Problem 2713: Maximum Strictly Increasing Cells in a Matrix

Problem Information

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

Problem Description

Given a

1-indexed

m x n

integer matrix

mat

, you can select any cell in the matrix as your

starting cell

.

From the starting cell, you can move to any other cell

in the

same row or column

, but only if the value of the destination cell is

strictly greater

than the value of the current cell. You can repeat this process as many times as possible, moving from cell to cell until you can no longer make any moves.

Your task is to find the

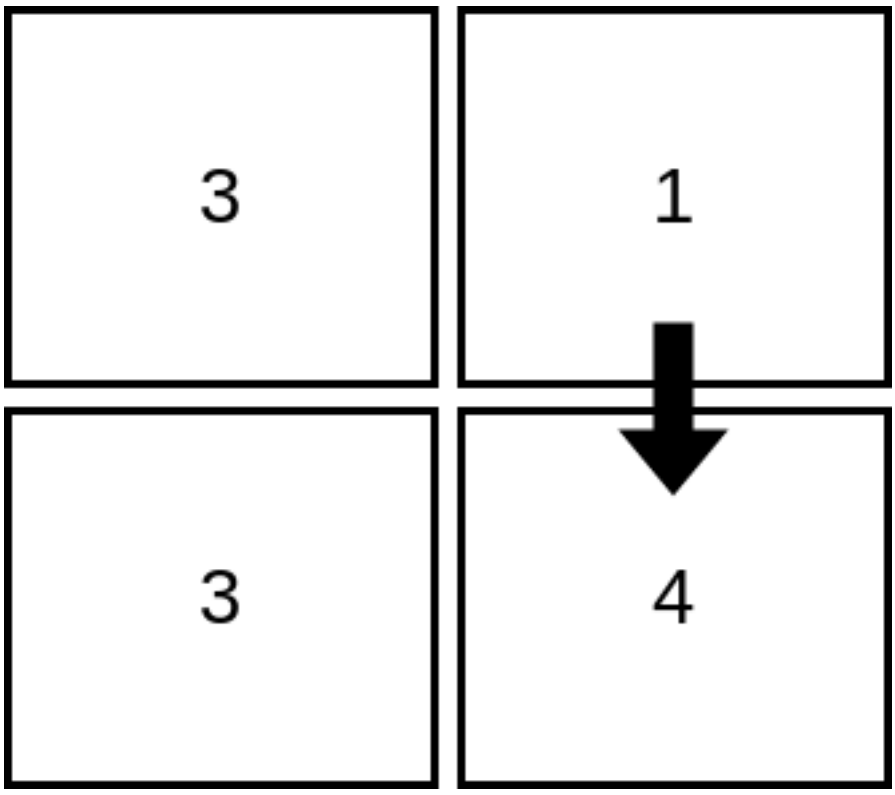
maximum number of cells

that you can visit in the matrix by starting from some cell.

Return

an integer denoting the maximum number of cells that can be visited.

Example 1:



Input:

mat = [[3,1],[3,4]]

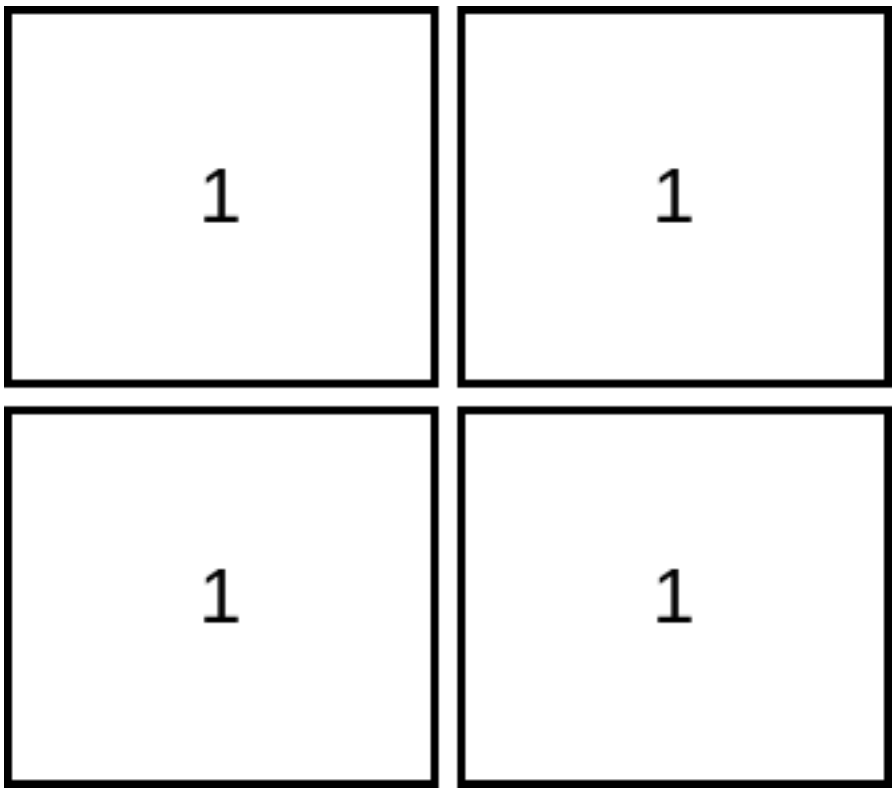
Output:

