Problem 3661: Maximum Walls Destroyed by Robots

Problem Information

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

Problem Description

There is an endless straight line populated with some robots and walls. You are given integer arrays

robots

,

distance

, and

walls

:

robots[i]

is the position of the

i

th

robot.

distance[i]

is the

maximum

distance the

i

th

robot's bullet can travel.

walls[j]

is the position of the

j

th

wall.

Every robot has

one

bullet that can either fire to the left or the right

at most

distance[i]

meters.

A bullet destroys every wall in its path that lies within its range. Robots are fixed obstacles: if a bullet hits another robot before reaching a wall, it

immediately stops

at that robot and cannot continue.

Return the

maximum

number of

unique

walls that can be destroyed by the robots.

Notes:

A wall and a robot may share the same position; the wall can be destroyed by the robot at that position.

Robots are not destroyed by bullets.

Example 1:

Input:

robots = [4], distance = [3], walls = [1,10]

Output:

1

Explanation:

robots[0] = 4

fires

left

with

distance[0] = 3

, covering

[1, 4]

and destroys

walls[0] = 1

.

Thus, the answer is 1.

Example 2:

Input:

robots = [10,2], distance = [5,1], walls = [5,2,7]

Output:

3

Explanation:

robots[0] = 10

fires

left

with

distance[0] = 5

, covering

[5, 10]

and destroys

walls[0] = 5

and

walls[2] = 7

.

robots[1] = 2

fires

left

with

distance[1] = 1

, covering

[1, 2]

and destroys

walls[1] = 2

.

Thus, the answer is 3.

Example 3:

Input:

robots = [1,2], distance = [100,1], walls = [10]

Output:

0

Explanation:

In this example, only

robots[0]

can reach the wall, but its shot to the

right

is blocked by

robots[1]

; thus the answer is 0.

Constraints:

1 <= robots.length == distance.length <= 10

5

1 <= walls.length <= 10

5

1 <= robots[i], walls[j] <= 10

9

1 <= distance[i] <= 10

5

All values in

robots

are

unique

All values in

walls

are

unique

Code Snippets

C++:

```
class Solution {
public:
int maxWalls(vector<int>& robots, vector<int>& distance, vector<int>& walls)
{

}
};
```

Java:

```
class Solution {
public int maxWalls(int[] robots, int[] distance, int[] walls) {

}
}
```

Python3:

```
class Solution:
def maxWalls(self, robots: List[int], distance: List[int], walls: List[int])
-> int:
```

Python:

```
class Solution(object):
def maxWalls(self, robots, distance, walls):
"""
:type robots: List[int]
:type distance: List[int]
:type walls: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} robots
* @param {number[]} distance
* @param {number[]} walls
* @return {number}
*/
var maxWalls = function(robots, distance, walls) {

};
```

TypeScript:

```
function maxWalls(robots: number[], distance: number[], walls: number[]):
number {

};
```

C#:

```
public class Solution {
public int MaxWalls(int[] robots, int[] distance, int[] walls) {

}
}
```

C:

```
int maxWalls(int* robots, int robotsSize, int* distance, int distanceSize,
int* walls, int wallsSize) {

}
```

Go:

```
func maxWalls(robots []int, distance []int, walls []int) int {

}
```

Kotlin:

```
class Solution {
fun maxWalls(robots: IntArray, distance: IntArray, walls: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func maxWalls(_ robots: [Int], _ distance: [Int], _ walls: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn max_walls(robots: Vec<i32>, distance: Vec<i32>, walls: Vec<i32>) ->
i32 {

}
}
```

Ruby:

```
# @param {Integer[]} robots
# @param {Integer[]} distance
# @param {Integer[]} walls
# @return {Integer}
def max_walls(robots, distance, walls)
```

```
end
```

PHP:

```
class Solution {

/**
* @param Integer[] $robots
* @param Integer[] $distance
* @param Integer[] $walls
* @return Integer
*/
function maxWalls($robots, $distance, $walls) {

}
}
```

