Problem 2534: Time Taken to Cross the Door

Problem Information

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

Problem Description

There are

n

persons numbered from

0

to

n - 1

and a door. Each person can enter or exit through the door once, taking one second.

You are given a

non-decreasing

integer array

arrival

of size

n

, where

arrival[i]

is the arrival time of the

i

th

person at the door. You are also given an array

state

of size

n

, where

state[i]

is

0

if person

i

wants to enter through the door or

1

if they want to exit through the door.

If two or more persons want to use the door at the

same

time, they follow the following rules:

If the door was

not

used in the previous second, then the person who wants to

exit

goes first.

If the door was used in the previous second for

entering

, the person who wants to enter goes first.

If the door was used in the previous second for

exiting

, the person who wants to

exit

goes first.

If multiple persons want to go in the same direction, the person with the

smallest

index goes first.

Return

an array

answer

of size

n

where

answer[i]

is the second at which the

i

th

person crosses the door

.

Note

that:

Only one person can cross the door at each second.

A person may arrive at the door and wait without entering or exiting to follow the mentioned rules.

Example 1:

Input:

arrival = [0,1,1,2,4], state = [0,1,0,0,1]

Output:

[0,3,1,2,4]

Explanation:

At each second we have the following: - At t = 0: Person 0 is the only one who wants to enter, so they just enter through the door. - At t = 1: Person 1 wants to exit, and person 2 wants to enter. Since the door was used the previous second for entering, person 2 enters. - At t = 2: Person 1 still wants to exit, and person 3 wants to enter. Since the door was used the previous second for entering, person 3 enters. - At t = 3: Person 1 is the only one who wants to exit, so they just exit through the door. - At t = 4: Person 4 is the only one who wants to exit, so they just exit through the door.

Example 2:

Input:

arrival = [0,0,0], state = [1,0,1]

Output:

[0,2,1]

Explanation:

At each second we have the following: - At t = 0: Person 1 wants to enter while persons 0 and 2 want to exit. Since the door was not used in the previous second, the persons who want to exit get to go first. Since person 0 has a smaller index, they exit first. - At t = 1: Person 1 wants to enter, and person 2 wants to exit. Since the door was used in the previous second for exiting, person 2 exits. - At t = 2: Person 1 is the only one who wants to enter, so they just enter through the door.

Constraints:

n == arrival.length == state.length

1 <= n <= 10

5

0 <= arrival[i] <= n

arrival

is sorted in

non-decreasing

order.

state[i]

is either

0

or

1

.

Code Snippets

C++:

```
class Solution {
public:
vector<int> timeTaken(vector<int>& arrival, vector<int>& state) {

}
};
```

Java:

```
class Solution {
public int[] timeTaken(int[] arrival, int[] state) {

}
}
```

Python3:

```
class Solution:
def timeTaken(self, arrival: List[int], state: List[int]) -> List[int]:
```

Python:

```
class Solution(object):
def timeTaken(self, arrival, state):
"""
:type arrival: List[int]
:type state: List[int]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[]} arrival
* @param {number[]} state
* @return {number[]}
*/
var timeTaken = function(arrival, state) {

};
```

TypeScript:

```
function timeTaken(arrival: number[], state: number[]): number[] {

};
```

C#:

```
public class Solution {
public int[] TimeTaken(int[] arrival, int[] state) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
```

```
int* timeTaken(int* arrival, int arrivalSize, int* state, int stateSize, int*
returnSize) {

}
```

Go:

```
func timeTaken(arrival []int, state []int) []int {

}
```

Kotlin:

```
class Solution {
fun timeTaken(arrival: IntArray, state: IntArray): IntArray {

}
}
```

Swift:

```
class Solution {
func timeTaken(_ arrival: [Int], _ state: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn time_taken(arrival: Vec<i32>, state: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer[]} arrival
# @param {Integer[]} state
# @return {Integer[]}
def time_taken(arrival, state)
```

```
end
```

PHP:

```
class Solution {

/**
* @param Integer[] $arrival
* @param Integer[] $state
* @return Integer[]
*/
function timeTaken($arrival, $state) {

}
}
```

Dart:

```
class Solution {
List<int> timeTaken(List<int> arrival, List<int> state) {

}
}
```

Scala:

```
object Solution {
def timeTaken(arrival: Array[Int], state: Array[Int]): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec time_taken(arrival :: [integer], state :: [integer]) :: [integer]
def time_taken(arrival, state) do

end
end
```

Erlang:

```
-spec time_taken(Arrival :: [integer()], State :: [integer()]) ->
[integer()].
time_taken(Arrival, State) ->
.
```

Racket:

```
(define/contract (time-taken arrival state)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Time Taken to Cross the Door
* Difficulty: Hard
* Tags: array, sort, queue
*
* 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> timeTaken(vector<int>& arrival, vector<int>& state) {

}
};
```

Java Solution:

```
/**
* Problem: Time Taken to Cross the Door
* Difficulty: Hard
* Tags: array, sort, queue
*
* 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[] timeTaken(int[] arrival, int[] state) {

}
}
```

Python3 Solution:

```
"""
Problem: Time Taken to Cross the Door
Difficulty: Hard
Tags: array, sort, queue

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 timeTaken(self, arrival: List[int], state: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def timeTaken(self, arrival, state):
"""
:type arrival: List[int]
:type state: List[int]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Time Taken to Cross the Door
* Difficulty: Hard
```

```
* Tags: array, sort, queue
*
* 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[]} arrival
* @param {number[]} state
* @return {number[]}
*/
var timeTaken = function(arrival, state) {

};
```

TypeScript Solution:

```
/**
* Problem: Time Taken to Cross the Door
* Difficulty: Hard
* Tags: array, sort, queue
*
* 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 timeTaken(arrival: number[], state: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Time Taken to Cross the Door
* Difficulty: Hard
* Tags: array, sort, queue
*
* 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[] TimeTaken(int[] arrival, int[] state) {

}
}
```

C Solution:

```
/*
* Problem: Time Taken to Cross the Door
* Difficulty: Hard
* Tags: array, sort, queue
*
* 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* timeTaken(int* arrival, int arrivalSize, int* state, int stateSize, int*
returnSize) {

}
```

Go Solution:

```
// Problem: Time Taken to Cross the Door
// Difficulty: Hard
// Tags: array, sort, queue
//
// 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 timeTaken(arrival []int, state []int) []int {
```

```
}
```

Kotlin Solution:

```
class Solution {
fun timeTaken(arrival: IntArray, state: IntArray): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func timeTaken(_ arrival: [Int], _ state: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Time Taken to Cross the Door
// Difficulty: Hard
// Tags: array, sort, queue
//
// 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 time_taken(arrival: Vec<i32>, state: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} arrival
# @param {Integer[]} state
# @return {Integer[]}
def time_taken(arrival, state)
```

```
end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $arrival
* @param Integer[] $state
* @return Integer[]
*/
function timeTaken($arrival, $state) {

}
}
```

Dart Solution:

```
class Solution {
List<int> timeTaken(List<int> arrival, List<int> state) {

}
}
```

Scala Solution:

```
object Solution {
def timeTaken(arrival: Array[Int], state: Array[Int]): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec time_taken(arrival :: [integer], state :: [integer]) :: [integer]
def time_taken(arrival, state) do

end
end
```

Erlang Solution:

```
-spec time_taken(Arrival :: [integer()], State :: [integer()]) ->
[integer()].
time_taken(Arrival, State) ->
.
```

Racket Solution:

```
(define/contract (time-taken arrival state)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?))
)
```