Problem 2751: Robot Collisions

Problem Information

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

Problem Description

There are

n

1-indexed

robots, each having a position on a line, health, and movement direction.

You are given

0-indexed

integer arrays

positions

,

healths

, and a string

directions

(

directions[i]

is either

'L'

for

left

or

'R'

for

right

). All integers in

positions

are

unique

.

All robots start moving on the line

simultaneously

at the

same speed

in their given directions. If two robots ever share the same position while moving, they will

collide

.

If two robots collide, the robot with

lower health

is

removed

from the line, and the health of the other robot

decreases

by one

. The surviving robot continues in the

same

direction it was going. If both robots have the

same

health, they are both

removed from the line.

Your task is to determine the

health

of the robots that survive the collisions, in the same

order

that the robots were given,

i.e. final health of robot 1 (if survived), final health of robot 2 (if survived), and so on. If there are no survivors, return an empty array.

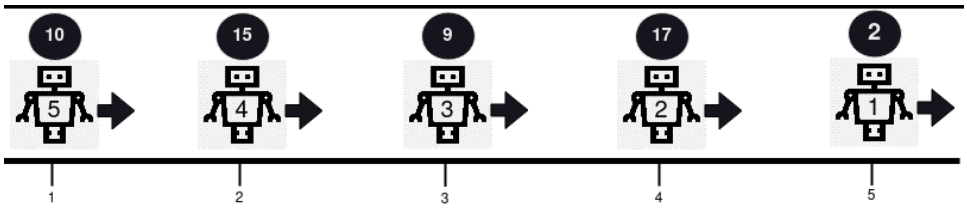
Return

an array containing the health of the remaining robots (in the order they were given in the input), after no further collisions can occur.

Note:

The positions may be unsorted.

Example 1:



Input:

positions = [5,4,3,2,1], healths = [2,17,9,15,10], directions = "RRRRR"

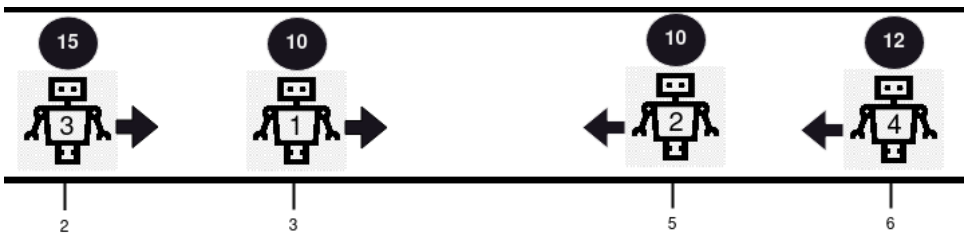
Output:

[2,17,9,15,10]

Explanation:

No collision occurs in this example, since all robots are moving in the same direction. So, the health of the robots in order from the first robot is returned, [2, 17, 9, 15, 10].

Example 2:



Input:

positions = [3,5,2,6], healths = [10,10,15,12], directions = "RLRL"

Output:

[14]

Explanation:

There are 2 collisions in this example. Firstly, robot 1 and robot 2 will collide, and since both have the same health, they will be removed from the line. Next, robot 3 and robot 4 will collide and since robot 4's health is smaller, it gets removed, and robot 3's health becomes 15 - 1 = 14. Only robot 3 remains, so we return [14].

Example 3:

Input:

positions = [1,2,5,6], healths = [10,10,11,11], directions = "RLRL"

Output:

[]

Explanation:

Robot 1 and robot 2 will collide and since both have the same health, they are both removed. Robot 3 and 4 will collide and since both have the same health, they are both removed. So, we return an empty array, [].

Constraints:

1 <= positions.length == healths.length == directions.length == n <= 10

5

1 <= positions[i], healths[i] <= 10

9

directions[i] == 'L'

or

directions[i] == 'R'

All values in

positions

are distinct

Code Snippets

C++:

```
class Solution {
public:
vector<int> survivedRobotsHealths(vector<int>& positions, vector<int>&
healths, string directions) {

}
};
```

Java:

```
class Solution {
public List<Integer> survivedRobotsHealths(int[] positions, int[] healths,
String directions) {

}
```

```
}
```

Python3:

```
class Solution:
def survivedRobotsHealths(self, positions: List[int], healths: List[int],
directions: str) -> List[int]:
```

Python:

```
class Solution(object):
def survivedRobotsHealths(self, positions, healths, directions):
"""
:type positions: List[int]
:type healths: List[int]
:type directions: str
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[]} positions
* @param {number[]} healths
* @param {string} directions
* @return {number[]}
*/
var survivedRobotsHealths = function(positions, healths, directions) {

};
```

TypeScript:

```
function survivedRobotsHealths(positions: number[], healths: number[],
directions: string): number[] {

};
```

C#:

```
public class Solution {
public IList<int> SurvivedRobotsHealths(int[] positions, int[] healths,
```

```
string directions) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* survivedRobotsHealths(int* positions, int positionsSize, int* healths,
int healthsSize, char* directions, int* returnSize) {

}
```

Go:

```
func survivedRobotsHealths(positions []int, healths []int, directions string)
[]int {

}
```

Kotlin:

```
class Solution {
fun survivedRobotsHealths(positions: IntArray, healths: IntArray, directions:
String): List<Int> {

}
}
```

