Problem 980: Unique Paths III

Problem Information

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

Problem Description

You are given an

m x n

integer array

grid

where

grid[i][j]

could be:

1

representing the starting square. There is exactly one starting square.

2

representing the ending square. There is exactly one ending square.

0

representing empty squares we can walk over.

-1

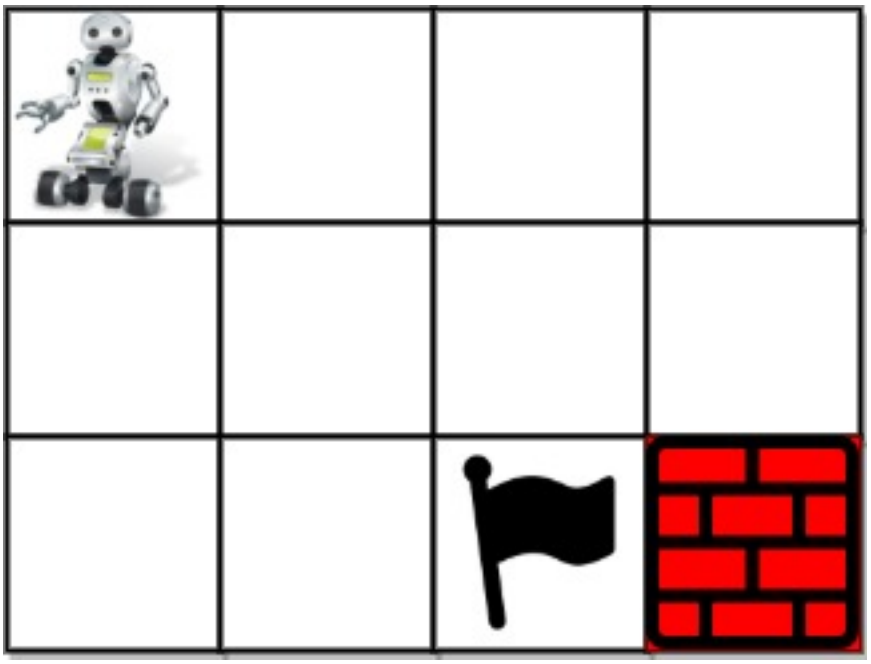
representing obstacles that we cannot walk over.

Return

the number of 4-directional walks from the starting square to the ending square, that walk over every non-obstacle square exactly once

.

Example 1:



Input:

grid = [[1,0,0,0],[0,0,0,0],[0,0,2,-1]]

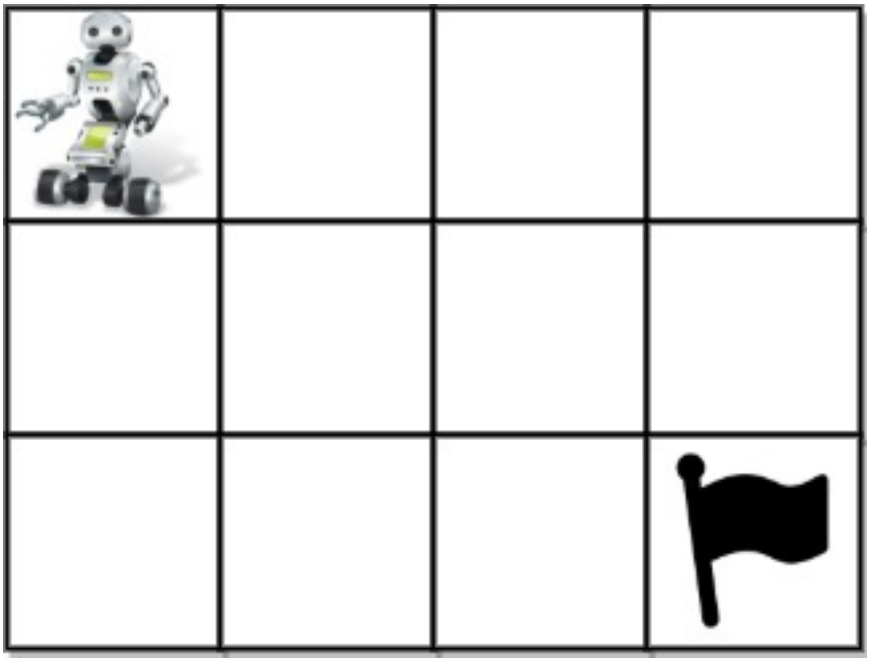
Output:

2

Explanation:

We have the following two paths: 1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2) 2. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2)

Example 2:



Input:

grid = [[1,0,0,0],[0,0,0,0],[0,0,0,2]]

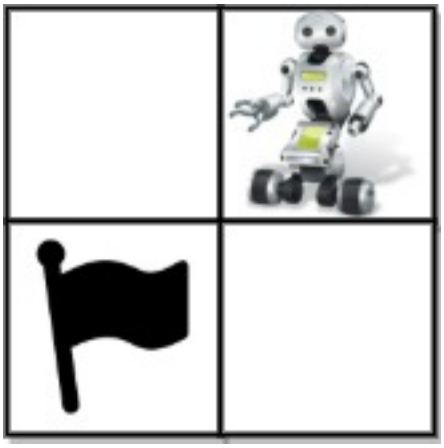
Output:

4

Explanation:

We have the following four paths: 1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2),(2,3) 2. (0,0),(0,1),(1,1),(1,0),(2,0),(2,1),(2,2),(1,2),(0,2),(0,3),(1,3),(2,3) 3. (0,0),(1,0),(2,0),(2,1),(2,2),(1,2),(1,1),(0,1),(0,2),(0,3),(1,3),(2,3) 4. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2),(2,3)

Example 3:



Input:

grid = [[0,1],[2,0]]

Output:

0

Explanation:

There is no path that walks over every empty square exactly once. Note that the starting and ending square can be anywhere in the grid.

