Problem 353: Design Snake Game

Problem Information

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

Problem Description

Design a

Snake game

that is played on a device with screen size

height x width

.

Play the game online

if you are not familiar with the game.

The snake is initially positioned at the top left corner

(0, 0)

with a length of

1

unit.

You are given an array

food

where

food[i] = (r

i

, c

i

)

is the row and column position of a piece of food that the snake can eat. When a snake eats a piece of food, its length and the game's score both increase by

1

.

Each piece of food appears one by one on the screen, meaning the second piece of food will not appear until the snake eats the first piece of food.

When a piece of food appears on the screen, it is

guaranteed

that it will not appear on a block occupied by the snake.

The game is over if the snake goes out of bounds (hits a wall) or if its head occupies a space that its body occupies

after

moving (i.e. a snake of length 4 cannot run into itself).

Implement the

SnakeGame

class:

SnakeGame(int width, int height, int[][] food)

Initializes the object with a screen of size

height x width

and the positions of the

food

.

int move(String direction)

Returns the score of the game after applying one

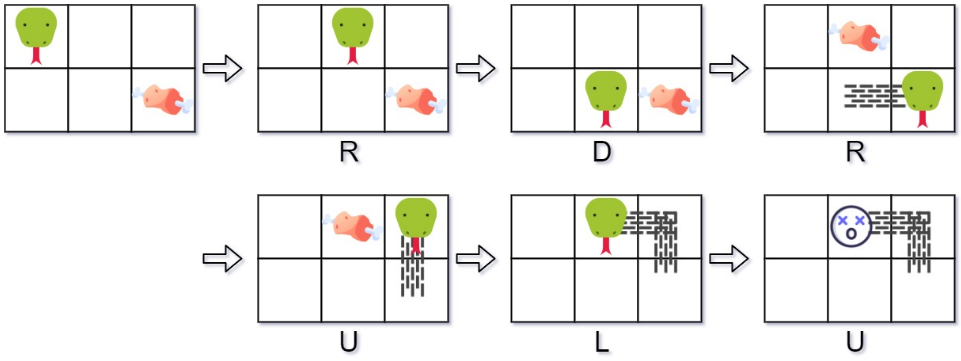
direction

move by the snake. If the game is over, return

-1

.

Example 1:



Input

["SnakeGame", "move", "move", "move", "move", "move", "move"] [[3, 2, [[1, 2], [0, 1]]], ["R"], ["D"], ["R"], ["U"], ["L"], ["U"]]

Output

[null, 0, 0, 1, 1, 2, -1]

Explanation

SnakeGame snakeGame = new SnakeGame(3, 2, [[1, 2], [0, 1]]); snakeGame.move("R"); // return 0 snakeGame.move("D"); // return 0 snakeGame.move("R"); // return 1, snake eats the first piece of food. The second piece of food appears at (0, 1). snakeGame.move("U"); // return 1 snakeGame.move("L"); // return 2, snake eats the second food. No more food appears. snakeGame.move("U"); // return -1, game over because snake collides with border

Constraints:

1 <= width, height <= 10

4

1 <= food.length <= 50

food[i].length == 2

0 <= r

i

< height

0 <= c

i

< width

direction.length == 1

direction

is

'U'

,

'D'

,

'L'

, or

'R'

.

At most

10

4

calls will be made to

move

.

Code Snippets

C++:

```
class SnakeGame {
public:
SnakeGame(int width, int height, vector<vector<int>>& food) {

}

int move(string direction) {

}
};

/**
* Your SnakeGame object will be instantiated and called as such:
* SnakeGame* obj = new SnakeGame(width, height, food);
* int param_1 = obj->move(direction);
*/
```

Java:

```
class SnakeGame {

public SnakeGame(int width, int height, int[][] food) {

}

public int move(String direction) {

}
}

/**
* Your SnakeGame object will be instantiated and called as such:
* SnakeGame obj = new SnakeGame(width, height, food);
* int param_1 = obj.move(direction);
*/
```

Python3:

```
class SnakeGame:

def __init__(self, width: int, height: int, food: List[List[int]]):
```

```

def move(self, direction: str) -> int:




# Your SnakeGame object will be instantiated and called as such:
# obj = SnakeGame(width, height, food)
# param_1 = obj.move(direction)
```

Python:

```
class SnakeGame(object):

def __init__(self, width, height, food):
"""
:type width: int
:type height: int
:type food: List[List[int]]
"""



def move(self, direction):
"""
:type direction: str
:rtype: int
"""




# Your SnakeGame object will be instantiated and called as such:
# obj = SnakeGame(width, height, food)
# param_1 = obj.move(direction)
```

