Problem 2463: Minimum Total Distance Traveled

Problem Information

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

Problem Description

There are some robots and factories on the X-axis. You are given an integer array

robot

where

robot[i]

is the position of the

i

th

robot. You are also given a 2D integer array

factory

where

factory[j] = [position

j

, limit

j

]

indicates that

position

j

is the position of the

j

th

factory and that the

j

th

factory can repair at most

limit

j

robots.

The positions of each robot are

unique

. The positions of each factory are also

unique

. Note that a robot can be

in the same position

as a factory initially.

All the robots are initially broken; they keep moving in one direction. The direction could be the negative or the positive direction of the X-axis. When a robot reaches a factory that did not reach its limit, the factory repairs the robot, and it stops moving.

At any moment

, you can set the initial direction of moving for

some

robot. Your target is to minimize the total distance traveled by all the robots.

Return

the minimum total distance traveled by all the robots

. The test cases are generated such that all the robots can be repaired.

Note that

All robots move at the same speed.

If two robots move in the same direction, they will never collide.

If two robots move in opposite directions and they meet at some point, they do not collide. They cross each other.

If a robot passes by a factory that reached its limits, it crosses it as if it does not exist.

If the robot moved from a position

x

to a position

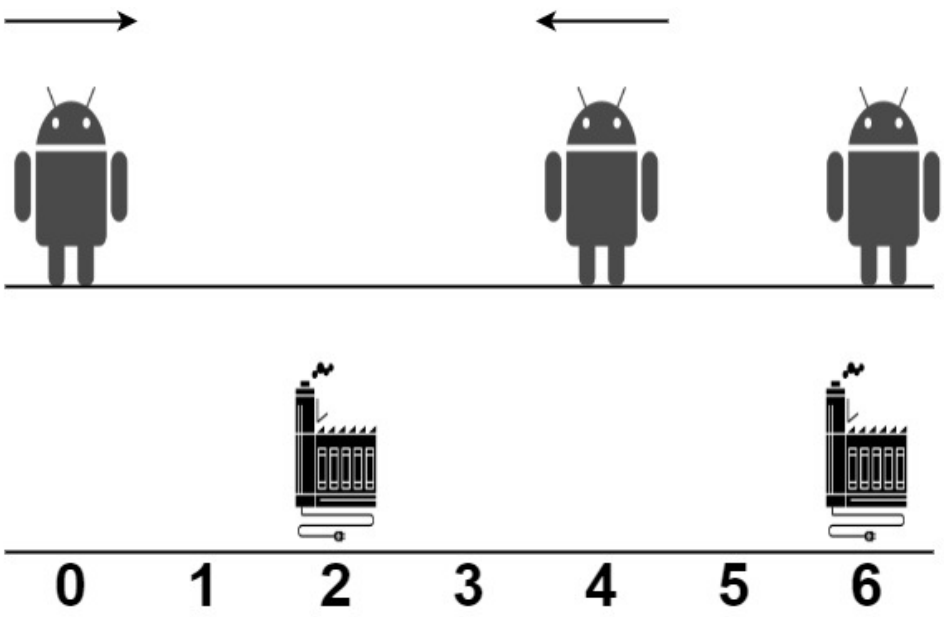
y

, the distance it moved is

|y - x|

.

Example 1:



Input:

robot = [0,4,6], factory = [[2,2],[6,2]]

Output:

