Problem 778: Swim in Rising Water

Problem Information

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

Problem Description

You are given an

n x n

integer matrix

grid

where each value

grid[i][j]

represents the elevation at that point

(i, j)

.

It starts raining, and water gradually rises over time. At time

t

, the water level is

t

, meaning

any

cell with elevation less than equal to

t

is submerged or reachable.

You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most

t

. You can swim infinite distances in zero time. Of course, you must stay within the boundaries of the grid during your swim.

Return

the minimum time until you can reach the bottom right square

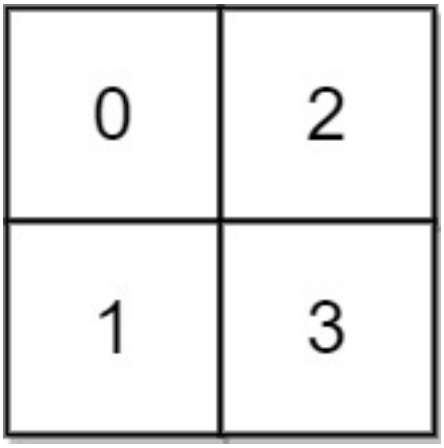
(n - 1, n - 1)

if you start at the top left square

(0, 0)

.

Example 1:



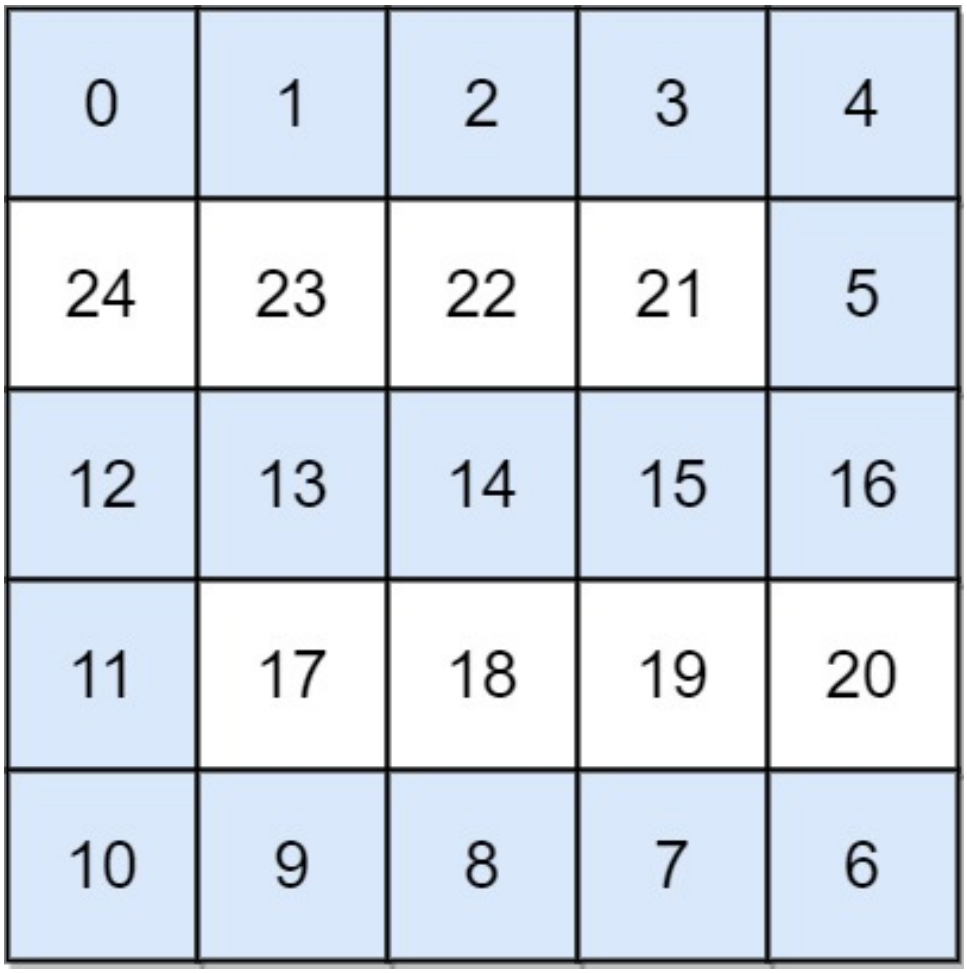
Input:

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

Output:

3 Explanation: At time 0, you are in grid location (0, 0). You cannot go anywhere else because 4-directionally adjacent neighbors have a higher elevation than t = 0. You cannot reach point (1, 1) until time 3. When the depth of water is 3, we can swim anywhere inside the grid.

Example 2:



Input:

grid = [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,6]]

Output:

16

Explanation:

The final route is shown. We need to wait until time 16 so that (0, 0) and (4, 4) are connected.

Constraints:

n == grid.length

n == grid[i].length

1 <= n <= 50

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

2

Each value

grid[i][j]

is

unique

.

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

```
}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $grid
* @return Integer
*/
```

```
function swimInWater($grid) {

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (swim-in-water grid)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Swim in Rising Water
* 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 swimInWater(vector<vector<int>>& grid) {

}
};
```

Java Solution:

```
/**
* Problem: Swim in Rising Water
* 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 swimInWater(int[][] grid) {

}
}
```

Python3 Solution:

```
"""
Problem: Swim in Rising Water
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 swimInWater(self, grid: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Swim in Rising Water
* 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 swimInWater = function(grid) {
```

```
};
```

TypeScript Solution:

```
/**
* Problem: Swim in Rising Water
* 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 swimInWater(grid: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Swim in Rising Water
* 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 SwimInWater(int[][] grid) {

}
}
```

C Solution:

```
/*
* Problem: Swim in Rising Water
* 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 swimInWater(int** grid, int gridSize, int* gridColSize) {

}
```

Go Solution:

```
// Problem: Swim in Rising Water
// 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 swimInWater(grid [][]int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Swim in Rising Water
// 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 swim_in_water(grid: Vec<Vec<i32>>) -> i32 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

```
}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (swim-in-water grid)
(-> (listof (listof exact-integer?)) exact-integer?)
)
```