<https://leetcode.com/problems/longest-nice-subarray/description/>

You are given an array nums consisting of positive integers.

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

**Attempt 1: 2024-12-26**

**Solution 1: Bit Manipulation + Not fixed length Sliding Window (60 min)**

class Solution {

    public int longestNiceSubarray(int[] nums) {

        int maxLen = 0;

        // Create a bitmask to keep track of the bits in the current subarray

        // it is also initiated to 0, this mask will store the bitwise OR of

        // all numbers currently in the window

        int mask = 0;

        int i = 0;

        for(int j = 0; j < nums.length; j++) {

            // 'And &' used to check if adding nums[right] violates the

            // "nice" condition, requires '0', if not return '0' violates

            // Note: '&' is bit operation

            // (mask & nums[j]) != 0 NOT EQUAL to (mask & nums[j]) == 1

            while((mask & nums[j]) != 0) {

                // 'XOR ^' used to remove the leftmost number from the

                // bitMask

                // Why XOR is working is because xor will only make those

                // bits 1 which are different. So suppose we have 11001

                // and we want to remove 9 - 1001 so . 11001^1001 will

                // give 10000 , and we can see that only those bits are

                // off which we wanted to remove.

                mask ^= nums[i];

                i++;

            }

            // 'OR |' used to add the current number to the bitMask

            // OR operation is setting those bits on which are already

            // on by any of the element in the window.

            mask |= nums[j];

            maxLen = Math.max(maxLen, j - i + 1);

        }

        return maxLen;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Explain why and how we initialize 'mask' as '0' and use 'AND', 'OR' and 'XOR' in the Sliding Window process**

**Refer to**

<https://leetcode.com/problems/longest-nice-subarray/solutions/2527272/sliding-window/comments/1586685>

A sub array is valid when no two elements have same position set bit. Like 1001 and 110 can be in one window but not with 1111.

1. We are checking using **AND** operation because if no two bit in our window have same set bit then the AND operation will give 0 if not then we need to shrink the window for making it valid for our current element's insertion.

2. So **OR** operation is setting those bits on which are already on by any of the element in the window. So if the window becomes invalid then then the cur element has some bit which has already been set by any of the previously added element, so we will shrink the window till it becomes valid again by removing the left most element.

3. When we are removing an element then we are also removing it's set bits so we do the **XOR** operation of used with the leftmost element to remove it's set bits from consideration.

4. **Why XOR is working is because xor will only make those bits 1 which are different. So suppose we have 11001 and we want to remove 9 - 1001 so . 11001^1001 will give 10000 , and we can see that only those bits are off which we wanted to remove.**

5. Then when the window becomes valid we can insert our new element in it by doing OR operation with the used variable , so that we can mark those bits used which are set in cur element.

Then we can check the size of the window for our answer.

**Refer to chatGPT**

**Problem Description:**

You are given an integer array nums. A subarray is "nice" if the bitwise AND of every pair of elements in the subarray is 0. You need to return the length of the longest "nice" subarray.

**Solution Explanation:**

Use a **sliding window** to find the longest subarray where the bitwise AND of every pair is 0.

Maintain a variable bitMask to represent the bitwise OR of the current window.

Expand the window by adding elements to it. If adding an element violates the condition (i.e., bitMask & nums[i] != 0), shrink the window from the left until the condition is restored.

public class Solution {

public int longestNiceSubarray(int[] nums) {

int n = nums.length;

int left = 0, bitMask = 0, maxLen = 0;

for (int right = 0; right < n; right++) {

// Check if the current number overlaps with the existing bitMask

while ((bitMask & nums[right]) != 0) {

// Remove the leftmost number from the bitMask

bitMask ^= nums[left];

left++;

}

// Add the current number to the bitMask

bitMask |= nums[right];

// Update the maximum length

maxLen = Math.max(maxLen, right - left + 1);

}

return maxLen;

}

}

**Key Points:**

Bit Manipulation:

**(bitMask & nums[right]) != 0: Checks if adding nums[right] violates the "nice" condition.**

**bitMask ^= nums[left]: Removes the leftmost number from the bitMask using XOR.**

**bitMask |= nums[right]: Adds the current number to the bitMask using OR.**

Sliding Window:

Adjust the window by incrementing left when a conflict occurs.

Expand the window by incrementing right.

Time Complexity:

Each number is added and removed from the bitMask at most once, so the time complexity is O(n).

Space Complexity:

The solution uses O(1) additional space since the bitMask is a single integer.

This is an efficient and elegant solution leveraging bit manipulation and sliding window techniques!

**Refer to**

<https://algo.monster/liteproblems/2401>

**Problem Description**

This LeetCode problem requires us to find the longest subarray in a given array nums of positive integers where the subarray is considered *nice*. A subarray is defined as nice if the bitwise AND of every pair of elements at different positions is equal to 0. A subarray is a contiguous part of the array, and it is given that any subarray with a length of 1 is always nice.

**Intuition**

**The solution for the problem leverages the properties of bitwise AND operation. The bitwise AND of any two numbers is 0 only if they do not share any common set bits in their binary representation.**

The approach taken here uses a [sliding window](https://algo.monster/problems/sliding_window_maximum) technique to iterate over the array, using two pointers (or indices) to define the current window which is examined for being a nice subarray. **As we iterate from 0 to n-1 (where n is the length of the array), we maintain a variable mask which is the bitwise OR of all numbers in the current window. This mask helps in checking if any incoming number will create a pair with a non-zero AND with any number in the current window.**

When we attempt to extend the window by including the next number, we check if the bitwise AND of this new number (x) with the current mask is zero. If it is not zero, this means the number shares at least one common set bit with one of the numbers in the current window and hence cannot be included to maintain it a nice subarray. We then shrink the window from the left by removing the leftmost element and updating the mask until the mask AND the new number is 0.

