

# Problem 3022: Minimize OR of Remaining Elements Using Operations

## Problem Information

Difficulty: Hard

Acceptance Rate: 0.00%

Paid Only: No

## Problem Description

You are given a

0-indexed

integer array

nums

and an integer

k

.

In one operation, you can pick any index

i

of

nums

such that

$0 \leq i < \text{nums.length} - 1$

and replace

`nums[i]`

and

`nums[i + 1]`

with a single occurrence of

`nums[i] & nums[i + 1]`

, where

&

represents the bitwise

AND

operator.

Return

the

minimum

possible value of the bitwise

OR

of the remaining elements of

`nums`

after applying

at most

k

operations

.

Example 1:

Input:

nums = [3,5,3,2,7], k = 2

Output:

3

Explanation:

Let's do the following operations: 1. Replace nums[0] and nums[1] with (nums[0] & nums[1]) so that nums becomes equal to [1,3,2,7]. 2. Replace nums[2] and nums[3] with (nums[2] & nums[3]) so that nums becomes equal to [1,3,2]. The bitwise-or of the final array is 3. It can be shown that 3 is the minimum possible value of the bitwise OR of the remaining elements of nums after applying at most k operations.

Example 2:

Input:

nums = [7,3,15,14,2,8], k = 4

Output:

2

Explanation:

Let's do the following operations: 1. Replace  $\text{nums}[0]$  and  $\text{nums}[1]$  with  $(\text{nums}[0] \& \text{nums}[1])$  so that  $\text{nums}$  becomes equal to [3,15,14,2,8]. 2. Replace  $\text{nums}[0]$  and  $\text{nums}[1]$  with  $(\text{nums}[0] \& \text{nums}[1])$  so that  $\text{nums}$  becomes equal to [3,14,2,8]. 3. Replace  $\text{nums}[0]$  and  $\text{nums}[1]$  with  $(\text{nums}[0] \& \text{nums}[1])$  so that  $\text{nums}$  becomes equal to [2,2,8]. 4. Replace  $\text{nums}[1]$  and  $\text{nums}[2]$  with  $(\text{nums}[1] \& \text{nums}[2])$  so that  $\text{nums}$  becomes equal to [2,0]. The bitwise-or of the final array is 2. It can be shown that 2 is the minimum possible value of the bitwise OR of the remaining elements of  $\text{nums}$  after applying at most k operations.

Example 3:

Input:

$\text{nums} = [10,7,10,3,9,14,9,4]$ ,  $k = 1$

Output:

15

Explanation:

Without applying any operations, the bitwise-or of  $\text{nums}$  is 15. It can be shown that 15 is the minimum possible value of the bitwise OR of the remaining elements of  $\text{nums}$  after applying at most k operations.

Constraints:

$1 \leq \text{nums.length} \leq 10$

5

$0 \leq \text{nums}[i] < 2$

30

$0 \leq k < \text{nums.length}$

## Code Snippets

**C++:**

```
class Solution {  
public:  
    int minOrAfterOperations(vector<int>& nums, int k) {  
  
    }  
};
```

**Java:**

```
class Solution {  
public int minOrAfterOperations(int[] nums, int k) {  
  
}  
}
```

**Python3:**

```
class Solution:  
    def minOrAfterOperations(self, nums: List[int], k: int) -> int:
```

**Python:**

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

**JavaScript:**

```
/**  
 * @param {number[]} nums  
 * @param {number} k  
 * @return {number}  
 */  
var minOrAfterOperations = function(nums, k) {  
  
};
```

**TypeScript:**

```
function minOrAfterOperations(nums: number[], k: number): number {  
}  
};
```

**C#:**

```
public class Solution {  
    public int MinOrAfterOperations(int[] nums, int k) {  
  
    }  
}
```

**C:**

```
int minOrAfterOperations(int* nums, int numssSize, int k) {  
  
}
```

**Go:**

```
func minOrAfterOperations(nums []int, k int) int {  
  
}
```

**Kotlin:**

```
class Solution {  
    fun minOrAfterOperations(nums: IntArray, k: Int): Int {  
  
    }  
}
```

**Swift:**

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

**Rust:**

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

### Ruby:

```
# @param {Integer[]} nums
# @param {Integer} k
# @return {Integer}
def min_or_after_operations(nums, k)

end
```

### PHP:

```
class Solution {

    /**
     * @param Integer[] $nums
     * @param Integer $k
     * @return Integer
     */
    function minOrAfterOperations($nums, $k) {

    }
}
```

