# 2411. Smallest Subarrays With Maximum Bitwise OR

Bit Manipulation Array Binary Search Sliding Window Medium

### **Problem Description**

to n - 1, the length of the smallest subarray starting at i that produces the maximum bitwise OR value compared to all possible subarrays starting at that same index i. We define the bitwise OR of an array as the result of applying the bitwise OR operation to all numbers in it. To clarify with an example, if the array was [1,2,3], the subarrays starting from index 0 would be [1], [1,2], and [1,2,3]. If

In this problem, you're presented with an array nums of non-negative integers. Your task is to determine, for each index i from 0

the maximum bitwise OR achievable is with subarray [1,2], and that value cannot be achieved with a smaller subarray from that index, then the length you should report for index 0 would be the length of [1,2] which is 2. You need to perform this computation efficiently for every index of the array.

Intuition

extend far enough to include any bits that wouldn't otherwise be set to 1 without it. What's less intuitive is figuring out how far

Bitwise OR operation has an 'accumulative' property; once a bit is set to 1, it remains 1 irrespective of the OR operations that

By doing this, when we're considering a given index i, we can look at our collection of indices, and figure out the furthest one we

Your final output should be an integer array answer, where answer[i] is the length of the smallest subarray starting at index i

The solution to this problem uses a bit-manipulation trick. For each index, we know that the maximum bitwise OR subarray must

with the maximum bitwise OR value.

#### that is without checking every subarray.

follow. This leads us to a realization: we don't need to continue our subarray beyond the point where all bits that can be set to 1, are set. The insight is to iterate backwards from the end of the array, keeping track of the most recent index where each bit was set to 1.

The process involves a 32-length array (for each bit in the integer) where we keep updating our 'latest' indices as we move from the end to the start of the array. For each number, we consider every bit, update the latest index if the bit is 1, and find the distance to the farthest index we need to include if the bit is not set. Hence, we derive the minimum length for the maximum OR

subarray starting at that index. The solution brilliantly captures the bit-level details and aggregates them into a subarray problem.

need to reach - this gives us the length of the minimum subarray with the maximum bitwise OR.

1. Iterate through each element in nums in reverse order (starting from the last element).

represents the length needed to include the j-th bit.

 $\circ$  f = [-1, -1, -1, -1] (one entry for each bit)

• i=2 (Last element, nums[2] = 0111):

Start iterating through **nums** in reverse order:

i=1 (Second element, nums[1] = 1011):

For bit 1: It is set, so update f[1] = 1.

3. Iterate through each of the 32 bits (0 to 31) to check which bits are set to 1 in the current element.

binary representations of the numbers in the array are 0011, 1011, and 0111 respectively.

■ Initialize t = 1 (the smallest subarray always includes at least nums[i] itself).

**Solution Approach** The given Python solution first initializes two arrays: ans which will contain our result - the smallest subarray sizes, and f to keep track of the most recent index for each of the 32 bits (representing an integer in the array).

#### If the bit at position j in nums[i] is set to 1, update f[j] to be i. This step guarantees that f[j] always points to the latest index i

position just once.

(the element itself).

where the j-th bit was set. ∘ If the bit at position j is not set in nums[i] (the value is 0), and f[j] isn't -1 (indicating that we've seen the j-th bit set in some higher index), then we must extend our subarray to include this bit. Thus, t is updated to be the maximum of itself and f[j] - i + 1 which

2. For the current element nums [i], initialize a temporary variable t with the value 1 as the smallest subarray size to consider is always at least 1

- maximizes the bitwise OR value.
- 5. Once the loop is finished, we return the ans array which now contains the desired subarray sizes for each index in the original nums array. The solution employs bit manipulation and array indexing in an elegant way to avoid checking every possible subarray, reducing

what would be a potentially quadratic-time algorithm to a linear-time solution since it iterates through the array and each bit

Let's walk through a simple example to illustrate the solution approach. Imagine we have the array nums = [3, 11, 7]. The

4. After checking all bits, ans [i] is updated with the value in t, which now represents the minimum subarray size starting at index i that

- **Example Walkthrough**
- Initialize ans array to hold the result and f array to keep track of the most recent indices where each bit is set to 1. Both arrays have a size of 3, the length of nums, and f is initialized with -1 to indicate that no indices have been visited yet: o ans = [0, 0, 0] (to be filled with results)

## Check each bit:

For bit 0: It is set, so update f[0] = 2. For bit 1: It is set, so update f[1] = 2. For bit 2: It is set, so update f[2] = 2.

- Update ans[2] with t = 1. At this point:  $\circ$  ans = [0, 0, 1] f = [2, 2, 2, -1]
- Reset t = 1. Check each bit: For bit 0: It is set, so update f[0] = 1.

