Problem 3402: Minimum Operations to Make Columns Strictly Increasing

Problem Information

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

Problem Description

You are given a

m x n

matrix

grid

consisting of

non-negative

integers.

In one operation, you can increment the value of any

grid[i][j]

by 1.

Return the

minimum

number of operations needed to make all columns of

grid

strictly increasing

.

Example 1:

Input:

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

Output:

15

Explanation:

To make the

0

th

column strictly increasing, we can apply 3 operations on

grid[1][0]

, 2 operations on

grid[2][0]

, and 6 operations on

grid[3][0]

.

To make the

1

st

column strictly increasing, we can apply 4 operations on

grid[3][1]

.

3	2
1	3
3	4
0	1

Example 2:

Input:

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

Output:

12

Explanation:

To make the

0

th

column strictly increasing, we can apply 2 operations on

grid[1][0]

, and 4 operations on

grid[2][0]

.

To make the

1

st

column strictly increasing, we can apply 2 operations on

grid[1][1]

, and 2 operations on

grid[2][1]

.

To make the

2

nd

column strictly increasing, we can apply 2 operations on

grid[1][2]

.

3	2	1
2	1	0
1	2	3

Constraints:

m == grid.length

n == grid[i].length

1 <= m, n <= 50

0 <= grid[i][j] < 2500

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

```
/**
* @param {number[][]} grid
```

```
 * @return {number}
 */
var minimumOperations = function(grid) {

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

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

Solutions

C++ Solution:

```
/*
* Problem: Minimum Operations to Make Columns Strictly Increasing
* Difficulty: Easy
* Tags: array, greedy
*
* 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 minimumOperations(vector<vector<int>>& grid) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Operations to Make Columns Strictly Increasing
* Difficulty: Easy
* Tags: array, greedy
*
* 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 minimumOperations(int[][] grid) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Operations to Make Columns Strictly Increasing
Difficulty: Easy
Tags: array, greedy

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

```
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Minimum Operations to Make Columns Strictly Increasing
* Difficulty: Easy
* Tags: array, greedy
*
* 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 minimumOperations = function(grid) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Operations to Make Columns Strictly Increasing
* Difficulty: Easy
* Tags: array, greedy
*
* 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 minimumOperations(grid: number[][]): number {

};
```

C# Solution:

```
/*
* Problem: Minimum Operations to Make Columns Strictly Increasing
* Difficulty: Easy
* Tags: array, greedy
*
* 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 MinimumOperations(int[][] grid) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Operations to Make Columns Strictly Increasing
* Difficulty: Easy
* Tags: array, greedy
*
* 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 minimumOperations(int** grid, int gridSize, int* gridColSize) {

}
```

Go Solution:

```
// Problem: Minimum Operations to Make Columns Strictly Increasing
// Difficulty: Easy
// Tags: array, greedy
//
// 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 minimumOperations(grid [][]int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Minimum Operations to Make Columns Strictly Increasing
// Difficulty: Easy
// Tags: array, greedy
//
// 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_operations(grid: Vec<Vec<i32>>) -> i32 {

}
```

```
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

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