JavaScript:

```
/**
* @param {number} width
* @param {number} height
* @param {number[][]} food
*/
var SnakeGame = function(width, height, food) {

};
```

```
/**
* @param {string} direction
* @return {number}
*/
SnakeGame.prototype.move = function(direction) {

};

/**
* Your SnakeGame object will be instantiated and called as such:
* var obj = new SnakeGame(width, height, food)
* var param_1 = obj.move(direction)
*/
```

TypeScript:

```
class SnakeGame {
constructor(width: number, height: number, food: number[][]) {

}

move(direction: string): number {

}
}

/**
* Your SnakeGame object will be instantiated and called as such:
* var obj = new SnakeGame(width, height, food)
* var param_1 = obj.move(direction)
*/
```

C#:

```
public class SnakeGame {

public SnakeGame(int width, int height, int[][] food) {

}

public int Move(string direction) {
}
}

/**
* Your SnakeGame object will be instantiated and called as such:
* SnakeGame obj = new SnakeGame(width, height, food);
* int param_1 = obj.Move(direction);
*/
```

C:

```
typedef struct {

} SnakeGame;


SnakeGame* snakeGameCreate(int width, int height, int** food, int foodSize,
int* foodColSize) {

}

int snakeGameMove(SnakeGame* obj, char * direction) {

}

void snakeGameFree(SnakeGame* obj) {

}

/**
* Your SnakeGame struct will be instantiated and called as such:
* SnakeGame* obj = snakeGameCreate(width, height, food, foodSize,
foodColSize);
* int param_1 = snakeGameMove(obj, direction);

* snakeGameFree(obj);
*/
```

Go:

```
type SnakeGame struct {

}


func Constructor(width int, height int, food [][]int) SnakeGame {

}


func (this *SnakeGame) Move(direction string) int {

}


/**
* Your SnakeGame object will be instantiated and called as such:
* obj := Constructor(width, height, food);
* param_1 := obj.Move(direction);
*/
```

Kotlin:

```
class SnakeGame(width: Int, height: Int, food: Array<IntArray>) {

fun move(direction: String): Int {

}

}

/**
* Your SnakeGame object will be instantiated and called as such:
* var obj = SnakeGame(width, height, food)
* var param_1 = obj.move(direction)
*/
```

Swift:

```


class SnakeGame {
```

```


init(_ width: Int, _ height: Int, _ food: [[Int]]) {

}

func move(_ direction: String) -> Int {

}
}

/**
* Your SnakeGame object will be instantiated and called as such:
* let obj = SnakeGame(width, height, food)
* let ret_1: Int = obj.move(direction)
*/
```

Rust:

```
struct SnakeGame {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl SnakeGame {

fn new(width: i32, height: i32, food: Vec<Vec<i32>>) -> Self {

}

fn make_a_move(&self, direction: String) -> i32 {

}
}

/**
* Your SnakeGame object will be instantiated and called as such:
* let obj = SnakeGame::new(width, height, food);
* let ret_1: i32 = obj.move(direction);
```

```
*/
```

Ruby:

```
class SnakeGame

=begin
:type width: Integer
:type height: Integer
:type food: Integer[][]
=end
def initialize(width, height, food)

end



=begin
:type direction: String
:rtype: Integer
=end
def move(direction)

end



end

# Your SnakeGame object will be instantiated and called as such:
# obj = SnakeGame.new(width, height, food)
# param_1 = obj.move(direction)
```

PHP:

```
class SnakeGame {
/**
* @param Integer $width
* @param Integer $height
* @param Integer[][] $food
*/
function __construct($width, $height, $food) {

}
/**
* @param String $direction
* @return Integer
*/
function move($direction) {

}
}

/**
* Your SnakeGame object will be instantiated and called as such:
* $obj = SnakeGame($width, $height, $food);
* $ret_1 = $obj->move($direction);
*/
```

Scala:

```
class SnakeGame(_width: Int, _height: Int, _food: Array[Array[Int]]) {

def move(direction: String): Int = {

}

}

/**
* Your SnakeGame object will be instantiated and called as such:
* var obj = new SnakeGame(width, height, food)
* var param_1 = obj.move(direction)
*/
```

Elixir:

```
defmodule SnakeGame do
@spec init_(width :: integer, height :: integer, food :: [[integer]]) :: any
def init_(width, height, food) do

end

@spec move(direction :: String.t) :: integer
def move(direction) do
```

```
end
end

# Your functions will be called as such:
# SnakeGame.init_(width, height, food)
# param_1 = SnakeGame.move(direction)

# SnakeGame.init_ will be called before every test case, in which you can do
some necessary initializations.
```