2

Explanation:

The image shows how we can visit 2 cells starting from row 1, column 2. It can be shown that we cannot visit more than 2 cells no matter where we start from, so the answer is 2.

Example 2:



Input:

mat = [[1,1],[1,1]]

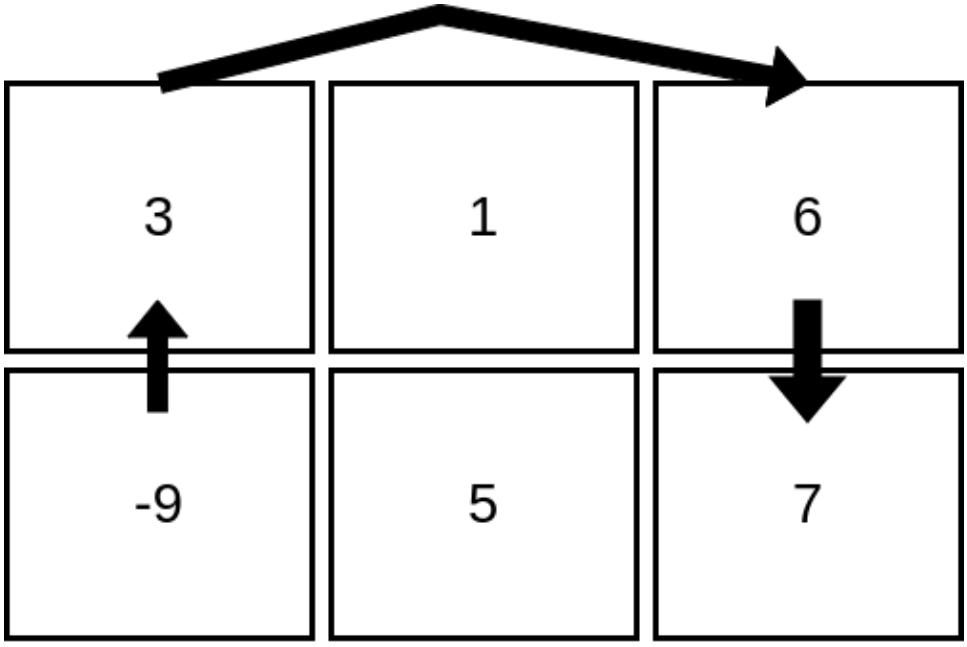
Output:

1

Explanation:

Since the cells must be strictly increasing, we can only visit one cell in this example.

Example 3:



Input:

mat = [[3,1,6],[-9,5,7]]

Output:

4

Explanation:

The image above shows how we can visit 4 cells starting from row 2, column 1. It can be shown that we cannot visit more than 4 cells no matter where we start from, so the answer is 4.

