Problem 874: Walking Robot Simulation

Problem Information

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

Problem Description

A robot on an infinite XY-plane starts at point

(0, 0)

facing north. The robot receives an array of integers

commands

, which represents a sequence of moves that it needs to execute. There are only three possible types of instructions the robot can receive:

-2

: Turn left

90

degrees.

-1

: Turn right

90

degrees.

1 <= k <= 9

: Move forward

k

units, one unit at a time.

Some of the grid squares are

obstacles

. The

i

th

obstacle is at grid point

obstacles[i] = (x

i

, y

i

)

. If the robot runs into an obstacle, it will stay in its current location (on the block adjacent to the obstacle) and move onto the next command.

Return the

maximum squared Euclidean distance

that the robot reaches at any point in its path (i.e. if the distance is

5

, return

25

).

Note:

There can be an obstacle at

(0, 0)

. If this happens, the robot will ignore the obstacle until it has moved off the origin. However, it will be unable to return to

(0, 0)

due to the obstacle.

North means +Y direction.

East means +X direction.

South means -Y direction.

West means -X direction.

Example 1:

Input:

commands = [4,-1,3], obstacles = []

Output:

25

Explanation:

The robot starts at

(0, 0)

:

Move north 4 units to

(0, 4)

.

Turn right.

Move east 3 units to

(3, 4)

.

The furthest point the robot ever gets from the origin is

(3, 4)

, which squared is

3

2

+ 4

2

= 25

units away.

Example 2:

Input:

commands = [4,-1,4,-2,4], obstacles = [[2,4]]

Output:

65

Explanation:

The robot starts at

(0, 0)

:

Move north 4 units to

(0, 4)

.

Turn right.

Move east 1 unit and get blocked by the obstacle at

(2, 4)

, robot is at

(1, 4)

.

Turn left.

Move north 4 units to

(1, 8)

.

The furthest point the robot ever gets from the origin is

(1, 8)

, which squared is

1

2

+ 8

2

= 65

units away.

Example 3:

Input:

commands = [6,-1,-1,6], obstacles = [[0,0]]

Output:

36

Explanation:

The robot starts at

(0, 0)

:

Move north 6 units to

(0, 6)

.

Turn right.

Turn right.

Move south 5 units and get blocked by the obstacle at

(0,0)

, robot is at

(0, 1)

.

The furthest point the robot ever gets from the origin is

(0, 6)

, which squared is

6

2

= 36

units away.

Constraints:

1 <= commands.length <= 10

4

commands[i]

is either

-2

,

-1

, or an integer in the range

[1, 9]

.

0 <= obstacles.length <= 10

4

-3 * 10

4

<= x

i

, y

i

<= 3 * 10

4

The answer is guaranteed to be less than

2

31

.

Code Snippets

C++:

```
class Solution {
public:
int robotSim(vector<int>& commands, vector<vector<int>>& obstacles) {

}
};
```

Java:

```
class Solution {
public int robotSim(int[] commands, int[][] obstacles) {

}
}
```

Python3:

```
class Solution:
def robotSim(self, commands: List[int], obstacles: List[List[int]]) -> int:
```

Python:

```
class Solution(object):
def robotSim(self, commands, obstacles):
"""
:type commands: List[int]
```

```
:type obstacles: List[List[int]]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} commands
* @param {number[][]} obstacles
* @return {number}
*/
var robotSim = function(commands, obstacles) {

};
```

TypeScript:

```
function robotSim(commands: number[], obstacles: number[][]): number {

};
```

C#:

```
public class Solution {
public int RobotSim(int[] commands, int[][] obstacles) {

}
}
```

C:

```
int robotSim(int* commands, int commandsSize, int** obstacles, int
obstaclesSize, int* obstaclesColSize) {

}
```

Go:

```
func robotSim(commands []int, obstacles [][]int) int {

}
```

Kotlin:

```
class Solution {
fun robotSim(commands: IntArray, obstacles: Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func robotSim(_ commands: [Int], _ obstacles: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn robot_sim(commands: Vec<i32>, obstacles: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} commands
# @param {Integer[][]} obstacles
# @return {Integer}
def robot_sim(commands, obstacles)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $commands
* @param Integer[][] $obstacles
* @return Integer
*/
function robotSim($commands, $obstacles) {
```

```
}
}
```

