# Problem 3603: Minimum Cost Path with Alternating Directions II

## Problem Information

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

## 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 <= 10

5

2 <= m * n <= 10

5

waitCost.length == m

waitCost[0].length == n

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

5

## 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:
```

**Python:**

```
class Solution(object):
def minCost(self, m, n, waitCost):
"""
:type m: int
:type n: int
:type waitCost: List[List[int]]
:rtype: int
"""
```

**JavaScript:**

```
/**
* @param {number} m
* @param {number} n
* @param {number[][]} waitCost
* @return {number}
*/
var minCost = function(m, n, waitCost) {

};
```

**TypeScript:**

```
function minCost(m: number, n: number, waitCost: number[][]): number {

};
```

**C#:**

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

}
}
```

**C:**

```
long long minCost(int m, int n, int** waitCost, int waitCostSize, int*
waitCostColSize) {

}
```

**Go:**

```
func minCost(m int, n int, waitCost [][]int) int64 {

}
```

**Kotlin:**

```
class Solution {
fun minCost(m: Int, n: Int, waitCost: Array<IntArray>): Long {
```

```
}
}
```

**Swift:**

```
class Solution {
func minCost(_ m: Int, _ n: Int, _ waitCost: [[Int]]) -> Int {

}
}
```

**Rust:**

```
impl Solution {
pub fn min_cost(m: i32, n: i32, wait_cost: Vec<Vec<i32>>) -> i64 {

}
}
```

**Ruby:**

```
# @param {Integer} m
# @param {Integer} n
# @param {Integer[][]} wait_cost
# @return {Integer}
def min_cost(m, n, wait_cost)

end
```

**PHP:**

```
class Solution {

/**
* @param Integer $m
* @param Integer $n
* @param Integer[][] $waitCost
* @return Integer
*/
function minCost($m, $n, $waitCost) {
```

```
}
}
```

**Dart:**

```
class Solution {
int minCost(int m, int n, List<List<int>> waitCost) {

}
}
```

**Scala:**

```
object Solution {
def minCost(m: Int, n: Int, waitCost: Array[Array[Int]]): Long = {

}
}
```

**Elixir:**

```
defmodule Solution do
@spec min_cost(m :: integer, n :: integer, wait_cost :: [[integer]]) ::
integer
def min_cost(m, n, wait_cost) do

end
end
```

**Erlang:**

```
-spec min_cost(M :: integer(), N :: integer(), WaitCost :: [[integer()]]) ->
integer().
min_cost(M, N, WaitCost) ->
.
```

**Racket:**

```
(define/contract (min-cost m n waitCost)
(-> exact-integer? exact-integer? (listof (listof exact-integer?))
exact-integer?)
)
```

## Solutions

### C++ Solution:

```
/*
* Problem: Minimum Cost Path with Alternating Directions II
* Difficulty: Medium
* 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:
long long minCost(int m, int n, vector<vector<int>>& waitCost) {

}
};
```

### Java Solution:

```
/**
* Problem: Minimum Cost Path with Alternating Directions II
* Difficulty: Medium
* 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 long minCost(int m, int n, int[][] waitCost) {

}
}
```

### Python3 Solution:

```
"""
Problem: Minimum Cost Path with Alternating Directions II
Difficulty: Medium
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, m: int, n: int, waitCost: List[List[int]]) -> int:
# TODO: Implement optimized solution
pass
```

### Python Solution:

```
class Solution(object):
def minCost(self, m, n, waitCost):
"""
:type m: int
:type n: int
:type waitCost: List[List[int]]
:rtype: int
"""
```

### JavaScript Solution:

```
/**
* Problem: Minimum Cost Path with Alternating Directions II
* Difficulty: Medium
* 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} m
* @param {number} n
* @param {number[][]} waitCost
```

```
* @return {number}
*/
var minCost = function(m, n, waitCost) {

};
```

### TypeScript Solution:

```
/**
* Problem: Minimum Cost Path with Alternating Directions II
* Difficulty: Medium
* 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(m: number, n: number, waitCost: number[][]): number {

};
```

### C# Solution:

```
/*
* Problem: Minimum Cost Path with Alternating Directions II
* Difficulty: Medium
* 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 long MinCost(int m, int n, int[][] waitCost) {

}
}
```

### C Solution:

```
/*
* Problem: Minimum Cost Path with Alternating Directions II
* Difficulty: Medium
* 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
*/

long long minCost(int m, int n, int** waitCost, int waitCostSize, int*
waitCostColSize) {

}
```

### Go Solution:

```
// Problem: Minimum Cost Path with Alternating Directions II
// Difficulty: Medium
// 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(m int, n int, waitCost [][]int) int64 {

}
```

### Kotlin Solution:

```
class Solution {
fun minCost(m: Int, n: Int, waitCost: Array<IntArray>): Long {

}
}
```

### Swift Solution:

```
class Solution {
func minCost(_ m: Int, _ n: Int, _ waitCost: [[Int]]) -> Int {
```

```
}
}
```

**Rust Solution:**

```
// Problem: Minimum Cost Path with Alternating Directions II
// Difficulty: Medium
// 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(m: i32, n: i32, wait_cost: Vec<Vec<i32>>) -> i64 {

}
}
```

**Ruby Solution:**

```
# @param {Integer} m
# @param {Integer} n
# @param {Integer[][]} wait_cost
# @return {Integer}
def min_cost(m, n, wait_cost)

end
```

**PHP Solution:**

```
class Solution {

/**
* @param Integer $m
* @param Integer $n
* @param Integer[][] $waitCost
* @return Integer
*/
function minCost($m, $n, $waitCost) {
```

```
}
}
```

**Dart Solution:**

```
class Solution {
int minCost(int m, int n, List<List<int>> waitCost) {

}
}
```

**Scala Solution:**

```
object Solution {
def minCost(m: Int, n: Int, waitCost: Array[Array[Int]]): Long = {

}
}
```

**Elixir Solution:**

```
defmodule Solution do
@spec min_cost(m :: integer, n :: integer, wait_cost :: [[integer]]) ::
integer
def min_cost(m, n, wait_cost) do

end
end
```

**Erlang Solution:**

```
-spec min_cost(M :: integer(), N :: integer(), WaitCost :: [[integer()]]) ->
integer().
min_cost(M, N, WaitCost) ->
.
```

**Racket Solution:**

```
(define/contract (min-cost m n waitCost)
(-> exact-integer? exact-integer? (listof (listof exact-integer?))
exact-integer?)
```

```
)
```