Swift:

```
class Solution {
func survivedRobotsHealths(_ positions: [Int], _ healths: [Int], _
directions: String) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn survived_robots_healths(positions: Vec<i32>, healths: Vec<i32>,
directions: String) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer[]} positions
# @param {Integer[]} healths
# @param {String} directions
# @return {Integer[]}
def survived_robots_healths(positions, healths, directions)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $positions
* @param Integer[] $healths
* @param String $directions
* @return Integer[]
*/
function survivedRobotsHealths($positions, $healths, $directions) {

}
}
```

Dart:

```
class Solution {
List<int> survivedRobotsHealths(List<int> positions, List<int> healths,
String directions) {

}
}
```

Scala:

```
object Solution {
def survivedRobotsHealths(positions: Array[Int], healths: Array[Int],
directions: String): List[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec survived_robots_healths(positions :: [integer], healths :: [integer],
directions :: String.t) :: [integer]
def survived_robots_healths(positions, healths, directions) do

end
end
```

Erlang:

```
-spec survived_robots_healths(Positions :: [integer()], Healths ::
[integer()], Directions :: unicode:unicode_binary()) -> [integer()].
survived_robots_healths(Positions, Healths, Directions) ->
.
```

Racket:

```
(define/contract (survived-robots-healths positions healths directions)
(-> (listof exact-integer?) (listof exact-integer?) string? (listof
exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Robot Collisions
* Difficulty: Hard
* Tags: array, string, sort, stack
*
* 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> survivedRobotsHealths(vector<int>& positions, vector<int>&
healths, string directions) {

}
};
```

Java Solution:

```
/**
* Problem: Robot Collisions
* Difficulty: Hard
* Tags: array, string, sort, stack
*
* 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 List<Integer> survivedRobotsHealths(int[] positions, int[] healths,
String directions) {

}
}
```

Python3 Solution:

```
"""
Problem: Robot Collisions
Difficulty: Hard
Tags: array, string, sort, stack

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 survivedRobotsHealths(self, positions: List[int], healths: List[int],
directions: str) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def survivedRobotsHealths(self, positions, healths, directions):
"""
:type positions: List[int]
:type healths: List[int]
:type directions: str
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Robot Collisions
* Difficulty: Hard
* Tags: array, string, sort, stack
*
* 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[]} positions
* @param {number[]} healths
* @param {string} directions
* @return {number[]}
*/
var survivedRobotsHealths = function(positions, healths, directions) {

};
```

TypeScript Solution:

```
/**
* Problem: Robot Collisions
* Difficulty: Hard
* Tags: array, string, sort, stack
*
* 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 survivedRobotsHealths(positions: number[], healths: number[],
directions: string): number[] {

};
```

C# Solution:

```
/*
* Problem: Robot Collisions
* Difficulty: Hard
* Tags: array, string, sort, stack
*
* 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 IList<int> SurvivedRobotsHealths(int[] positions, int[] healths,
string directions) {

}
}
```

C Solution:

```
/*
* Problem: Robot Collisions
* Difficulty: Hard
* Tags: array, string, sort, stack
*
```

```
* 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* survivedRobotsHealths(int* positions, int positionsSize, int* healths,
int healthsSize, char* directions, int* returnSize) {

}
```

Go Solution:

```
// Problem: Robot Collisions
// Difficulty: Hard
// Tags: array, string, sort, stack
//
// 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 survivedRobotsHealths(positions []int, healths []int, directions string)
[]int {

}
```

Kotlin Solution:

```
class Solution {
fun survivedRobotsHealths(positions: IntArray, healths: IntArray, directions:
String): List<Int> {

}
}
```

Swift Solution:

```
class Solution {
func survivedRobotsHealths(_ positions: [Int], _ healths: [Int], _
```

```
directions: String) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Robot Collisions
// Difficulty: Hard
// Tags: array, string, sort, stack
//
// 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 survived_robots_healths(positions: Vec<i32>, healths: Vec<i32>,
directions: String) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} positions
# @param {Integer[]} healths
# @param {String} directions
# @return {Integer[]}
def survived_robots_healths(positions, healths, directions)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $positions
* @param Integer[] $healths
* @param String $directions
* @return Integer[]
*/
```

```
function survivedRobotsHealths($positions, $healths, $directions) {

}
}
```

Dart Solution:

```
class Solution {
List<int> survivedRobotsHealths(List<int> positions, List<int> healths,
String directions) {

}
}
```

Scala Solution:

```
object Solution {
def survivedRobotsHealths(positions: Array[Int], healths: Array[Int],
directions: String): List[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec survived_robots_healths(positions :: [integer], healths :: [integer],
directions :: String.t) :: [integer]
def survived_robots_healths(positions, healths, directions) do

end
end
```

Erlang Solution:

```
-spec survived_robots_healths(Positions :: [integer()], Healths ::
[integer()], Directions :: unicode:unicode_binary()) -> [integer()].
survived_robots_healths(Positions, Healths, Directions) ->
.
```

Racket Solution:

```
(define/contract (survived-robots-healths positions healths directions)
(-> (listof exact-integer?) (listof exact-integer?) string? (listof
exact-integer?))
)
```