Erlang:

```
-spec snake_game_init_(Width :: integer(), Height :: integer(), Food ::
[[integer()]]) -> any().
snake_game_init_(Width, Height, Food) ->
.

-spec snake_game_move(Direction :: unicode:unicode_binary()) -> integer().
snake_game_move(Direction) ->
.


%% Your functions will be called as such:
%% snake_game_init_(Width, Height, Food),
%% Param_1 = snake_game_move(Direction),

%% snake_game_init_ will be called before every test case, in which you can
do some necessary initializations.
```

Racket:

```
(define snake-game%
(class object%
(super-new)

; width : exact-integer?

; height : exact-integer?

; food : (listof (listof exact-integer?))
(init-field
width
height
food)

; move : string? -> exact-integer?
(define/public (move direction)

)))

;; Your snake-game% object will be instantiated and called as such:
;; (define obj (new snake-game% [width width] [height height] [food food]))
;; (define param_1 (send obj move direction))
```

Solutions

C++ Solution:

```
/*
* Problem: Design Snake Game
* Difficulty: Medium
* Tags: array, string, hash, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class SnakeGame {
public:
SnakeGame(int width, int height, vector<vector<int>>& food) {

}

int move(string direction) {

}
};

/**
* Your SnakeGame object will be instantiated and called as such:
```

```
* SnakeGame* obj = new SnakeGame(width, height, food);
* int param_1 = obj->move(direction);
*/
```

Java Solution:

```
/**
* Problem: Design Snake Game
* Difficulty: Medium
* Tags: array, string, hash, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class SnakeGame {

public SnakeGame(int width, int height, int[][] food) {

}

public int move(String direction) {

}
}

/**
* Your SnakeGame object will be instantiated and called as such:
* SnakeGame obj = new SnakeGame(width, height, food);
* int param_1 = obj.move(direction);
*/
```

Python3 Solution:

```
"""
Problem: Design Snake Game
Difficulty: Medium
Tags: array, string, hash, queue

Approach: Use two pointers or sliding window technique
```

```
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class SnakeGame:

def __init__(self, width: int, height: int, food: List[List[int]]):


def move(self, direction: str) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class SnakeGame(object):

def __init__(self, width, height, food):
"""
:type width: int
:type height: int
:type food: List[List[int]]
"""



def move(self, direction):
"""
:type direction: str
:rtype: int
"""




# Your SnakeGame object will be instantiated and called as such:
# obj = SnakeGame(width, height, food)
# param_1 = obj.move(direction)
```

JavaScript Solution:

```
/**
* Problem: Design Snake Game
* Difficulty: Medium
* Tags: array, string, hash, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {number} width
* @param {number} height
* @param {number[][]} food
*/
var SnakeGame = function(width, height, food) {

};

/**
* @param {string} direction
* @return {number}
*/
SnakeGame.prototype.move = function(direction) {

};

/**
* Your SnakeGame object will be instantiated and called as such:
* var obj = new SnakeGame(width, height, food)
* var param_1 = obj.move(direction)
*/
```

TypeScript Solution:

```
/**
* Problem: Design Snake Game
* Difficulty: Medium
* Tags: array, string, hash, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
```

```
*/

class SnakeGame {
constructor(width: number, height: number, food: number[][]) {

}

move(direction: string): number {

}
}

/**
* Your SnakeGame object will be instantiated and called as such:
* var obj = new SnakeGame(width, height, food)
* var param_1 = obj.move(direction)
*/
```

C# Solution:

```
/*
* Problem: Design Snake Game
* Difficulty: Medium
* Tags: array, string, hash, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class SnakeGame {

public SnakeGame(int width, int height, int[][] food) {

}

public int Move(string direction) {

}
}
```

```
/**
* Your SnakeGame object will be instantiated and called as such:
* SnakeGame obj = new SnakeGame(width, height, food);
* int param_1 = obj.Move(direction);
*/
```

C Solution:

```
/*
* Problem: Design Snake Game
* Difficulty: Medium
* Tags: array, string, hash, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/





typedef struct {

} SnakeGame;



SnakeGame* snakeGameCreate(int width, int height, int** food, int foodSize,
int* foodColSize) {

}

int snakeGameMove(SnakeGame* obj, char * direction) {

}

void snakeGameFree(SnakeGame* obj) {

}

/**
* Your SnakeGame struct will be instantiated and called as such:
```

```
* SnakeGame* obj = snakeGameCreate(width, height, food, foodSize,
foodColSize);
* int param_1 = snakeGameMove(obj, direction);

* snakeGameFree(obj);
*/
```

