Problem 2577: Minimum Time to Visit a Cell In a Grid

Problem Information

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

Problem Description

You are given a

m x n

matrix

grid

consisting of

non-negative

integers where

grid[row][col]

represents the

minimum

time required to be able to visit the cell

(row, col)

, which means you can visit the cell

(row, col)

only when the time you visit it is greater than or equal to

grid[row][col]

.

You are standing in the

top-left

cell of the matrix in the

0

th

second, and you must move to

any

adjacent cell in the four directions: up, down, left, and right. Each move you make takes 1 second.

Return

the

minimum

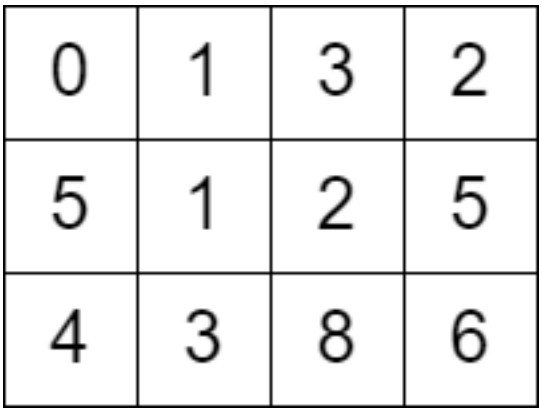
time required in which you can visit the bottom-right cell of the matrix

. If you cannot visit the bottom-right cell, then return

-1

.

Example 1:



Input:

grid = [[0,1,3,2],[5,1,2,5],[4,3,8,6]]

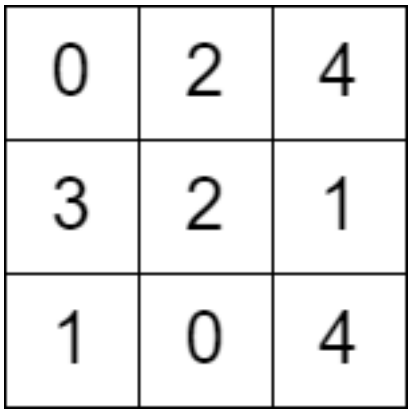
Output:

7

Explanation:

One of the paths that we can take is the following: - at t = 0, we are on the cell (0,0). - at t = 1, we move to the cell (0,1). It is possible because grid[0][1] <= 1. - at t = 2, we move to the cell (1,1). It is possible because grid[1][1] <= 2. - at t = 3, we move to the cell (1,2). It is possible because grid[1][2] <= 3. - at t = 4, we move to the cell (1,1). It is possible because grid[1][1] <= 4. - at t = 5, we move to the cell (1,2). It is possible because grid[1][2] <= 5. - at t = 6, we move to the cell (1,3). It is possible because grid[1][3] <= 6. - at t = 7, we move to the cell (2,3). It is possible because grid[2][3] <= 7. The final time is 7. It can be shown that it is the minimum time possible.

Example 2:



Input:

grid = [[0,2,4],[3,2,1],[1,0,4]]

Output:

-1

Explanation:

There is no path from the top left to the bottom-right cell.

Constraints:

m == grid.length

n == grid[i].length

2 <= m, n <= 1000

4 <= m * n <= 10

5

0 <= grid[i][j] <= 10

5

grid[0][0] == 0

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

```
};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
```

```
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (minimum-time grid)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Time to Visit a Cell In a Grid
* Difficulty: Hard
* Tags: array, graph, search, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

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

}
};
```

Java Solution:

```
/**
* Problem: Minimum Time to Visit a Cell In a Grid
* Difficulty: Hard
* Tags: array, graph, search, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int minimumTime(int[][] grid) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Time to Visit a Cell In a Grid
Difficulty: Hard
Tags: array, graph, search, queue, heap

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Minimum Time to Visit a Cell In a Grid
* Difficulty: Hard
* Tags: array, graph, search, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Time to Visit a Cell In a Grid
* Difficulty: Hard
* Tags: array, graph, search, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

```
};
```

C# Solution:

```
/*
* Problem: Minimum Time to Visit a Cell In a Grid
* Difficulty: Hard
* Tags: array, graph, search, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int MinimumTime(int[][] grid) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Time to Visit a Cell In a Grid
* Difficulty: Hard
* Tags: array, graph, search, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

int minimumTime(int** grid, int gridSize, int* gridColSize) {

}
```

Go Solution:

```
// Problem: Minimum Time to Visit a Cell In a Grid
// Difficulty: Hard
```

```
// Tags: array, graph, search, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func minimumTime(grid [][]int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Minimum Time to Visit a Cell In a Grid
// Difficulty: Hard
// Tags: array, graph, search, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

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

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

```
end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (minimum-time grid)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```