Constraints:

m == grid.length

n == grid[i].length

1 <= m, n <= 20

1 <= m * n <= 20

-1 <= grid[i][j] <= 2

There is exactly one starting cell and one ending cell.

Code Snippets

C++:

```
class Solution {
public:
int uniquePathsIII(vector<vector<int>>& grid) {

}
};
```

Java:

```
class Solution {
public int uniquePathsIII(int[][] grid) {

}
}
```

Python3:

```
class Solution:
def uniquePathsIII(self, grid: List[List[int]]) -> int:
```

Python:

```
class Solution(object):
def uniquePathsIII(self, grid):
"""
:type grid: List[List[int]]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[][]} grid
* @return {number}
*/
var uniquePathsIII = function(grid) {

};
```

TypeScript:

```
function uniquePathsIII(grid: number[][]): number {

};
```

C#:

```
public class Solution {
public int UniquePathsIII(int[][] grid) {

}
}
```

C:

```
int uniquePathsIII(int** grid, int gridSize, int* gridColSize) {

}
```

Go:

```
func uniquePathsIII(grid [][]int) int {

}
```

Kotlin:

```
class Solution {
fun uniquePathsIII(grid: Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func uniquePathsIII(_ grid: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn unique_paths_iii(grid: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[][]} grid
# @return {Integer}
def unique_paths_iii(grid)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $grid
* @return Integer
*/
function uniquePathsIII($grid) {

}
}
```

Dart:

```
class Solution {
int uniquePathsIII(List<List<int>> grid) {

}
}
```

Scala:

```
object Solution {
def uniquePathsIII(grid: Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec unique_paths_iii(grid :: [[integer]]) :: integer
def unique_paths_iii(grid) do

end
end
```

Erlang:

```
-spec unique_paths_iii(Grid :: [[integer()]]) -> integer().
unique_paths_iii(Grid) ->
.
```

Racket:

```
(define/contract (unique-paths-iii grid)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Unique Paths III
* Difficulty: Hard
* Tags: array, tree
*
* 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 uniquePathsIII(vector<vector<int>>& grid) {

}
};
```

Java Solution:

```
/**
* Problem: Unique Paths III
* Difficulty: Hard
* Tags: array, tree
*
* 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 uniquePathsIII(int[][] grid) {

}
}
```

Python3 Solution:

```
"""
Problem: Unique Paths III
Difficulty: Hard
Tags: array, tree

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 uniquePathsIII(self, grid: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def uniquePathsIII(self, grid):
"""
:type grid: List[List[int]]
:rtype: int
```

```
"""
```

JavaScript Solution:

```
/**
* Problem: Unique Paths III
* Difficulty: Hard
* Tags: array, tree
*
* 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[][]} grid
* @return {number}
*/
var uniquePathsIII = function(grid) {

};
```

TypeScript Solution:

```
/**
* Problem: Unique Paths III
* Difficulty: Hard
* Tags: array, tree
*
* 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 uniquePathsIII(grid: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Unique Paths III
* Difficulty: Hard
* Tags: array, tree
*
* 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 UniquePathsIII(int[][] grid) {

}
}
```

C Solution:

```
/*
* Problem: Unique Paths III
* Difficulty: Hard
* Tags: array, tree
*
* 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 uniquePathsIII(int** grid, int gridSize, int* gridColSize) {

}
```

Go Solution:

```
// Problem: Unique Paths III
// Difficulty: Hard
// Tags: array, tree
//
// 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 uniquePathsIII(grid [][]int) int {

}
```

Kotlin Solution:

```
class Solution {
fun uniquePathsIII(grid: Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func uniquePathsIII(_ grid: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Unique Paths III
// Difficulty: Hard
// Tags: array, tree
//
// 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 unique_paths_iii(grid: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} grid
# @return {Integer}
def unique_paths_iii(grid)
```

```
end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $grid
* @return Integer
*/
function uniquePathsIII($grid) {

}
}
```

Dart Solution:

```
class Solution {
int uniquePathsIII(List<List<int>> grid) {

}
}
```

Scala Solution:

```
object Solution {
def uniquePathsIII(grid: Array[Array[Int]]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec unique_paths_iii(grid :: [[integer]]) :: integer
def unique_paths_iii(grid) do

end
end
```

Erlang Solution:

```
-spec unique_paths_iii(Grid :: [[integer()]]) -> integer().
unique_paths_iii(Grid) ->
.
```

Racket Solution:

```
(define/contract (unique-paths-iii grid)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```