Constraints:

m == mat.length

n == mat[i].length

1 <= m, n <= 10

5

1 <= m * n <= 10

5

-10

5

<= mat[i][j] <= 10

5

Code Snippets

C++:

```
class Solution {
public:
int maxIncreasingCells(vector<vector<int>>& mat) {

}
};
```

Java:

```
class Solution {
public int maxIncreasingCells(int[][] mat) {

}
}
```

Python3:

```
class Solution:
def maxIncreasingCells(self, mat: List[List[int]]) -> int:
```

Python:

```
class Solution(object):
def maxIncreasingCells(self, mat):
```

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

JavaScript:

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

};
```

TypeScript:

```
function maxIncreasingCells(mat: number[][]): number {

};
```

C#:

```
public class Solution {
public int MaxIncreasingCells(int[][] mat) {

}
}
```

C:

```
int maxIncreasingCells(int** mat, int matSize, int* matColSize) {

}
```

Go:

```
func maxIncreasingCells(mat [][]int) int {

}
```

Kotlin:

```
class Solution {
fun maxIncreasingCells(mat: Array<IntArray>): Int {

}
}
```

Swift:

```
class Solution {
func maxIncreasingCells(_ mat: [[Int]]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn max_increasing_cells(mat: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[][]} mat
# @return {Integer}
def max_increasing_cells(mat)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $mat
* @return Integer
*/
function maxIncreasingCells($mat) {

}
}
```

Dart:

```
class Solution {
int maxIncreasingCells(List<List<int>> mat) {

}
}
```

Scala:

```
object Solution {
def maxIncreasingCells(mat: Array[Array[Int]]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec max_increasing_cells(mat :: [[integer]]) :: integer
def max_increasing_cells(mat) do

end
end
```

Erlang:

```
-spec max_increasing_cells(Mat :: [[integer()]]) -> integer().
max_increasing_cells(Mat) ->
.
```

Racket:

```
(define/contract (max-increasing-cells mat)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Maximum Strictly Increasing Cells in a Matrix
* Difficulty: Hard
* Tags: array, dp, hash, 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 maxIncreasingCells(vector<vector<int>>& mat) {

}
};
```

Java Solution:

```
/**
* Problem: Maximum Strictly Increasing Cells in a Matrix
* Difficulty: Hard
* Tags: array, dp, hash, 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 maxIncreasingCells(int[][] mat) {

}
}
```

Python3 Solution:

```
"""
Problem: Maximum Strictly Increasing Cells in a Matrix
Difficulty: Hard
Tags: array, dp, hash, 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 maxIncreasingCells(self, mat: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Maximum Strictly Increasing Cells in a Matrix
* Difficulty: Hard
* Tags: array, dp, hash, 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[][]} mat
* @return {number}
*/
var maxIncreasingCells = function(mat) {

};
```

TypeScript Solution:

```
/**
* Problem: Maximum Strictly Increasing Cells in a Matrix
* Difficulty: Hard
* Tags: array, dp, hash, 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 maxIncreasingCells(mat: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Maximum Strictly Increasing Cells in a Matrix
* Difficulty: Hard
* Tags: array, dp, hash, 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 MaxIncreasingCells(int[][] mat) {

}
}
```

C Solution:

```
/*
* Problem: Maximum Strictly Increasing Cells in a Matrix
* Difficulty: Hard
* Tags: array, dp, hash, 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 maxIncreasingCells(int** mat, int matSize, int* matColSize) {

}
```

Go Solution:

```
// Problem: Maximum Strictly Increasing Cells in a Matrix
// Difficulty: Hard
// Tags: array, dp, hash, 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 maxIncreasingCells(mat [][]int) int {

}
```

Kotlin Solution:

```
class Solution {
fun maxIncreasingCells(mat: Array<IntArray>): Int {

}
}
```

Swift Solution:

```
class Solution {
func maxIncreasingCells(_ mat: [[Int]]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Maximum Strictly Increasing Cells in a Matrix
// Difficulty: Hard
// Tags: array, dp, hash, 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_increasing_cells(mat: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} mat
# @return {Integer}
def max_increasing_cells(mat)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $mat
* @return Integer
*/
function maxIncreasingCells($mat) {

}
}
```

Dart Solution:

```
class Solution {
int maxIncreasingCells(List<List<int>> mat) {

}
}
```

Scala Solution:

```
object Solution {
def maxIncreasingCells(mat: Array[Array[Int]]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec max_increasing_cells(mat :: [[integer]]) :: integer
def max_increasing_cells(mat) do

end
end
```

Erlang Solution:

```
-spec max_increasing_cells(Mat :: [[integer()]]) -> integer().
max_increasing_cells(Mat) ->
.
```

Racket Solution:

```
(define/contract (max-increasing-cells mat)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```