

# Problem 2411: Smallest Subarrays With Maximum Bitwise OR

## Problem Information

Difficulty: Medium

Acceptance Rate: 0.00%

Paid Only: No

## Problem Description

You are given a

0-indexed

array

nums

of length

n

, consisting of non-negative integers. For each index

i

from

0

to

n - 1

, you must determine the size of the

minimum sized

non-empty subarray of

nums

starting at

i

(

inclusive

) that has the

maximum

possible

bitwise OR

.

In other words, let

B

ij

be the bitwise OR of the subarray

nums[i...j]

. You need to find the smallest subarray starting at

i

, such that bitwise OR of this subarray is equal to

$\max(B$

$i$

$)$

where

$i \leq k \leq n - 1$

.

The bitwise OR of an array is the bitwise OR of all the numbers in it.

Return

an integer array

answer

of size

$n$

where

answer[i]

is the length of the

minimum

sized subarray starting at

$i$

with

maximum

bitwise OR.

A

subarray

is a contiguous non-empty sequence of elements within an array.

Example 1:

Input:

nums = [1,0,2,1,3]

Output:

[3,3,2,2,1]

Explanation:

The maximum possible bitwise OR starting at any index is 3. - Starting at index 0, the shortest subarray that yields it is [1,0,2]. - Starting at index 1, the shortest subarray that yields the maximum bitwise OR is [0,2,1]. - Starting at index 2, the shortest subarray that yields the maximum bitwise OR is [2,1]. - Starting at index 3, the shortest subarray that yields the maximum bitwise OR is [1,3]. - Starting at index 4, the shortest subarray that yields the maximum bitwise OR is [3]. Therefore, we return [3,3,2,2,1].

Example 2:

Input:

nums = [1,2]

Output:

[2,1]

Explanation:

Starting at index 0, the shortest subarray that yields the maximum bitwise OR is of length 2.  
Starting at index 1, the shortest subarray that yields the maximum bitwise OR is of length 1.  
Therefore, we return [2,1].

Constraints:

$n == \text{nums.length}$

$1 \leq n \leq 10$

5

$0 \leq \text{nums}[i] \leq 10$

9

## Code Snippets

**C++:**

```
class Solution {
public:
    vector<int> smallestSubarrays(vector<int>& nums) {

    }
};
```

**Java:**

```
class Solution {
    public int[] smallestSubarrays(int[] nums) {

    }
}
```

**Python3:**

```

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

```

## Python:

```

class Solution(object):
    def smallestSubarrays(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """

```

## JavaScript:

```

/**
 * @param {number[]} nums
 * @return {number[]}
 */
var smallestSubarrays = function(nums) {

};

```

## TypeScript:

```

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

};

```

## C#:

```

public class Solution {
    public int[] SmallestSubarrays(int[] nums) {

    }
}

```

## C:

```

/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* smallestSubarrays(int* nums, int numsSize, int* returnSize) {

```

```
}
```

### Go:

```
func smallestSubarrays(nums []int) []int {  
  
}
```

### Kotlin:

```
class Solution {  
    fun smallestSubarrays(nums: IntArray): IntArray {  
  
    }  
}
```

### Swift:

```
class Solution {  
    func smallestSubarrays(_ nums: [Int]) -> [Int] {  
  
    }  
}
```

### Rust:

```
impl Solution {  
    pub fn smallest_subarrays(nums: Vec<i32>) -> Vec<i32> {  
  
    }  
}
```

### Ruby:

```
# @param {Integer[]} nums  
# @return {Integer[]}  
def smallest_subarrays(nums)  
  
end
```

### PHP:

```

class Solution {

  /**
   * @param Integer[] $nums
   * @return Integer[]
   */
  function smallestSubarrays($nums) {

  }

}

```

### Dart:

```

class Solution {
  List<int> smallestSubarrays(List<int> nums) {

  }

}

```

### Scala:

```

object Solution {
  def smallestSubarrays(nums: Array[Int]): Array[Int] = {

  }

}

```

### Elixir:

```

defmodule Solution do
  @spec smallest_subarrays(nums :: [integer]) :: [integer]
  def smallest_subarrays(nums) do

  end

end