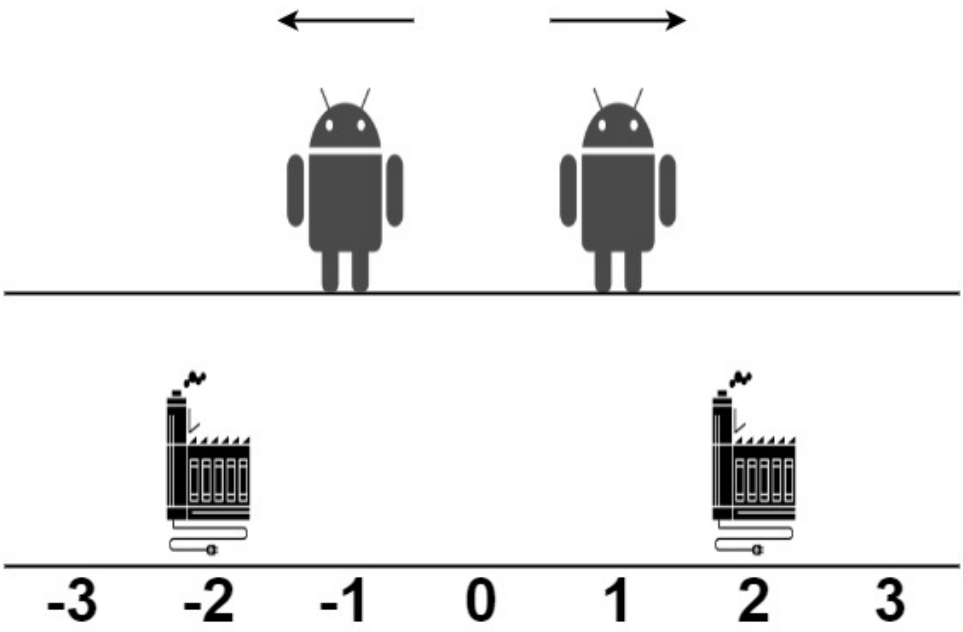
4

Explanation:

As shown in the figure: - The first robot at position 0 moves in the positive direction. It will be repaired at the first factory. - The second robot at position 4 moves in the negative direction. It will be repaired at the first factory. - The third robot at position 6 will be repaired at the second

factory. It does not need to move. The limit of the first factory is 2, and it fixed 2 robots. The limit of the second factory is 2, and it fixed 1 robot. The total distance is |2 - 0| + |2 - 4| + |6 - 6| = 4. It can be shown that we cannot achieve a better total distance than 4.

Example 2:



Input:

robot = [1,-1], factory = [[-2,1],[2,1]]

Output:

2

Explanation:

As shown in the figure: - The first robot at position 1 moves in the positive direction. It will be repaired at the second factory. - The second robot at position -1 moves in the negative direction. It will be repaired at the first factory. The limit of the first factory is 1, and it fixed 1 robot. The limit of the second factory is 1, and it fixed 1 robot. The total distance is |2 - 1| + |(-2) - (-1)| = 2. It can be shown that we cannot achieve a better total distance than 2.

Constraints:

1 <= robot.length, factory.length <= 100

factory[j].length == 2

-10

9

<= robot[i], position

j

<= 10

9

0 <= limit

j

<= robot.length

The input will be generated such that it is always possible to repair every robot.

Code Snippets

C++:

```
class Solution {
public:
long long minimumTotalDistance(vector<int>& robot, vector<vector<int>>&
factory) {

}
};
```

Java:

```
class Solution {
public long minimumTotalDistance(List<Integer> robot, int[][] factory) {
```

```
}
}
```

Python3:

```
class Solution:
def minimumTotalDistance(self, robot: List[int], factory: List[List[int]]) ->
int:
```

Python:

```
class Solution(object):
def minimumTotalDistance(self, robot, factory):
"""
:type robot: List[int]
:type factory: List[List[int]]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} robot
* @param {number[][]} factory
* @return {number}
*/
var minimumTotalDistance = function(robot, factory) {

};
```

TypeScript:

```
function minimumTotalDistance(robot: number[], factory: number[][]): number {

};
```

C#:

```
public class Solution {
public long MinimumTotalDistance(IList<int> robot, int[][] factory) {

}
```

```
}
```

C:

```
long long minimumTotalDistance(int* robot, int robotSize, int** factory, int
factorySize, int* factoryColSize) {

}
```

Go:

```
func minimumTotalDistance(robot []int, factory [][]int) int64 {

}
```

Kotlin:

```
class Solution {
fun minimumTotalDistance(robot: List<Int>, factory: Array<IntArray>): Long {

}
}
```

Swift:

```
class Solution {
func minimumTotalDistance(_ robot: [Int], _ factory: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn minimum_total_distance(robot: Vec<i32>, factory: Vec<Vec<i32>>) -> i64
{

}
}
```

Ruby:

```
# @param {Integer[]} robot
# @param {Integer[][]} factory
# @return {Integer}
def minimum_total_distance(robot, factory)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $robot
* @param Integer[][] $factory
* @return Integer
*/
function minimumTotalDistance($robot, $factory) {

}
}
```