Dart:

```
class Solution {
int maxWalls(List<int> robots, List<int> distance, List<int> walls) {

}
}
```

Scala:

```
object Solution {
def maxWalls(robots: Array[Int], distance: Array[Int], walls: Array[Int]):
Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec max_walls(robots :: [integer], distance :: [integer], walls ::
[integer]) :: integer
def max_walls(robots, distance, walls) do
```

```
end
end
```

Erlang:

```
-spec max_walls(Robots :: [integer()], Distance :: [integer()], Walls ::
[integer()]) -> integer().
max_walls(Robots, Distance, Walls) ->
.
```

Racket:

```
(define/contract (max-walls robots distance walls)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?)
exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Walls Destroyed by Robots
* Difficulty: Hard
* Tags: array, tree, dp, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
int maxWalls(vector<int>& robots, vector<int>& distance, vector<int>& walls)
{

}
};
```

Java Solution:

```
/**
* Problem: Maximum Walls Destroyed by Robots
* Difficulty: Hard
* Tags: array, tree, dp, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int maxWalls(int[] robots, int[] distance, int[] walls) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Walls Destroyed by Robots
Difficulty: Hard
Tags: array, tree, dp, sort, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def maxWalls(self, robots: List[int], distance: List[int], walls: List[int])
-> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def maxWalls(self, robots, distance, walls):
"""
:type robots: List[int]
:type distance: List[int]
:type walls: List[int]
```

```
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Maximum Walls Destroyed by Robots
* Difficulty: Hard
* Tags: array, tree, dp, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number[]} robots
* @param {number[]} distance
* @param {number[]} walls
* @return {number}
*/
var maxWalls = function(robots, distance, walls) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Walls Destroyed by Robots
* Difficulty: Hard
* Tags: array, tree, dp, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function maxWalls(robots: number[], distance: number[], walls: number[]):
number {

};
```

C# Solution:

```
/*
* Problem: Maximum Walls Destroyed by Robots
* Difficulty: Hard
* Tags: array, tree, dp, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int MaxWalls(int[] robots, int[] distance, int[] walls) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Walls Destroyed by Robots
* Difficulty: Hard
* Tags: array, tree, dp, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

int maxWalls(int* robots, int robotsSize, int* distance, int distanceSize,
int* walls, int wallsSize) {

}
```

Go Solution:

```
// Problem: Maximum Walls Destroyed by Robots
// Difficulty: Hard
// Tags: array, tree, dp, sort, search
//
// Approach: Use two pointers or sliding window technique
```

```
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func maxWalls(robots []int, distance []int, walls []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun maxWalls(robots: IntArray, distance: IntArray, walls: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func maxWalls(_ robots: [Int], _ distance: [Int], _ walls: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Maximum Walls Destroyed by Robots
// Difficulty: Hard
// Tags: array, tree, dp, sort, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn max_walls(robots: Vec<i32>, distance: Vec<i32>, walls: Vec<i32>) ->
i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} robots
# @param {Integer[]} distance
# @param {Integer[]} walls
# @return {Integer}
def max_walls(robots, distance, walls)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $robots
* @param Integer[] $distance
* @param Integer[] $walls
* @return Integer
*/
function maxWalls($robots, $distance, $walls) {

}
}
```

Dart Solution:

```
class Solution {
int maxWalls(List<int> robots, List<int> distance, List<int> walls) {

}
}
```

Scala Solution:

```
object Solution {
def maxWalls(robots: Array[Int], distance: Array[Int], walls: Array[Int]):
Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec max_walls(robots :: [integer], distance :: [integer], walls ::
[integer]) :: integer
def max_walls(robots, distance, walls) do

end
end
```

Erlang Solution:

```
-spec max_walls(Robots :: [integer()], Distance :: [integer()], Walls ::
[integer()]) -> integer().
max_walls(Robots, Distance, Walls) ->
.
```

Racket Solution:

```
(define/contract (max-walls robots distance walls)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?)
exact-integer?)
)
```