

# Problem 2087: Minimum Cost Homecoming of a Robot in a Grid

## Problem Information

Difficulty: Medium

Acceptance Rate: 0.00%

Paid Only: No

## Problem Description

There is an

$m \times n$

grid, where

$(0, 0)$

is the top-left cell and

$(m - 1, n - 1)$

is the bottom-right cell. You are given an integer array

`startPos`

where

`startPos = [start`

`row`

`, start`

`col`

]

indicates that

initially

, a

robot

is at the cell

(start

row

, start

col

)

. You are also given an integer array

homePos

where

homePos = [home

row

, home

col

]

indicates that its

home

is at the cell

(home

row

, home

col

)

The robot needs to go to its home. It can move one cell in four directions:

left

,

right

,

up

, or

down

, and it can not move outside the boundary. Every move incurs some cost. You are further given two

0-indexed

integer arrays:

rowCosts

of length

m

and

colCosts

of length

n

.

If the robot moves

up

or

down

into a cell whose

row

is

r

, then this move costs

rowCosts[r]

.

If the robot moves

left

or

right

into a cell whose

column

is

c

, then this move costs

colCosts[c]

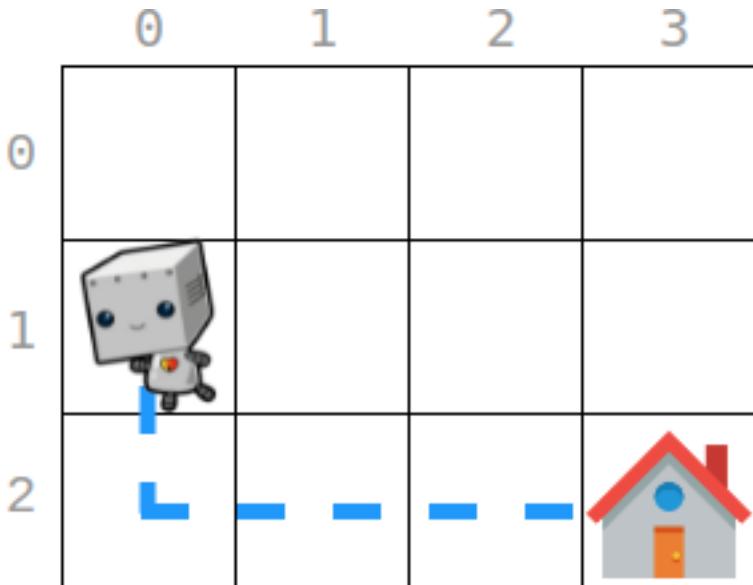
Return

the

minimum total cost

for this robot to return home

Example 1:



Input:

`startPos = [1, 0], homePos = [2, 3], rowCosts = [5, 4, 3], colCosts = [8, 2, 6, 7]`

Output:

18

Explanation:

One optimal path is that: Starting from (1, 0) -> It goes down to (

2

, 0). This move costs `rowCosts[2] = 3`. -> It goes right to (2,

1

). This move costs `colCosts[1] = 2`. -> It goes right to (2,

2

). This move costs `colCosts[2] = 6`. -> It goes right to (2,

3

). This move costs  $\text{colCosts}[3] = 7$ . The total cost is  $3 + 2 + 6 + 7 = 18$

Example 2:

Input:

`startPos = [0, 0], homePos = [0, 0], rowCosts = [5], colCosts = [26]`

Output:

0

Explanation:

The robot is already at its home. Since no moves occur, the total cost is 0.

Constraints:

`m == rowCosts.length`

`n == colCosts.length`

`1 <= m, n <= 10`

5

`0 <= rowCosts[r], colCosts[c] <= 10`

4

`startPos.length == 2`

`homePos.length == 2`

`0 <= start`

`row`

`, home`

row

< m

0 <= start

col

, home

col

< n

## Code Snippets

### C++:

```
class Solution {  
public:  
    int minCost(vector<int>& startPos, vector<int>& homePos, vector<int>&  
    rowCosts, vector<int>& colCosts) {  
  
    }  
};
```

### Java:

```
class Solution {  
    public int minCost(int[] startPos, int[] homePos, int[] rowCosts, int[]  
    colCosts) {  
  
    }  
}
```

### Python3:

```
class Solution:  
    def minCost(self, startPos: List[int], homePos: List[int], rowCosts:  
    List[int], colCosts: List[int]) -> int:
```

### **Python:**

```
class Solution(object):  
    def minCost(self, startPos, homePos, rowCosts, colCosts):  
        """  
        :type startPos: List[int]  
        :type homePos: List[int]  
        :type rowCosts: List[int]  
        :type colCosts: List[int]  
        :rtype: int  
        """
```