```

### Erlang:

```

-spec smallest_subarrays(Nums :: [integer()]) -> [integer()].
smallest_subarrays(Nums) ->
.

```

### Racket:

```
(define/contract (smallest-subarrays nums)
  (-> (listof exact-integer?) (listof exact-integer?))
)
```

## Solutions

### C++ Solution:

```
/*
 * Problem: Smallest Subarrays With Maximum Bitwise OR
 * Difficulty: Medium
 * Tags: array, search
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

class Solution {
public:
    vector<int> smallestSubarrays(vector<int>& nums) {

    }
};
```

### Java Solution:

```
/**
 * Problem: Smallest Subarrays With Maximum Bitwise OR
 * Difficulty: Medium
 * Tags: array, search
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

class Solution {
    public int[] smallestSubarrays(int[] nums) {

    }
}
```

```
}
```

### Python3 Solution:

```
"""
Problem: Smallest Subarrays With Maximum Bitwise OR
Difficulty: Medium
Tags: array, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
    def smallestSubarrays(self, nums: List[int]) -> List[int]:
        # TODO: Implement optimized solution
        pass
```

### Python Solution:

```
class Solution(object):
    def smallestSubarrays(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
```

### JavaScript Solution:

```
/**
 * Problem: Smallest Subarrays With Maximum Bitwise OR
 * Difficulty: Medium
 * Tags: array, search
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

/**
```

```

* @param {number[]} nums
* @return {number[]}
*/
var smallestSubarrays = function(nums) {

};

```

### TypeScript Solution:

```

/**
 * Problem: Smallest Subarrays With Maximum Bitwise OR
 * Difficulty: Medium
 * Tags: array, search
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

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

};

```

### C# Solution:

```

/*
 * Problem: Smallest Subarrays With Maximum Bitwise OR
 * Difficulty: Medium
 * Tags: array, search
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

public class Solution {
    public int[] SmallestSubarrays(int[] nums) {

    }
}

```

### C Solution:

```
/*
 * Problem: Smallest Subarrays With Maximum Bitwise OR
 * Difficulty: Medium
 * Tags: array, search
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(1) to O(n) depending on approach
 */

/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* smallestSubarrays(int* nums, int numsSize, int* returnSize) {

}
```

### Go Solution:

```
// Problem: Smallest Subarrays With Maximum Bitwise OR
// Difficulty: Medium
// Tags: array, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func smallestSubarrays(nums []int) []int {

}
```

### Kotlin Solution:

```
class Solution {
    fun smallestSubarrays(nums: IntArray): IntArray {

    }
}
```

### Swift Solution:

```

class Solution {
  func smallestSubarrays(_ nums: [Int]) -> [Int] {

  }
}

```

### Rust Solution:

```

// Problem: Smallest Subarrays With Maximum Bitwise OR
// Difficulty: Medium
// Tags: array, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
  pub fn smallest_subarrays(nums: Vec<i32>) -> Vec<i32> {

  }
}

```

### Ruby Solution:

```

# @param {Integer[]} nums
# @return {Integer[]}
def smallest_subarrays(nums)

end

```

### PHP Solution:

```

class Solution {

  /**
   * @param Integer[] $nums
   * @return Integer[]
   */
  function smallestSubarrays($nums) {

  }
}

```

### Dart Solution:

```
class Solution {  
  List<int> smallestSubarrays(List<int> nums) {  
  
  }  
}
```

### Scala Solution:

```
object Solution {  
  def smallestSubarrays(nums: Array[Int]): Array[Int] = {  
  
  }  
}
```

### Elixir Solution:

```
defmodule Solution do  
  @spec smallest_subarrays(nums :: [integer]) :: [integer]  
  def smallest_subarrays(nums) do  
  
  end  
end
```

### Erlang Solution:

```
-spec smallest_subarrays(Nums :: [integer()]) -> [integer()].  
smallest_subarrays(Nums) ->  
.
```

### Racket Solution:

```
(define/contract (smallest-subarrays nums)  
  (-> (listof exact-integer?) (listof exact-integer?))  
  )
```