Go Solution:

```
// Problem: Design Snake Game
// Difficulty: Medium
// Tags: array, string, hash, queue
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

type SnakeGame struct {

}


func Constructor(width int, height int, food [][]int) SnakeGame {

}


func (this *SnakeGame) Move(direction string) int {

}


/**
* Your SnakeGame object will be instantiated and called as such:
* obj := Constructor(width, height, food);
* param_1 := obj.Move(direction);
*/
```

Kotlin Solution:

```
class SnakeGame(width: Int, height: Int, food: Array<IntArray>) {

fun move(direction: String): Int {

}

}

/**
* Your SnakeGame object will be instantiated and called as such:
* var obj = SnakeGame(width, height, food)
* var param_1 = obj.move(direction)
*/
```

Swift Solution:

```

class SnakeGame {

init(_ width: Int, _ height: Int, _ food: [[Int]]) {

}

func move(_ direction: String) -> Int {

}
}

/**
* Your SnakeGame object will be instantiated and called as such:
* let obj = SnakeGame(width, height, food)
* let ret_1: Int = obj.move(direction)
*/
```

Rust Solution:

```
// Problem: Design Snake Game
// Difficulty: Medium
// Tags: array, string, hash, queue
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

struct SnakeGame {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl SnakeGame {

fn new(width: i32, height: i32, food: Vec<Vec<i32>>) -> Self {

}

fn make_a_move(&self, direction: String) -> i32 {

}
}

/**
* Your SnakeGame object will be instantiated and called as such:
* let obj = SnakeGame::new(width, height, food);
* let ret_1: i32 = obj.move(direction);
*/
```

Ruby Solution:

```
class SnakeGame

=begin
:type width: Integer
:type height: Integer
:type food: Integer[][]
=end
def initialize(width, height, food)

end
```

```
=begin
:type direction: String
:rtype: Integer
=end
def move(direction)

end



end

# Your SnakeGame object will be instantiated and called as such:
# obj = SnakeGame.new(width, height, food)
# param_1 = obj.move(direction)
```

PHP Solution:

```
class SnakeGame {
/**
* @param Integer $width
* @param Integer $height
* @param Integer[][] $food
*/
function __construct($width, $height, $food) {

}

/**
* @param String $direction
* @return Integer
*/
function move($direction) {

}
}

/**
* Your SnakeGame object will be instantiated and called as such:
* $obj = SnakeGame($width, $height, $food);
* $ret_1 = $obj->move($direction);
```

```

*/
```

Scala Solution:

```
class SnakeGame(_width: Int, _height: Int, _food: Array[Array[Int]]) {

def move(direction: String): Int = {

}

}

/**
* Your SnakeGame object will be instantiated and called as such:
* var obj = new SnakeGame(width, height, food)
* var param_1 = obj.move(direction)
*/
```

Elixir Solution:

```
defmodule SnakeGame do
@spec init_(width :: integer, height :: integer, food :: [[integer]]) :: any
def init_(width, height, food) do

end

@spec move(direction :: String.t) :: integer
def move(direction) do

end
end

# Your functions will be called as such:
# SnakeGame.init_(width, height, food)
# param_1 = SnakeGame.move(direction)

# SnakeGame.init_ will be called before every test case, in which you can do
some necessary initializations.
```

Erlang Solution:

```
-spec snake_game_init_(Width :: integer(), Height :: integer(), Food ::
[[integer()]]) -> any().
snake_game_init_(Width, Height, Food) ->
.

-spec snake_game_move(Direction :: unicode:unicode_binary()) -> integer().
snake_game_move(Direction) ->
.



%% Your functions will be called as such:
%% snake_game_init_(Width, Height, Food),
%% Param_1 = snake_game_move(Direction),

%% snake_game_init_ will be called before every test case, in which you can
do some necessary initializations.
```

Racket Solution:

```
(define snake-game%
(class object%
(super-new)

; width : exact-integer?

; height : exact-integer?

; food : (listof (listof exact-integer?))
(init-field
width
height
food)

; move : string? -> exact-integer?
(define/public (move direction)

)))

;; Your snake-game% object will be instantiated and called as such:
;; (define obj (new snake-game% [width width] [height height] [food food]))
;; (define param_1 (send obj move direction))
```