Problem 683: K Empty Slots

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You have

n

bulbs in a row numbered from

1

to

n

. Initially, all the bulbs are turned off. We turn on

exactly one

bulb every day until all bulbs are on after

n

days.

You are given an array

bulbs

of length

n

where

bulbs[i] = x

means that on the

(i+1)

th

day, we will turn on the bulb at position

x

where

i

is

0-indexed

and

x

is

1-indexed.

Given an integer

k

, return

the

minimum day number

such that there exists two

turned on

bulbs that have

exactly

k

bulbs between them that are

all turned off

. If there isn't such day, return

-1

.

Example 1:

Input:

bulbs = [1,3,2], k = 1

Output:

2

Explanation:

On the first day: bulbs[0] = 1, first bulb is turned on: [1,0,0] On the second day: bulbs[1] = 3, third bulb is turned on: [1,0,1] On the third day: bulbs[2] = 2, second bulb is turned on: [1,1,1] We return 2 because on the second day, there were two on bulbs with one off bulb between them.

Example 2:

Input:

bulbs = [1,2,3], k = 1

Output:

-1

Constraints:

n == bulbs.length

1 <= n <= 2 * 10

4

1 <= bulbs[i] <= n

bulbs

is a permutation of numbers from

1

to

n

.

0 <= k <= 2 * 10

4

Code Snippets

C++:

```
class Solution {
public:
int kEmptySlots(vector<int>& bulbs, int k) {

}
};
```

Java:

```
class Solution {
public int kEmptySlots(int[] bulbs, int k) {

}
}
```

Python3:

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

Python:

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

JavaScript:

```
/**
* @param {number[]} bulbs
* @param {number} k
* @return {number}
*/
var kEmptySlots = function(bulbs, k) {

};
```

TypeScript:

```
function kEmptySlots(bulbs: number[], k: number): number {

};
```

C#:

```
public class Solution {
public int KEmptySlots(int[] bulbs, int k) {

}
}
```

C:

```
int kEmptySlots(int* bulbs, int bulbsSize, int k) {

}
```

Go:

```
func kEmptySlots(bulbs []int, k int) int {

}
```

Kotlin:

```
class Solution {
fun kEmptySlots(bulbs: IntArray, k: Int): Int {

}
}
```

Swift:

```
class Solution {
func kEmptySlots(_ bulbs: [Int], _ k: Int) -> Int {

}

}
```

Rust:

```
impl Solution {
pub fn k_empty_slots(bulbs: Vec<i32>, k: i32) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} bulbs
# @param {Integer} k
# @return {Integer}
def k_empty_slots(bulbs, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $bulbs
* @param Integer $k
* @return Integer
*/
function kEmptySlots($bulbs, $k) {

}
}
```

Dart:

```
class Solution {
int kEmptySlots(List<int> bulbs, int k) {

}
}
```

Scala:

```
object Solution {
def kEmptySlots(bulbs: Array[Int], k: Int): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec k_empty_slots(bulbs :: [integer], k :: integer) :: integer
def k_empty_slots(bulbs, k) do

end
end
```

Erlang:

```
-spec k_empty_slots(Bulbs :: [integer()], K :: integer()) -> integer().
k_empty_slots(Bulbs, K) ->
.
```

Racket:

```
(define/contract (k-empty-slots bulbs k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: K Empty Slots
* Difficulty: Hard
* Tags: array, tree, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/
```

```
class Solution {
public:
int kEmptySlots(vector<int>& bulbs, int k) {

}
};
```

Java Solution:

```
/**
* Problem: K Empty Slots
* Difficulty: Hard
* Tags: array, tree, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public int kEmptySlots(int[] bulbs, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: K Empty Slots
Difficulty: Hard
Tags: array, tree, queue, heap

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: K Empty Slots
* Difficulty: Hard
* Tags: array, tree, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* @param {number[]} bulbs
* @param {number} k
* @return {number}
*/
var kEmptySlots = function(bulbs, k) {

};
```

TypeScript Solution:

```
/**
* Problem: K Empty Slots
* Difficulty: Hard
* Tags: array, tree, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/
```

```
function kEmptySlots(bulbs: number[], k: number): number {

};
```

C# Solution:

```
/*
* Problem: K Empty Slots
* Difficulty: Hard
* Tags: array, tree, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

public class Solution {
public int KEmptySlots(int[] bulbs, int k) {

}
}
```

C Solution:

```
/*
* Problem: K Empty Slots
* Difficulty: Hard
* Tags: array, tree, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

int kEmptySlots(int* bulbs, int bulbsSize, int k) {

}
```

Go Solution:

```
// Problem: K Empty Slots
// Difficulty: Hard
// Tags: array, tree, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

func kEmptySlots(bulbs []int, k int) int {

}
```

Kotlin Solution:

```
class Solution {
fun kEmptySlots(bulbs: IntArray, k: Int): Int {

}
}
```

Swift Solution:

```
class Solution {
func kEmptySlots(_ bulbs: [Int], _ k: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: K Empty Slots
// Difficulty: Hard
// Tags: array, tree, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

impl Solution {
pub fn k_empty_slots(bulbs: Vec<i32>, k: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} bulbs
# @param {Integer} k
# @return {Integer}
def k_empty_slots(bulbs, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $bulbs
* @param Integer $k
* @return Integer
*/
function kEmptySlots($bulbs, $k) {

}
}
```

Dart Solution:

```
class Solution {
int kEmptySlots(List<int> bulbs, int k) {

}
}
```

Scala Solution:

```
object Solution {
def kEmptySlots(bulbs: Array[Int], k: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec k_empty_slots(bulbs :: [integer], k :: integer) :: integer
def k_empty_slots(bulbs, k) do

end
end
```

Erlang Solution:

```
-spec k_empty_slots(Bulbs :: [integer()], K :: integer()) -> integer().
k_empty_slots(Bulbs, K) ->
.
```

Racket Solution:

```
(define/contract (k-empty-slots bulbs k)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```