Problem 3603: Minimum Cost Path with Alternating Directions II

Problem Information

Difficulty: Medium
Acceptance Rate: 44.28%
Paid Only: No
Tags: Array, Dynamic Programming, Matrix

Problem Description

You are given two integers `m` and `n` representing the number of rows and columns of a grid, respectively.

The cost to enter cell `(i, j)` is defined as `(i + 1) * (j + 1)`.

You are also given a 2D integer array `waitCost` where `waitCost[i][j]` defines the cost to **wait** on that cell.

The path will always begin by entering cell `(0, 0)` on move 1 and paying the entrance cost.

At each step, you follow an alternating pattern:

* On **odd-numbered** seconds, you must move **right** or **down** to an **adjacent** cell, paying its entry cost. * On **even-numbered** seconds, you must **wait** in place for **exactly** one second and pay `waitCost[i][j]` during that second.

Return the **minimum** total cost required to reach `(m - 1, n - 1)`.

Example 1:

Input: m = 1, n = 2, waitCost = [[1,2]]

Output: 3

Explanation:

The optimal path is:

* Start at cell `(0, 0)` at second 1 with entry cost `(0 + 1) * (0 + 1) = 1`. * **Second 1** : Move right to cell `(0, 1)` with entry cost `(0 + 1) * (1 + 1) = 2`.

Thus, the total cost is `1 + 2 = 3`.

Example 2:

Input: m = 2, n = 2, waitCost = [[3,5],[2,4]]

Output: 9

Explanation:

The optimal path is:

* Start at cell `(0, 0)` at second 1 with entry cost `(0 + 1) * (0 + 1) = 1`. * **Second 1** : Move down to cell `(1, 0)` with entry cost `(1 + 1) * (0 + 1) = 2`. * **Second 2** : Wait at cell `(1, 0)`, paying `waitCost[1][0] = 2`. * **Second 3** : Move right to cell `(1, 1)` with entry cost `(1 + 1) * (1 + 1) = 4`.

Thus, the total cost is `1 + 2 + 2 + 4 = 9`.

Example 3:

Input: m = 2, n = 3, waitCost = [[6,1,4],[3,2,5]]

Output: 16

Explanation:

The optimal path is:

* Start at cell `(0, 0)` at second 1 with entry cost `(0 + 1) * (0 + 1) = 1`. * **Second 1** : Move right to cell `(0, 1)` with entry cost `(0 + 1) * (1 + 1) = 2`. * **Second 2** : Wait at cell `(0, 1)`, paying `waitCost[0][1] = 1`. * **Second 3** : Move down to cell `(1, 1)` with entry cost `(1 + 1) *

(1 + 1) = 4`. * **Second 4** : Wait at cell `(1, 1)`, paying `waitCost[1][1] = 2`. * **Second 5** : Move right to cell `(1, 2)` with entry cost `(1 + 1) * (2 + 1) = 6`.

Thus, the total cost is `1 + 2 + 1 + 4 + 2 + 6 = 16`.

Constraints:

* `1 <= m, n <= 105` * `2 <= m * n <= 105` * `waitCost.length == m` * `waitCost[0].length == n` * `0 <= waitCost[i][j] <= 105`

Code Snippets

C++:

```
class Solution {
public:
long long minCost(int m, int n, vector<vector<int>>& waitCost) {

}
};
```

Java:

```
class Solution {
public long minCost(int m, int n, int[][] waitCost) {

}
}
```

Python3:

```
class Solution:
def minCost(self, m: int, n: int, waitCost: List[List[int]]) -> int:
```