Problem 3651: Minimum Cost Path with Teleportations

Problem Information

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

Problem Description

You are given a

m x n

2D integer array

grid

and an integer

k

. You start at the top-left cell

(0, 0)

and your goal is to reach the bottom■right cell

(m - 1, n - 1)

.

There are two types of moves available:

Normal move

: You can move right or down from your current cell

(i, j)

, i.e. you can move to

(i, j + 1)

(right) or

(i + 1, j)

(down). The cost is the value of the destination cell.

Teleportation

: You can teleport from any cell

(i, j)

, to any cell

(x, y)

such that

grid[x][y] <= grid[i][j]

; the cost of this move is 0. You may teleport at most

k

times.

Return the

minimum

total cost to reach cell

(m - 1, n - 1)

from

(0, 0)

.

Example 1:

Input:

grid = [[1,3,3],[2,5,4],[4,3,5]], k = 2

Output:

7

Explanation:

Initially we are at (0, 0) and cost is 0.

Current Position

Move

New Position

Total Cost

(0, 0)

Move Down

(1, 0)

0 + 2 = 2

(1, 0)

Move Right

(1, 1)

2 + 5 = 7

(1, 1)

Teleport to

(2, 2)

(2, 2)

7 + 0 = 7

The minimum cost to reach bottom-right cell is 7.

Example 2:

Input:

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

Output:

9

Explanation:

Initially we are at (0, 0) and cost is 0.

Current Position

Move

New Position

Total Cost

(0, 0)

Move Down

(1, 0)

0 + 2 = 2

(1, 0)

Move Right

(1, 1)

2 + 3 = 5

(1, 1)

Move Down

(2, 1)

5 + 4 = 9

The minimum cost to reach bottom-right cell is 9.

Constraints:

2 <= m, n <= 80

m == grid.length

n == grid[i].length

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

4

0 <= k <= 10

Code Snippets

C++:

```
class Solution {
public:
int minCost(vector<vector<int>>& grid, int k) {

}
};
```

Java:

```
class Solution {
public int minCost(int[][] grid, int k) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function minCost(grid: number[][], k: number): number {

};
```

C#:

```
public class Solution {
public int MinCost(int[][] grid, int k) {

}
}
```

C:

```
int minCost(int** grid, int gridSize, int* gridColSize, int k) {

}
```

Go:

```
func minCost(grid [][]int, k int) int {

}
```

Kotlin:

```
class Solution {
fun minCost(grid: Array<IntArray>, k: Int): Int {

}
}
```

Swift:

```
class Solution {
func minCost(_ grid: [[Int]], _ k: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn min_cost(grid: Vec<Vec<i32>>, k: i32) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[][]} grid
# @param {Integer} k
# @return {Integer}
def min_cost(grid, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $grid
* @param Integer $k
* @return Integer
*/
function minCost($grid, $k) {

}
}
```

Dart:

```
class Solution {
int minCost(List<List<int>> grid, int k) {
```

```
}
}
```

Scala:

```
object Solution {
def minCost(grid: Array[Array[Int]], k: Int): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec min_cost(grid :: [[integer]], k :: integer) :: integer
def min_cost(grid, k) do

end
end
```

Erlang:

```
-spec min_cost(Grid :: [[integer()]], K :: integer()) -> integer().
min_cost(Grid, K) ->
.
```

Racket:

```
(define/contract (min-cost grid k)
(-> (listof (listof exact-integer?)) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Cost Path with Teleportations
* Difficulty: Hard
```

```
* Tags: array, dp
*
* 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 minCost(vector<vector<int>>& grid, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Cost Path with Teleportations
* Difficulty: Hard
* Tags: array, dp
*
* 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 minCost(int[][] grid, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Cost Path with Teleportations
Difficulty: Hard
Tags: array, dp

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Minimum Cost Path with Teleportations
* Difficulty: Hard
* Tags: array, dp
*
* 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[][]} grid
* @param {number} k
* @return {number}
*/
var minCost = function(grid, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Cost Path with Teleportations
* Difficulty: Hard
* Tags: array, dp
*
* 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 minCost(grid: number[][], k: number): number {

};
```

C# Solution:

```
/*
* Problem: Minimum Cost Path with Teleportations
* Difficulty: Hard
* Tags: array, dp
*
* 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 MinCost(int[][] grid, int k) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Cost Path with Teleportations
* Difficulty: Hard
* Tags: array, dp
*
* 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 minCost(int** grid, int gridSize, int* gridColSize, int k) {

}
```

Go Solution:

```
// Problem: Minimum Cost Path with Teleportations
// Difficulty: Hard
// Tags: array, dp
//
// 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 minCost(grid [][]int, k int) int {

}
```

Kotlin Solution:

```
class Solution {
fun minCost(grid: Array<IntArray>, k: Int): Int {

}
}
```

Swift Solution:

```
class Solution {
func minCost(_ grid: [[Int]], _ k: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Minimum Cost Path with Teleportations
// Difficulty: Hard
// Tags: array, dp
```

```
//
// 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 min_cost(grid: Vec<Vec<i32>>, k: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} grid
# @param {Integer} k
# @return {Integer}
def min_cost(grid, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $grid
* @param Integer $k
* @return Integer
*/
function minCost($grid, $k) {

}
}
```

Dart Solution:

```
class Solution {
int minCost(List<List<int>> grid, int k) {

}
}
```

Scala Solution:

```
object Solution {
def minCost(grid: Array[Array[Int]], k: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec min_cost(grid :: [[integer]], k :: integer) :: integer
def min_cost(grid, k) do

end
end
```

Erlang Solution:

```
-spec min_cost(Grid :: [[integer()]], K :: integer()) -> integer().
min_cost(Grid, K) ->
.
```

Racket Solution:

```
(define/contract (min-cost grid k)
(-> (listof (listof exact-integer?)) exact-integer? exact-integer?)
)
```