For bit 2: It is not set, but f[2] is 2, so update t to 2 − 1 + 1 = 2.

■ For bit 2: It is not set, and f[2] is 2, so update t to the maximum of t and (2 - 0 + 1) = 3.

smallest subarray sizes starting at each index i that achieves the maximum bitwise OR value.

■ For bit 3: It is not set, and f[3] is -1 (no previous 1's).

For bit 3: It is set, so update f[3] = 1. ■ Update ans[1] with t = 2.

At this point:

 $\circ$  f = [1, 1, 2, 1]

- $\circ$  ans = [0, 2, 1]
  - i=0 (First element, nums[0] = 0011): ■ Reset t = 1.
    - Check each bit: For bit 0: It is set, so update f[0] = 0. For bit 1: It is set, so update f[1] = 0.

showcasing the power of bit manipulation and dynamic programming.

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

# Traverse the input array in reverse order

if (((nums[i] >> j) & 1) == 1) {

latestOneBitIndices[i] = i;

// Set the computed minimum subarray size.

answer[i] = subarraySize;

// of any subarray starting at that element.

\* @param nums The input array of numbers.

for (let i = size - 1; i >= 0; --i) {

function smallestSubarrays(nums: number[]): number[] {

// the smallest subarray starting at each index

// Check each bit position from 0 to 31

} else if (lastIndex[bit] !== -1) {

for (let bit = 0; bit < 32; ++bit) {

lastIndex[bit] = i;

\* @returns An array of the smallest subarray sizes for each element.

// Iterate through the array in reverse order to determine

const size: number = nums.length; // Store the size of the input array

const result: number[] = new Array(size); // This will store the answer

let subarraySize: number = 1; // Minimum subarray size is 1 (the element itself)

// If the current bit is not set, calculate the subarraySize needed

if ((nums[i] >> bit) & 1) { // If the current bit is set in nums[i]

// to include this bit from further elements in the array

subarraySize = Math.max(subarraySize, lastIndex[bit] - i + 1);

// Update the last position of this bit to the current index

vector<int> smallestSubarrays(vector<int>& nums) {

return answer;

} else if (latest0neBitIndices[i] != -1) {

// Return the array containing the minimum subarray sizes.

// to calculate the size of the subarray.

// Function to find the smallest subarrays for each element such that each

// subarray's bitwise OR is at least as large as the maximum bitwise OR

# Get the length of the input array

for i in range(length -1, -1, -1):

- For bit 3: It is not set, and f[3] is 1, so update t to the maximum of t and (1 0 + 1) = 2. ■ Update ans [0] with the maximum t = 3.
- $\circ$  f = [0, 0, 2, 1] After iterating through all indices, and now contains the correct answer. The final array and = [3, 2, 1] represents the
- Solution Implementation

from typing import List

**Python** 

class Solution:

At this point:

 $\circ$  ans = [3, 2, 1]

# Initialize the result array with 1s, as the smallest non-empty subarray is the number itself. result = [1] \* length # Initialize an array to store the latest positions of bits (0 to 31) seen in the binary representation of the numbers

length = len(nums)

 $last_seen_at = [-1] * 32$ 

for i in range(32):

result[i] = max\_size

return result

Java

```
if (nums[i] >> j) & 1: # Check if bit 'j' is set in nums[i]
    last seen at[i] = i # Update the position of bit 'i'
elif last seen at[j] != -1: # If bit 'j' was found at a position ahead of 'i'
    # Update 'max size': the size of the subarray that includes the current number and
   # the last occurrence of every bit that is not present in the current number
```

 $\max size = 1$  # Initialize current size to 1 (the size of a subarray containing only nums[i])

# Go through each of the 32 bits to update 'last\_seen\_at' and calculate the subarray size

max size = max(max size, last seen at[i] - i + 1)

// Update the latest index where this bit was set.

// If the bit is not set, we use the latest index where this bit was set,

subarraySize = Math.max(subarraySize, latestOneBitIndices[j] - i + 1);

# Return the array containing the sizes of the smallest subarrays

# Record the required subarray size for the current starting index 'i'

In this example, the smallest subarray starting at index 0 with the maximum bitwise OR is [3, 11, 7], at index 1 is [11, 7], and

at index 2 is [7]. The algorithm efficiently identifies these subarrays without having to examine every possible subarray,

```
import java.util.Arrays;
class Solution {
   public int[] smallestSubarrays(int[] nums) {
       // Number of elements in the given array.
       int length = nums.length;
       // Array to store the answer which is the length of smallest subarrays.
       int[] answer = new int[length];
       // Frequency array for each bit position (0 to 31 for 32-bit integers).
       int[] latestOneBitIndices = new int[32];
       // Initialize with -1, it signifies that we haven't seen a bit 1 in that position so far.
       Arrays.fill(latestOneBitIndices, -1);
       // Start from the end of the input array and move towards the start.
        for (int i = length - 1; i >= 0; --i) {
            int subarraySize = 1; // Initialize the minimum subarray size to 1 for each number.
            // Check each bit position.
            for (int i = 0: i < 32: ++i) {
               // If the i-th bit of the current number is set (equal to 1).
```

```
C++
#include <vector>
using namespace std;
```