Dart:

```
class Solution {
int minimumTotalDistance(List<int> robot, List<List<int>> factory) {

}
}
```

Scala:

```
object Solution {
def minimumTotalDistance(robot: List[Int], factory: Array[Array[Int]]): Long
= {

}
}
```

Elixir:

```
defmodule Solution do
@spec minimum_total_distance(robot :: [integer], factory :: [[integer]]) ::
integer
```

```
def minimum_total_distance(robot, factory) do

end
end
```

Erlang:

```
-spec minimum_total_distance(Robot :: [integer()], Factory :: [[integer()]])
-> integer().
minimum_total_distance(Robot, Factory) ->
.
```

Racket:

```
(define/contract (minimum-total-distance robot factory)
(-> (listof exact-integer?) (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Total Distance Traveled
* Difficulty: Hard
* Tags: array, dp, sort
*
* 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:
long long minimumTotalDistance(vector<int>& robot, vector<vector<int>>&
factory) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Total Distance Traveled
* Difficulty: Hard
* Tags: array, dp, sort
*
* 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 long minimumTotalDistance(List<Integer> robot, int[][] factory) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Total Distance Traveled
Difficulty: Hard
Tags: array, dp, sort

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 minimumTotalDistance(self, robot: List[int], factory: List[List[int]]) ->
int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minimumTotalDistance(self, robot, factory):
"""
:type robot: List[int]
:type factory: List[List[int]]
```

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

JavaScript Solution:

```
/**
* Problem: Minimum Total Distance Traveled
* Difficulty: Hard
* Tags: array, dp, sort
*
* 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[]} robot
* @param {number[][]} factory
* @return {number}
*/
var minimumTotalDistance = function(robot, factory) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Total Distance Traveled
* Difficulty: Hard
* Tags: array, dp, sort
*
* 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 minimumTotalDistance(robot: number[], factory: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Minimum Total Distance Traveled
* Difficulty: Hard
* Tags: array, dp, sort
*
* 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 long MinimumTotalDistance(IList<int> robot, int[][] factory) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Total Distance Traveled
* Difficulty: Hard
* Tags: array, dp, sort
*
* 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
*/

long long minimumTotalDistance(int* robot, int robotSize, int** factory, int
factorySize, int* factoryColSize) {

}
```

Go Solution:

```
// Problem: Minimum Total Distance Traveled
// Difficulty: Hard
// Tags: array, dp, sort
//
// 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 minimumTotalDistance(robot []int, factory [][]int) int64 {

}
```

Kotlin Solution:

```
class Solution {
fun minimumTotalDistance(robot: List<Int>, factory: Array<IntArray>): Long {

}
}
```

Swift Solution:

```
class Solution {
func minimumTotalDistance(_ robot: [Int], _ factory: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Minimum Total Distance Traveled
// Difficulty: Hard
// Tags: array, dp, sort
//
// 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 minimum_total_distance(robot: Vec<i32>, factory: Vec<Vec<i32>>) -> i64
{

}
}
```

Ruby Solution:

```
# @param {Integer[]} robot
# @param {Integer[][]} factory
# @return {Integer}
def minimum_total_distance(robot, factory)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $robot
* @param Integer[][] $factory
* @return Integer
*/
function minimumTotalDistance($robot, $factory) {

}
}
```

Dart Solution:

```
class Solution {
int minimumTotalDistance(List<int> robot, List<List<int>> factory) {

}
}
```

Scala Solution:

```
object Solution {
def minimumTotalDistance(robot: List[Int], factory: Array[Array[Int]]): Long
= {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec minimum_total_distance(robot :: [integer], factory :: [[integer]]) ::
```

```
integer
def minimum_total_distance(robot, factory) do

end
end
```

Erlang Solution:

```
-spec minimum_total_distance(Robot :: [integer()], Factory :: [[integer()]])
-> integer().
minimum_total_distance(Robot, Factory) ->
.
```

Racket Solution:

```
(define/contract (minimum-total-distance robot factory)
(-> (listof exact-integer?) (listof (listof exact-integer?)) exact-integer?)
)
```