
            while(usedBits & nums[windowEnd]){
                usedBits^=nums[windowStart];
                windowStart++;
            }
            usedBits|=nums[windowEnd];
            maxLength=max(maxLength, windowEnd-windowStart+1);
        }
        return maxLength;
    }
};
```

More challenges
- 898. Bitwise ORs of Subarrays
- 1004. Max Consecutive Ones III
- 1839. Longest Substring Of All Vowels in Order

Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3 Case 4 Case 5 Case 6

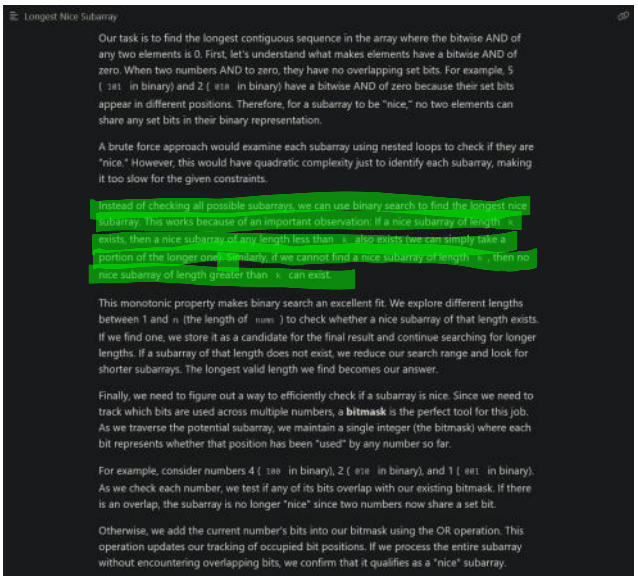
Input

nums =
```
[1,3,8,48,10]
```

Output
```
3
```

Expected
```
3
```

Approach 2: Binary Search :-



Longest Nice Subarray

Our task is to find the longest contiguous sequence in the array where the bitwise AND of any two elements is 0. First, let's understand what makes elements have a bitwise AND of zero. When two numbers AND to zero, they have no overlapping set bits. For example, 5 (`101` in binary) and 2 (`010` in binary) have a bitwise AND of zero because their set bits appear in different positions. Therefore, for a subarray to be "nice," no two elements can share any set bits in their binary representation.

A brute force approach would examine each subarray using nested loops to check if they are "nice." However, this would have quadratic complexity just to identify each subarray, making it too slow for the given constraints.

Instead of checking all possible subarrays, we can use binary search to find the longest nice subarray. This works because of an important observation: If a nice subarray of length `k` exists, then a nice subarray of any length less than `k` also exists (we can simply take a portion of the longer one). Similarly, if we cannot find a nice subarray of length `k`, then no nice subarray of length greater than `k` can exist.

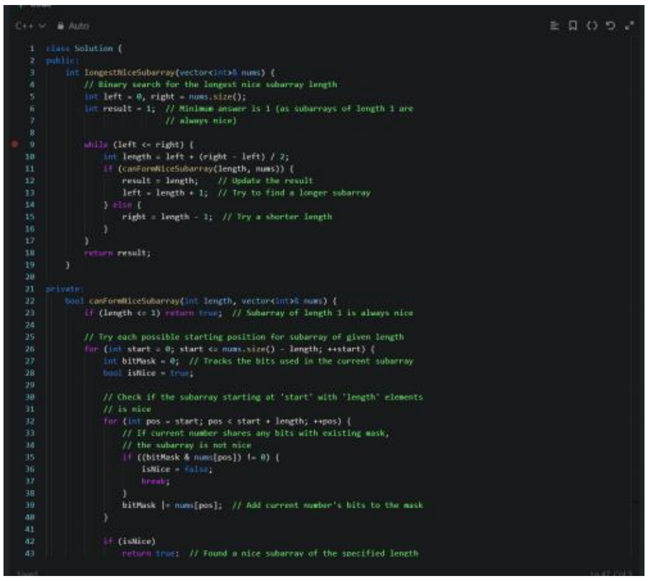
This monotonic property makes binary search an excellent fit. We explore different lengths between 1 and `n` (the length of `nums`) to check whether a nice subarray of that length exists. If we find one, we store it as a candidate for the final result and continue searching for longer lengths. If a subarray of that length does not exist, we reduce our search range and look for shorter subarrays. The longest valid length we find becomes our answer.

Finally, we need to figure out a way to efficiently check if a subarray is nice. Since we need to track which bits are used across multiple numbers, a **bitmask** is the perfect tool for this job. As we traverse the potential subarray, we maintain a single integer (the bitmask) where each bit represents whether that position has been "used" by any number so far.

For example, consider numbers 4 (`100` in binary), 2 (`010` in binary), and 1 (`001` in binary). As we check each number, we test if any of its bits overlap with our existing bitmask. If there is an overlap, the subarray is no longer "nice" since two numbers now share a set bit.

Otherwise, we add the current number's bits into our bitmask using the OR operation. This operation updates our tracking of occupied bit positions. If we process the entire subarray without encountering overlapping bits, we confirm that it qualifies as a "nice" subarray.

// We can use Binary Search in ques where we need to find longest subarray satisfying certain property as any shorter one would also satisfy that.



```cpp
class Solution {
public:
    int longestNiceSubarray(vector<int>& nums) {
        // Binary search for the longest nice subarray length
        int left = 0, right = nums.size();
        int result = 1;  // Minimum answer is 1 (as subarrays of length 1 are
                         // always nice)

        while (left <= right) {
            int length = left + (right - left) / 2;
            if (canFormNiceSubarray(length, nums)) {
                result = length;    // Update the result
                left = length + 1;  // Try to find a longer subarray
            } else {
                right = length - 1;  // Try a shorter length
            }
        }
        return result;
    }

private:
    bool canFormNiceSubarray(int length, vector<int>& nums) {
        if (length <= 1) return true;  // Subarray of length 1 is always nice

        // Try each possible starting position for subarray of given length
        for (int start = 0; start <= nums.size() - length; ++start) {
            int bitMask = 0;  // Tracks the bits used in the current subarray
            bool isNice = true;

            // Check if the subarray starting at 'start' with 'length' elements
            // is nice
            for (int pos = start; pos < start + length; ++pos) {
                // If current number shares any bits with existing mask,
                // the subarray is not nice
                if ((bitMask & nums[pos]) != 0) {
                    isNice = false;
                    break;
                }
                bitMask |= nums[pos];  // Add current number's bits to the mask
            }

            if (isNice)
                return true;  // Found a nice subarray of the specified length
```

T.C. $O(n^2 \log n)$

S.C. $O(1)$.