1826. Faulty Sensor

Two Pointers

Problem Description

a possibility that one of the sensors is defective. A defective sensor would skip exactly one data point. Once this happens, all subsequent data points shift one position to the left, and a random value that is distinct from the dropped value is placed at the end. Your task is to determine which sensor, if any, is defective. If sensor1[i] doesn't match sensor2[i], clearly one of the sensors has missed a data point. If no discrepancy is found or it's impossible to tell which sensor is defective, you should return -1. A key point of consideration is that any defect only happens in at most one sensor, never in both. To solve this, we need to analyze the sequences and look for anomalies that indicate the occurrence of a defect. Intuition

In this lab experiment, we have two sensors (sensor1 and sensor2) collecting data points simultaneously. Unfortunately, there is

The intuition behind the solution is to compare the data sequences item by item. Since the data is shifted after the point of defect, if two corresponding values are different, this indicates a potential defect. We can iterate through the sensor data until we find the first pair of unequal readings. Once this happens, we continue to compare subsequent values, but this time offset by one position. The logic behind these comparisons is as follows: If the rest of sensor1 matches the shifted sequence of sensor2, this suggests sensor2 dropped the data point. Conversely, if sensor1's shifted sequence matches the rest of sensor2, this suggests sensor1 is the defective one. If the sequences continue to match, we cannot determine which sensor is defective. In code, this is implemented by iterating twice, where the second iteration includes the offset to check the rest of the data sequences. Accordingly, we return 1 if sensor1 is defective, 2 if sensor2 is defective, or -1 if the defective sensor cannot be determined.

Solution Approach

The algorithm takes advantage of the fact that after the defect occurs, all subsequent data is shifted left by one, and the last

data value is arbitrary. The approach can be broken down into the following steps:

Initial Comparison: A while loop runs as long as the index i is less than the length of the sensor arrays minus one. We ignore the last values because we know they are unreliable after a data drop has occurred. Inside this loop, we compare sensor1[i]

to sensor2[i] for each index. The loop breaks on the first inequality, which is the possible point of defect. Check Remaining Values with Offset: After a potential defect point is found, a second while loop initiates. It has two

conditions that are checked at each index: o If sensor1[i + 1] is not equal to sensor2[i], this implies that sensor2 may have dropped a data point because the remaining elements of sensor1 match the shifted sequence of sensor2.

o If sensor1[i] is not equal to sensor2[i + 1], this implies that sensor1 may have dropped a data point because the remaining elements of

- sensor2 match the shifted sequence of sensor1. Return the Defective Sensor: Depending on which condition is met first and consistently after the potential defect point, the solution can identify which sensor is defective:
- return 1 if sensor1's remaining values match sensor2's shifted sequence, indicating that sensor1 is the defective one. • return 2 if sensor2's remaining values match sensor1's shifted sequence, indicating that sensor2 is the defective one. ∘ If neither condition matches or the initial comparison loop runs to the end without finding a discrepancy, return -1 because it's not
- possible to determine which sensor, if any, is defective. By separating the detection of the defect point and the identification of the defective sensor into two loops, the implementation
- simplifies the logic for detecting the defect and handles the edge cases where defect detection is impossible. Here is the key portion of the Python code reflecting this logic:
- i += 1# Check remaining values with offset sensor1Defect = sensor2Defect = False

while i < n - 1 and sensor1[i] == sensor2[i]:</pre>

if sensor1[j + 1] != sensor2[j]:

... assuming the initial part of the Solution class is defined ...

Initial comparison to find the potential defect point

i, n = 0, len(sensor1)

for j in range(i, n - 1):

badSensor(self, sensor1: List[int], sensor2: List[int]) -> int:

```
sensor1Defect = True
         if sensor1[i] != sensor2[j + 1]:
             sensor2Defect = True
    # Decide and return which sensor is defective, if possible
    if sensor1Defect and not sensor2Defect:
         return 2
    elif sensor2Defect and not sensor1Defect:
         return 1
    else:
         return -1
  This solution is easy to understand and has a linear time complexity of O(n), where n is the number of data points collected by
  the sensors.
Example Walkthrough
  Let's walk through a small example to illustrate the solution approach.
  Suppose we have the following sensor data:
 • sensor1: [1, 2, 3, 4, 5, 6]
```

 sensor1[0] is 1 and sensor2[0] is also 1. No discrepancy. sensor1[1] is 2 and sensor2[1] is also 2. No discrepancy. sensor1[2] is 3 but sensor2[2] is 4. We found a discrepancy at index 2. Check Remaining Values with Offset: We now check the remaining values considering an offset beginning from the point of