### Dart:

```
class Solution {
    int minOrAfterOperations(List<int> nums, int k) {
        }
    }
```

### Scala:

```
object Solution {
    def minOrAfterOperations(nums: Array[Int], k: Int): Int = {
        }
```

```
}
```

### Elixir:

```
defmodule Solution do
  @spec min_or_after_operations(nums :: [integer], k :: integer) :: integer
  def min_or_after_operations(nums, k) do
    end
  end
```

### Erlang:

```
-spec min_or_after_operations(Nums :: [integer()], K :: integer()) ->
  integer().
min_or_after_operations(Nums, K) ->
  .
```

### Racket:

```
(define/contract (min-or-after-operations nums k)
  (-> (listof exact-integer?) exact-integer? exact-integer?))
```

## Solutions

### C++ Solution:

```
/*
 * Problem: Minimize OR of Remaining Elements Using Operations
 * Difficulty: Hard
 * Tags: array, greedy
 *
 * 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 minOrAfterOperations(vector<int>& nums, int k) {  
}  
};
```

### Java Solution:

```
/**  
 * Problem: Minimize OR of Remaining Elements Using Operations  
 * Difficulty: Hard  
 * Tags: array, greedy  
 *  
 * 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 minOrAfterOperations(int[] nums, int k) {  
        }  
}
```

### Python3 Solution:

```
"""  
Problem: Minimize OR of Remaining Elements Using Operations  
Difficulty: Hard  
Tags: array, greedy  
  
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 minOrAfterOperations(self, nums: List[int], k: int) -> int:  
        # TODO: Implement optimized solution  
        pass
```

### Python Solution:

```

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

```

### JavaScript Solution:

```

/**
 * Problem: Minimize OR of Remaining Elements Using Operations
 * Difficulty: Hard
 * Tags: array, greedy
 *
 * 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
 * @param {number} k
 * @return {number}
 */
var minOrAfterOperations = function(nums, k) {
}
```

### TypeScript Solution:

```

/**
 * Problem: Minimize OR of Remaining Elements Using Operations
 * Difficulty: Hard
 * Tags: array, greedy
 *
 * 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 minOrAfterOperations(nums: number[], k: number): number {

```

```
};
```

### C# Solution:

```
/*
 * Problem: Minimize OR of Remaining Elements Using Operations
 * Difficulty: Hard
 * Tags: array, greedy
 *
 * 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 MinOrAfterOperations(int[] nums, int k) {
        ...
    }
}
```

### C Solution:

```
/*
 * Problem: Minimize OR of Remaining Elements Using Operations
 * Difficulty: Hard
 * Tags: array, greedy
 *
 * 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
 */

int minOrAfterOperations(int* nums, int numssize, int k) {
    ...
}
```

### Go Solution:

```
// Problem: Minimize OR of Remaining Elements Using Operations
// Difficulty: Hard
```

```

// Tags: array, greedy
//
// 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 minOrAfterOperations(nums []int, k int) int {
}

```

### Kotlin Solution:

```

class Solution {
    fun minOrAfterOperations(nums: IntArray, k: Int): Int {
        return 0
    }
}

```

### Swift Solution:

```

class Solution {
    func minOrAfterOperations(_ nums: [Int], _ k: Int) -> Int {
        return 0
    }
}

```

### Rust Solution:

```

// Problem: Minimize OR of Remaining Elements Using Operations
// Difficulty: Hard
// Tags: array, greedy
//
// 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 min_or_after_operations(nums: Vec<i32>, k: i32) -> i32 {
        return 0
    }
}

```

### Ruby Solution:

```
# @param {Integer[]} nums
# @param {Integer} k
# @return {Integer}
def min_or_after_operations(nums, k)

end
```

### PHP Solution:

```
class Solution {

    /**
     * @param Integer[] $nums
     * @param Integer $k
     * @return Integer
     */
    function minOrAfterOperations($nums, $k) {

    }
}
```

### Dart Solution:

```
class Solution {
    int minOrAfterOperations(List<int> nums, int k) {
    }
}
```

### Scala Solution:

```
object Solution {
    def minOrAfterOperations(nums: Array[Int], k: Int): Int = {
    }
}
```

### Elixir Solution:

```
defmodule Solution do
@spec min_or_after_operations(nums :: [integer], k :: integer) :: integer
def min_or_after_operations(nums, k) do

end
end
```

### Erlang Solution:

```
-spec min_or_after_operations(Nums :: [integer()], K :: integer()) ->
integer().
min_or_after_operations(Nums, K) ->
.
```

### Racket Solution:

```
(define/contract (min-or-after-operations nums k)
(-> (listof exact-integer?) exact-integer? exact-integer?))
```