The variable ans keeps track of the length of the longest nice subarray encountered so far, and we update it every time we find a larger nice subarray. The current window's length is calculated as i - j + 1, where i is the end of the current window and j is the start.

By scanning and adjusting the window this way until the end of the array is reached, we ensure that the longest nice subarray is found.

**Solution Approach**

The reference solution uses a [sliding window](https://algo.monster/problems/sliding_window_maximum) technique to find the length of the longest nice subarray. Here's how the implementation unfolds:

An integer ans is initialized to 0 to keep track of the length of the maximum nice subarray found so far.

Two pointers, i and j, are initialized to track the starting and ending index of the current subarray, starting from 0.

**An integer mask is also initiated to 0. This mask will store the bitwise OR of all numbers currently in the window.**

The implementation follows these steps:

Enumerate through each element x in nums using its index i.

While the current number x has a non-zero AND with the mask (i.e., mask & x is not 0), i.e., if the current element shares a common set bit with any element in the subarray represented by mask, it suggests that the subarray is not nice anymore with the addition of x:

**Exclude the leftmost element from the subarray to make room for x by XOR-ing the leftmost element nums[j] with the mask. This effectively removes the bits of nums[j] from mask.**

Increment j to shrink the subarray from the left.

After ensuring that including x will not break the nice property (the AND of every pair is 0), we can:

Update the ans variable with the maximum of its current value and the size of the window which is i - j + 1.

**Include x in the mask by performing a bitwise OR (mask |= x).**

This process continues for every element in the array. By iteratively adjusting the window and updating the mask, the algorithm ensures it never includes a pair that would result in a non-zero AND, hence maintaining the nice property of the subarray.

After completing the iteration over all elements, the final value of ans yields the length of the longest nice subarray.

**Example Walkthrough**

Let's walk through an example using the solution approach.

Consider the following array nums: [3,6,1,2]

Initialize ans to 0.

Start with i = 0, j = 0, and mask = 0.

Now, we'll iterate over the array while applying the steps outlined in the solution approach:

For i = 0, x = nums[0] = 3.

**mask & x is 0 since mask starts at 0 and any number AND 0 is 0.**

Update ans to max(ans, i - j + 1), which is 1.

**Update mask to mask | x, which now becomes 3.**

Move to i = 1, x = nums[1] = 6.

mask & x is 2 (binary 0010), which is not 0, so we need to adjust the subarray.

We shrink the window by removing nums[j] (3) from mask. New mask is 3 XOR 3 = 0.

Increment j to 1, and now the subarray is empty.

We retry with x = 6, mask & x is now 0, so we can proceed.

Update ans to max(ans, i - j + 1), which is 1.

Update mask to mask | x, which now becomes 6.

Move to i = 2, x = nums[2] = 1.

mask & x is 0, it's already a nice subarray upon adding x.

Update ans to max(ans, i - j + 1), which is now 2 as the subarray from nums[1] to nums[2] is nice.

Update mask to mask | x, which becomes 7.

Move to i = 3, x = nums[3] = 2.

mask & x is 2, which is not 0, so the subarray is not nice with the addition of x.

We shrink the window from the left by excluding nums[j] (6) from mask. After XOR with 6, the new mask is 1.

Increment j to 2, and now the subarray starts at nums[2].

We retry with x = 2, mask & x is now 0, so we can proceed.

Update ans to max(ans, i - j + 1), which remains 2.

Include x in the mask, new mask is 1 | 2 = 3.

After going through the array, the final value of ans is 2, indicating the length of the longest nice subarray is 2, which corresponds to the subarray [1,2].

**Solution Implementation**

class Solution {

public int longestNiceSubarray(int[] nums) {

// Initialize the answer to track the length of the longest nice subarray

int longestNiceLength = 0;

// Create a bitmask to keep track of the bits in the current subarray

int currentMask = 0;

// Two pointers - j represents the start of the current subarray

// i represents the current end of the subarray being considered

for (int startIdx = 0, endIdx = 0; endIdx < nums.length; ++endIdx) {

// Keep removing numbers from the start of the subarray until

// the current number can fit in without sharing any common set bits

while ((currentMask & nums[endIdx]) != 0) {

// XOR operation removes the bits of nums[startIdx] from currentMask

currentMask ^= nums[startIdx++];

}

// Update the longestNiceLength if the current subarray is longer

longestNiceLength = Math.max(longestNiceLength, endIdx - startIdx + 1);

// Include the current number's bits into the currentMask

currentMask |= nums[endIdx];

}

// Return the length of the longest nice subarray

return longestNiceLength;

}

}

**Time and Space Complexity**

The time complexity of the provided code is O(N), where N is the length of the nums array. This is because the code iterates over all the elements of nums exactly once with the outer loop (for i, x in enumerate(nums):). The inner while-loop (while mask & x:) only processes each element at most once across the entire runtime because once an element has been removed from the mask (mask ^= nums[j]), it does not get reprocessed. Thus, each element contributes at most two operations: one for adding it to the mask and one for possibly removing it from the mask, leading to a linear runtime overall.

The space complexity of the code is O(1) since the amount of extra space used by the algorithm does not scale with the input size N. The data structures used (ans, j, and mask) require a constant amount of space regardless of the input size.

**Refer to**

[L424.P2.6.Longest Repeating Character Replacement (Ref.L340)](note://1581773836A946879EB5237C85BFF6C2)

[L2024.Maximize the Confusion of an Exam (Ref.L424,L2379,L2401)](note://WEB67a85ae88eb0331483ca03c2b1075779)