3. Return the Defective Sensor: Since the remaining values after the discrepancy align when we offset sensor1 but not when we offset sensor2,

• For sensor1, we compare sensor2[2] with sensor1[3]. They match (both are 4). • We see that subsequent pairs also match when sensor1 is offset by one: sensor2[3] with sensor1[4] (both are 5) and sensor2[4] with

discrepancy.

sensor1[5] (both are 6).

Now let's check the other way:

Solution Implementation

index += 1

while index < length - 1:

return -1

elif mismatch_sensor2:

return 1

index, length = 0, len(sensor1)

from typing import List

Python

class Solution:

we can conclude that the defective sensor is sensor1.

• sensor2: [1, 2, 4, 5, 6, 7]

• For sensor2, we compare sensor1[2] with sensor2[3]. They do not match (sensor1[2] is 3 and sensor2[3] is 5). • We immediately know that sensor2 cannot be the defective one because the off-setting does not align the sequences.

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

Initialize index and get the length of the sensor arrays

mismatch_sensor1, mismatch_sensor2 = False, False

if sensor1[index + 1] != sensor2[index]:

if mismatch sensor1 and mismatch_sensor2:

If only sensor2 had mismatches, sensor1 is bad

Initial Comparison: We compare the elements at each corresponding index:

- Based on this example, the badSensor function would return 1, indicating that sensor1 is defective.
- # Find the first mismatch in the sensor readings while index < length - 1: if sensor1[index] != sensor2[index]: break

mismatch sensor1 = True # Check if the rest of sensor2 matches with sensor1 shifted once if sensor1[index] != sensor2[index + 1]: mismatch sensor2 = True

If both sensors have mismatches after shifting, we can't determine the bad one

Variables to track whether the mismatch pattern is consistent with one sensor failing

Continue checking for mismatches and determine which sensor, if any, is bad

Check if the rest of sensor1 matches with sensor2 shifted once

```
index += 1
# If only sensor1 had mismatches, sensor2 is bad
if mismatch sensor1:
    return 2
```

```
# If there were no mismatches, we cannot determine the bad sensor
        else:
            return -1
# Note: The method name 'badSensor' remains unchanged as requested.
Java
class Solution {
    /**
     * Determines which sensor, if any, is faulty.
     * @param sensorl Array of readings from the first sensor.
     * @param sensor2 Array of readings from the second sensor.
     * @return The number of the faulty sensor (1 or 2), or -1 if it cannot be determined.
     */
    public int badSensor(int[] sensor1, int[] sensor2) {
        int index = 0; // Index to iterate through sensor readings.
        int length = sensor1.length; // Assuming both sensors have the same length of readings.
        // Move through the readings while they are the same for both sensors.
        for (; index < length - 1 && sensor1[index] == sensor2[index]; ++index) {</pre>
            // No operation, just incrementing index.
        // Continue examining the readings after the first discrepancy.
        for (; index < length - 1; ++index) {</pre>
            // If the reading from sensor1 is not equal to the previous reading of sensor2, sensor1 is bad.
            if (sensor1[index + 1] != sensor2[index]) {
                return 1; // sensor1 is faulty.
            // If the reading from sensor2 is not equal to the previous reading of sensor1, sensor2 is bad.
            if (sensor1[index] != sensor2[index + 1]) {
                return 2; // sensor2 is faulty.
        // If neither sensor is conclusively found to be faulty, return -1.
        return -1;
C++
class Solution {
```

};

TypeScript

public:

