Problem 1329: Sort the Matrix Diagonally

Problem Information

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

Problem Description

A

matrix diagonal

is a diagonal line of cells starting from some cell in either the topmost row or leftmost column and going in the bottom-right direction until reaching the matrix's end. For example, the

matrix diagonal

starting from

mat[2][0]

, where

mat

is a

6 x 3

matrix, includes cells

mat[2][0]

,

mat[3][1]

, and

mat[4][2]

.

Given an

m x n

matrix

mat

of integers, sort each

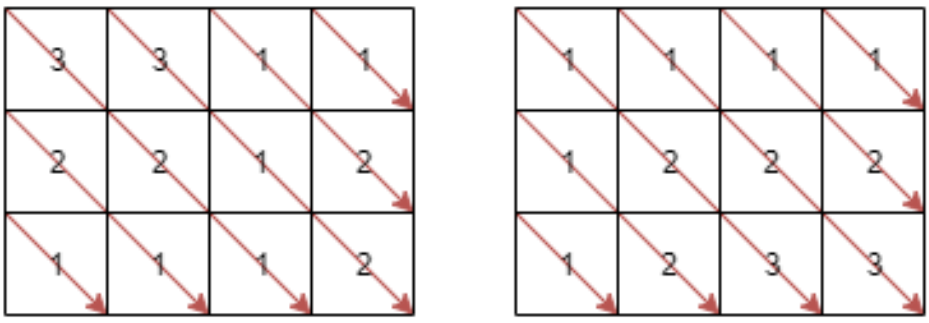
matrix diagonal

in ascending order and return

the resulting matrix

.

Example 1:



Input:

mat = [[3,3,1,1],[2,2,1,2],[1,1,1,2]]

Output:

[[1,1,1,1],[1,2,2,2],[1,2,3,3]]

Example 2:

Input:

mat = [[11,25,66,1,69,7],[23,55,17,45,15,52],[75,31,36,44,58,8],[22,27,33,25,68,4],[84,28,14,11,5,50]]

Output:

[[5,17,4,1,52,7],[11,11,25,45,8,69],[14,23,25,44,58,15],[22,27,31,36,50,66],[84,28,75,33,55,68]]

Constraints:

m == mat.length

n == mat[i].length

1 <= m, n <= 100

1 <= mat[i][j] <= 100

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

```
}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** diagonalSort(int** mat, int matSize, int* matColSize, int* returnSize,
int** returnColumnSizes) {

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

```
}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (diagonal-sort mat)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Sort the Matrix Diagonally
* Difficulty: Medium
* Tags: array, sort
*
* 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:
vector<vector<int>> diagonalSort(vector<vector<int>>& mat) {

}
};
```

Java Solution:

```
/**
* Problem: Sort the Matrix Diagonally
* Difficulty: Medium
* Tags: array, sort
*
* 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[][] diagonalSort(int[][] mat) {

}
}
```

Python3 Solution:

```
"""
Problem: Sort the Matrix Diagonally
Difficulty: Medium
Tags: array, sort

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Sort the Matrix Diagonally
* Difficulty: Medium
* Tags: array, sort
*
* 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[][]} mat
* @return {number[][]}
*/
var diagonalSort = function(mat) {

};
```

TypeScript Solution:

```
/**
* Problem: Sort the Matrix Diagonally
* Difficulty: Medium
* Tags: array, sort
*
* 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 diagonalSort(mat: number[][]): number[][] {

};
```

C# Solution:

```
/*
* Problem: Sort the Matrix Diagonally
* Difficulty: Medium
* Tags: array, sort
*
```

```
* 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[][] DiagonalSort(int[][] mat) {

}
}
```

C Solution:

```
/*
* Problem: Sort the Matrix Diagonally
* Difficulty: Medium
* Tags: array, sort
*
* 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
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** diagonalSort(int** mat, int matSize, int* matColSize, int* returnSize,
int** returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Sort the Matrix Diagonally
// Difficulty: Medium
// Tags: array, sort
//
// 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 diagonalSort(mat [][]int) [][]int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Sort the Matrix Diagonally
// Difficulty: Medium
// Tags: array, sort
//
// 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 diagonal_sort(mat: Vec<Vec<i32>>) -> Vec<Vec<i32>> {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
```

```
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (diagonal-sort mat)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?)))
)
```