Problem 1826: Faulty Sensor

Problem Information

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

Problem Description

An experiment is being conducted in a lab. To ensure accuracy, there are

two

sensors collecting data simultaneously. You are given two arrays

sensor1

and

sensor2

, where

sensor1[i]

and

sensor2[i]

are the

i

th

data points collected by the two sensors.

However, this type of sensor has a chance of being defective, which causes

exactly one

data point to be dropped. After the data is dropped, all the data points to the

right

of the dropped data are

shifted

one place to the left, and the last data point is replaced with some

random value

. It is guaranteed that this random value will

not

be equal to the dropped value.

For example, if the correct data is

[1,2,

3

,4,5]

and

3

is dropped, the sensor could return

[1,2,4,5,

7

]

(the last position can be

any

value, not just

7

).

We know that there is a defect in

at most one

of the sensors. Return

the sensor number (

1

or

2

) with the defect. If there is

no defect

in either sensor or if it is

impossible

to determine the defective sensor, return

-1

.

Example 1:

Input:

sensor1 = [2,3,4,5], sensor2 = [2,1,3,4]

Output:

1

Explanation:

Sensor 2 has the correct values. The second data point from sensor 2 is dropped, and the last value of sensor 1 is replaced by a 5.

Example 2:

Input:

sensor1 = [2,2,2,2,2], sensor2 = [2,2,2,2,5]

Output:

-1

Explanation:

It is impossible to determine which sensor has a defect. Dropping the last value for either sensor could produce the output for the other sensor.

Example 3:

Input:

sensor1 = [2,3,2,2,3,2], sensor2 = [2,3,2,3,2,7]

Output:

2

Explanation:

Sensor 1 has the correct values. The fourth data point from sensor 1 is dropped, and the last value of sensor 1 is replaced by a 7.

Constraints:

sensor1.length == sensor2.length

1 <= sensor1.length <= 100

1 <= sensor1[i], sensor2[i] <= 100

Code Snippets

C++:

```
class Solution {
public:
int badSensor(vector<int>& sensor1, vector<int>& sensor2) {

}
};
```

Java:

```
class Solution {
public int badSensor(int[] sensor1, int[] sensor2) {

}
}
```

Python3:

```
class Solution:
def badSensor(self, sensor1: List[int], sensor2: List[int]) -> int:
```

Python:

```
class Solution(object):
def badSensor(self, sensor1, sensor2):
"""
:type sensor1: List[int]
:type sensor2: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} sensor1
* @param {number[]} sensor2
* @return {number}
*/
var badSensor = function(sensor1, sensor2) {

};
```

TypeScript:

```
function badSensor(sensor1: number[], sensor2: number[]): number {

};
```

C#:

```
public class Solution {
public int BadSensor(int[] sensor1, int[] sensor2) {

}
}
```

C:

```
int badSensor(int* sensor1, int sensor1Size, int* sensor2, int sensor2Size) {

}
```

Go:

```
func badSensor(sensor1 []int, sensor2 []int) int {

}
```

Kotlin:

```
class Solution {
fun badSensor(sensor1: IntArray, sensor2: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func badSensor(_ sensor1: [Int], _ sensor2: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn bad_sensor(sensor1: Vec<i32>, sensor2: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} sensor1
# @param {Integer[]} sensor2
# @return {Integer}
def bad_sensor(sensor1, sensor2)

end
```

PHP:

```
class Solution {
```

```
/**
* @param Integer[] $sensor1
* @param Integer[] $sensor2
* @return Integer
*/
function badSensor($sensor1, $sensor2) {

}
}
```

Dart:

```
class Solution {
int badSensor(List<int> sensor1, List<int> sensor2) {

}
}
```

Scala:

```
object Solution {
def badSensor(sensor1: Array[Int], sensor2: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec bad_sensor(sensor1 :: [integer], sensor2 :: [integer]) :: integer
def bad_sensor(sensor1, sensor2) do

end
end
```

Erlang:

```
-spec bad_sensor(Sensor1 :: [integer()], Sensor2 :: [integer()]) ->
integer().
bad_sensor(Sensor1, Sensor2) ->
.
```

Racket:

```
(define/contract (bad-sensor sensor1 sensor2)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Faulty Sensor
* Difficulty: Easy
* Tags: array
*
* 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 badSensor(vector<int>& sensor1, vector<int>& sensor2) {

}
};
```

Java Solution:

```
/**
* Problem: Faulty Sensor
* Difficulty: Easy
* Tags: array
*
* 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 badSensor(int[] sensor1, int[] sensor2) {
```

```
}
}
```

Python3 Solution:

```
"""
Problem: Faulty Sensor
Difficulty: Easy
Tags: array

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 badSensor(self, sensor1: List[int], sensor2: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def badSensor(self, sensor1, sensor2):
"""
:type sensor1: List[int]
:type sensor2: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Faulty Sensor
* Difficulty: Easy
* Tags: array
*
* 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[]} sensor1
* @param {number[]} sensor2
* @return {number}
*/
var badSensor = function(sensor1, sensor2) {

};
```

TypeScript Solution:

```
/**
* Problem: Faulty Sensor
* Difficulty: Easy
* Tags: array
*
* 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 badSensor(sensor1: number[], sensor2: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Faulty Sensor
* Difficulty: Easy
* Tags: array
*
* 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 BadSensor(int[] sensor1, int[] sensor2) {
```

```
}
}
```

C Solution:

```
/*
* Problem: Faulty Sensor
* Difficulty: Easy
* Tags: array
*
* 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 badSensor(int* sensor1, int sensor1Size, int* sensor2, int sensor2Size) {

}
```

Go Solution:

```
// Problem: Faulty Sensor
// Difficulty: Easy
// Tags: array
//
// 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 badSensor(sensor1 []int, sensor2 []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun badSensor(sensor1: IntArray, sensor2: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func badSensor(_ sensor1: [Int], _ sensor2: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Faulty Sensor
// Difficulty: Easy
// Tags: array
//
// 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 bad_sensor(sensor1: Vec<i32>, sensor2: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} sensor1
# @param {Integer[]} sensor2
# @return {Integer}
def bad_sensor(sensor1, sensor2)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $sensor1
* @param Integer[] $sensor2
* @return Integer
*/
```

```
function badSensor($sensor1, $sensor2) {

}
}
```

Dart Solution:

```
class Solution {
int badSensor(List<int> sensor1, List<int> sensor2) {

}
}
```

Scala Solution:

```
object Solution {
def badSensor(sensor1: Array[Int], sensor2: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec bad_sensor(sensor1 :: [integer], sensor2 :: [integer]) :: integer
def bad_sensor(sensor1, sensor2) do

end
end
```

Erlang Solution:

```
-spec bad_sensor(Sensor1 :: [integer()], Sensor2 :: [integer()]) ->
integer().
bad_sensor(Sensor1, Sensor2) ->
.
```

Racket Solution:

```
(define/contract (bad-sensor sensor1 sensor2)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
```

```
)
```