// Function to determine which sensor is bad

for (; index < size - 1; ++index) {</pre>

if (index == size - 1) {

return -1;

return -1;

let index = 0:

int badSensor(vector<int>& sensor1, vector<int>& sensor2) {

// value in sensor2, then sensor1 is likely bad

if (sensor1[index + 1] != sensor2[index]) {

if (sensor1[index] != sensor2[index + 1]) {

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

// Initialize the index to compare both sensor arrays

// Get the length of the sensor arrays

return 1; // Sensor 1 is bad

// sensor2. then sensor2 is likely bad

return 2; // Sensor 2 is bad

int index = 0; // This will store the first index where the two sensors differ

int size = sensor1.size(); // Assuming both sensors have the same size readings

for (; index < size - 1 && sensor1[index] == sensor2[index]; ++index) {}</pre>

// If none of the sensors is conclusively found to be bad, return -1.

// Iterate until sensors readings are the same or we reach the second to last element

// If the loop had gone all the way to the second to last element without any difference found

// then there is no bad sensor or it is not possible to determine it based on the last reading.

// Iterate through the rest of the sensor readings starting from the point of discrepancy

// This also covers the case when the discrepancy is found at the last pair of readings

// If we find a discrepancy where the next value in sensor1 doesn't match the current

// If the discrepancy is such that the current value in sensor1 doesn't match the next value in

```
const length = sensor1.length;
    // Find the first mismatch between the two sensor readings
    while (index < length - 1 && sensor1[index] === sensor2[index]) {</pre>
        index++;
    // If no mismatch was found before the last element, return -1
    if (index === length - 1) {
        return -1;
    // Check if sensor1 could be the bad sensor
    if (isSensorBad(sensor1, sensor2, index)) {
        return 1;
    // Check if sensor2 could be the bad sensor
    if (isSensorBad(sensor2, sensor1, index)) {
        return 2;
    // If neither sensor is confirmed bad, return -1
   return -1;
function isSensorBad(sensorA: number[], sensorB: number[], startIndex: number): boolean {
    // Compare the readings between sensorA and sensorB from the startIndex
    for (let i = startIndex; i < sensorA.length - 1; i++) {</pre>
        if (sensorA[i + 1] !== sensorB[i]) {
            return true;
   // If all subsequent readings match after the startIndex, this sensor is not bad
    return false;
from typing import List
class Solution:
   def badSensor(self, sensor1: List[int], sensor2: List[int]) -> int:
       # Initialize index and get the length of the sensor arrays
        index, length = 0, len(sensor1)
       # Find the first mismatch in the sensor readings
       while index < length - 1:</pre>
            if sensor1[index] != sensor2[index]:
                break
            index += 1
       # Variables to track whether the mismatch pattern is consistent with one sensor failing
       mismatch_sensor1, mismatch_sensor2 = False, False
       # Continue checking for mismatches and determine which sensor, if any, is bad
        while index < length - 1:
           # Check if the rest of sensor1 matches with sensor2 shifted once
```

If there were no mismatches, we cannot determine the bad sensor else: return -1 # Note: The method name 'badSensor' remains unchanged as requested.

Time and Space Complexity

return -1

index += 1

if mismatch sensor1:

elif mismatch_sensor2:

return 2

return 1

if sensor1[index + 1] != sensor2[index]:

if sensor1[index] != sensor2[index + 1]:

if mismatch sensor1 and mismatch_sensor2:

If only sensor1 had mismatches, sensor2 is bad

If only sensor2 had mismatches, sensor1 is bad

Check if the rest of sensor2 matches with sensor1 shifted once

the input size. No additional data structures are used that grow with input size.

If both sensors have mismatches after shifting, we can't determine the bad one

mismatch sensor1 = True

mismatch sensor2 = True

The given Python code has two main loops. The first loop increments i until it finds the first mismatch between sensor1 and

sensor2 or reaches the second to last index. The worst-case scenario for this loop is 0(n - 1) if the mismatch is at the last compared element or there's no mismatch. The second loop can also iterate up to 0(n - 1) in the worst case (when checking for mismatches and deciding which sensor is bad). Therefore, the worst-case time complexity is 0(n - 1) + 0(n - 1) which simplifies to O(n) as we drop constants for Big O notation. **Space Complexity**

The space complexity of the solution is 0(1). It uses a constant amount of extra space for the variables i and n, regardless of

Time Complexity