Dart:

```
class Solution {
int robotSim(List<int> commands, List<List<int>> obstacles) {

}
}
```

Scala:

```
object Solution {
def robotSim(commands: Array[Int], obstacles: Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec robot_sim(commands :: [integer], obstacles :: [[integer]]) :: integer
def robot_sim(commands, obstacles) do

end
end
```

Erlang:

```
-spec robot_sim(Commands :: [integer()], Obstacles :: [[integer()]]) ->
integer().
robot_sim(Commands, Obstacles) ->
.
```

Racket:

```
(define/contract (robot-sim commands obstacles)
(-> (listof exact-integer?) (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Walking Robot Simulation
* Difficulty: Medium
* Tags: array, tree, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public:
int robotSim(vector<int>& commands, vector<vector<int>>& obstacles) {

}
};
```

Java Solution:

```
/**
* Problem: Walking Robot Simulation
* Difficulty: Medium
* Tags: array, tree, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public int robotSim(int[] commands, int[][] obstacles) {

}
}
```

Python3 Solution:

```
"""
Problem: Walking Robot Simulation
Difficulty: Medium
Tags: array, tree, hash

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

class Solution:
def robotSim(self, commands: List[int], obstacles: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def robotSim(self, commands, obstacles):
"""
:type commands: List[int]
:type obstacles: List[List[int]]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Walking Robot Simulation
* Difficulty: Medium
* Tags: array, tree, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* @param {number[]} commands
* @param {number[][]} obstacles
* @return {number}
*/
```

```
var robotSim = function(commands, obstacles) {

};
```

TypeScript Solution:

```
/**
* Problem: Walking Robot Simulation
* Difficulty: Medium
* Tags: array, tree, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

function robotSim(commands: number[], obstacles: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Walking Robot Simulation
* Difficulty: Medium
* Tags: array, tree, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

public class Solution {
public int RobotSim(int[] commands, int[][] obstacles) {

}
}
```

C Solution:

```
/*
* Problem: Walking Robot Simulation
* Difficulty: Medium
* Tags: array, tree, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

int robotSim(int* commands, int commandsSize, int** obstacles, int
obstaclesSize, int* obstaclesColSize) {

}
```

Go Solution:

```
// Problem: Walking Robot Simulation
// Difficulty: Medium
// Tags: array, tree, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

func robotSim(commands []int, obstacles [][]int) int {

}
```

Kotlin Solution:

```
class Solution {
fun robotSim(commands: IntArray, obstacles: Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func robotSim(_ commands: [Int], _ obstacles: [[Int]]) -> Int {
```

```
}
}
```

Rust Solution:

```
// Problem: Walking Robot Simulation
// Difficulty: Medium
// Tags: array, tree, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

impl Solution {
pub fn robot_sim(commands: Vec<i32>, obstacles: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} commands
# @param {Integer[][]} obstacles
# @return {Integer}
def robot_sim(commands, obstacles)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $commands
* @param Integer[][] $obstacles
* @return Integer
*/
function robotSim($commands, $obstacles) {

}
}
```

Dart Solution:

```
class Solution {
int robotSim(List<int> commands, List<List<int>> obstacles) {

}
}
```

Scala Solution:

```
object Solution {
def robotSim(commands: Array[Int], obstacles: Array[Array[Int]]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec robot_sim(commands :: [integer], obstacles :: [[integer]]) :: integer
def robot_sim(commands, obstacles) do

end
end
```

Erlang Solution:

```
-spec robot_sim(Commands :: [integer()], Obstacles :: [[integer()]]) ->
integer().
robot_sim(Commands, Obstacles) ->
.
```

Racket Solution:

```
(define/contract (robot-sim commands obstacles)
(-> (listof exact-integer?) (listof (listof exact-integer?)) exact-integer?)
)
```