public:

class Solution {

```
int size = nums.size(); // Store the size of the input array
        vector<int> lastIndex(32, -1); // Track the last position of each bit
        vector<int> result(size); // This will store the answer
        // Iterate through the array in reverse order to determine
        // the smallest subarray starting at each index
        for (int i = size - 1; i >= 0; --i) {
            int subarraySize = 1; // Minimum subarray size is 1 (the element itself)
            // Check each bit position from 0 to 31
            for (int bit = 0; bit < 32; ++bit) {
                if ((nums[i] >> bit) & 1) { // If the current bit is set in nums[i]
                    // Update the last position of this bit to the current index
                    lastIndex[bit] = i:
                } else if (lastIndex[bit] != -1) {
                    // If the current bit is not set, calculate the subarraySize needed
                    // to include this bit from further elements in the array
                    subarraySize = max(subarraySize, lastIndex[bit] - i + 1);
            // After checking all the bits, store the result subarray size
            result[i] = subarraySize;
        return result; // Return the final vector with smallest subarray sizes
};
// Note: Additional includes or namespace imports might be required
// depending on the scope of the snippet and the desired coding environment.
TypeScript
// TypeScript has its own types, so no need to import C++'s <vector>
// Defining a function to find the smallest subarrays
/**
* Finds the smallest subarrays for each element such that each subarray's bitwise OR
* is at least as large as the maximum bitwise OR of any subarray starting at that element.
```

const lastIndex: number[] = new Array(32).fill(-1); // Track the last position of each bit, for up to 32 bits

```
result[i] = subarraySize;
   return result; // Return the final array with smallest subarray sizes
// Example usage:
```

// After checking all the bits, store the result subarray size

```
// const nums: number[] = [1, 2, 3, 4];
// const result: number[] = smallestSubarrays(nums);
// console.log(result); Output: [4, 3, 2, 1]
from typing import List
class Solution:
   def smallestSubarrays(self, nums: List[int]) -> List[int]:
       # Get the length of the input array
        length = len(nums)
       # Initialize the result array with 1s, as the smallest non-empty subarray is the number itself.
       result = [1] * length
       # Initialize an array to store the latest positions of bits (0 to 31) seen in the binary representation of the numbers
        last_seen_at = [-1] * 32
       # Traverse the input array in reverse order
       for i in range(length -1, -1, -1):
           \max size = 1 # Initialize current size to 1 (the size of a subarray containing only nums[i])
           # Go through each of the 32 bits to update 'last_seen_at' and calculate the subarray size
            for j in range(32):
               if (nums[i] >> i) & 1: # Check if bit 'i' is set in nums[i]
                    last seen at [i] = i # Update the position of bit 'i'
```

elif last seen at[i] !=-1: # If bit 'i' was found at a position ahead of 'i'

max size = max(max size, last seen at[i] - i + 1)

# Return the array containing the sizes of the smallest subarrays

# Record the required subarray size for the current starting index 'i'

# Update 'max size': the size of the subarray that includes the current number and

# the last occurrence of every bit that is not present in the current number

#### of each element, including the element itself. **Time Complexity**

result[i] = max\_size

Time and Space Complexity

return result

The time complexity of the given code is 0(n \* b), where n is the length of the nums array, and b is the number of bits used to represent the numbers. Since the input numbers are integers and Python's integers have a fixed number of bits (which is 32 bits for standard integers), the value of b can be considered a constant, thus simplifying the time complexity further to O(n).

The given Python code is designed to find the smallest subarrays that contain all of the bits that appear at least once to the right

**Space Complexity** 

This complexity is derived from the outer loop running n times (from n - 1 to 0) and the inner loop over b bits running a constant 32 times. The algorithms only perform a fixed set of operations within the inner loop, which does not depend on n.

in an integer. Space is also used for the output list ans of size n, but when talking about space complexity we generally do not count the space required for output, or if we do, we state the complexity as additional space beyond the required output.

The space complexity of the code is O(b), since we are maintaining a fixed-size array f of size 32, corresponding to the 32 bits

If one includes the space taken by the output, then it would be O(n). However, since the space for the output is typically not counted, and the auxiliary space used is only the fixed-size list f, the space complexity remains 0(b), which reduces to 0(1) since the number of bits is constant and does not grow with the input.