### **JavaScript:**

```
/**  
 * @param {number[]} startPos  
 * @param {number[]} homePos  
 * @param {number[]} rowCosts  
 * @param {number[]} colCosts  
 * @return {number}  
 */  
var minCost = function(startPos, homePos, rowCosts, colCosts) {  
  
};
```

### **TypeScript:**

```
function minCost(startPos: number[], homePos: number[], rowCosts: number[],  
    colCosts: number[]): number {  
  
};
```

### **C#:**

```
public class Solution {  
    public int MinCost(int[] startPos, int[] homePos, int[] rowCosts, int[]  
        colCosts) {  
  
    }  
}
```

### **C:**

```
int minCost(int* startPos, int startPosSize, int* homePos, int homePosSize,
int* rowCosts, int rowCostsSize, int* colCosts, int colCostsSize) {
}
```

### Go:

```
func minCost(startPos []int, homePos []int, rowCosts []int, colCosts []int)
int {
}
```

### Kotlin:

```
class Solution {
    fun minCost(startPos: IntArray, homePos: IntArray, rowCosts: IntArray,
    colCosts: IntArray): Int {
}
```

### Swift:

```
class Solution {
    func minCost(_ startPos: [Int], _ homePos: [Int], _ rowCosts: [Int], _ colCosts: [Int]) -> Int {
}
```

### Rust:

```
impl Solution {
    pub fn min_cost(start_pos: Vec<i32>, home_pos: Vec<i32>, row_costs: Vec<i32>,
    col_costs: Vec<i32>) -> i32 {
}
```

### Ruby:

```
# @param {Integer[]} start_pos
# @param {Integer[]} home_pos
```

```

# @param {Integer[]} row_costs
# @param {Integer[]} col_costs
# @return {Integer}
def min_cost(start_pos, home_pos, row_costs, col_costs)

end

```

### **PHP:**

```

class Solution {

    /**
     * @param Integer[] $startPos
     * @param Integer[] $homePos
     * @param Integer[] $rowCosts
     * @param Integer[] $colCosts
     * @return Integer
     */
    function minCost($startPos, $homePos, $rowCosts, $colCosts) {

    }
}

```

### **Dart:**

```

class Solution {
  int minCost(List<int> startPos, List<int> homePos, List<int> rowCosts,
  List<int> colCosts) {
    }
}

```

### **Scala:**

```

object Solution {
  def minCost(startPos: Array[Int], homePos: Array[Int], rowCosts: Array[Int],
  colCosts: Array[Int]): Int = {
    }
}

```

### **Elixir:**

```

defmodule Solution do
@spec min_cost(start_pos :: [integer], home_pos :: [integer], row_costs :: [integer], col_costs :: [integer]) :: integer
def min_cost(start_pos, home_pos, row_costs, col_costs) do

end
end

```

### Erlang:

```

-spec min_cost(StartPos :: [integer()], HomePos :: [integer()], RowCosts :: [integer()], ColCosts :: [integer()]) -> integer().
min_cost(StartPos, HomePos, RowCosts, ColCosts) ->
    .

```

### Racket:

```

(define/contract (min-cost startPos homePos rowCosts colCosts)
  (-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?)
        (listof exact-integer?) exact-integer?))

```

## Solutions

### C++ Solution:

```

/*
 * Problem: Minimum Cost Homecoming of a Robot in a Grid
 * Difficulty: Medium
 * 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 minCost(vector<int>& startPos, vector<int>& homePos, vector<int>& rowCosts, vector<int>& colCosts) {

```

```
}
```

```
};
```

### Java Solution:

```
/**  
 * Problem: Minimum Cost Homecoming of a Robot in a Grid  
 * Difficulty: Medium  
 * 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 minCost(int[] startPos, int[] homePos, int[] rowCosts, int[] colCosts) {  
        // Implementation  
    }  
}
```

### Python3 Solution:

```
"""  
Problem: Minimum Cost Homecoming of a Robot in a Grid  
Difficulty: Medium  
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 minCost(self, startPos: List[int], homePos: List[int], rowCosts: List[int], colCosts: List[int]) -> int:  
        # TODO: Implement optimized solution  
        pass
```

### Python Solution:

```

class Solution(object):
    def minCost(self, startPos, homePos, rowCosts, colCosts):
        """
        :type startPos: List[int]
        :type homePos: List[int]
        :type rowCosts: List[int]
        :type colCosts: List[int]
        :rtype: int
        """

```

### JavaScript Solution:

```

/**
 * Problem: Minimum Cost Homecoming of a Robot in a Grid
 * Difficulty: Medium
 * 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[]} startPos
 * @param {number[]} homePos
 * @param {number[]} rowCosts
 * @param {number[]} colCosts
 * @return {number}
 */
var minCost = function(startPos, homePos, rowCosts, colCosts) {

```

}

### TypeScript Solution:

```

/**
 * Problem: Minimum Cost Homecoming of a Robot in a Grid
 * Difficulty: Medium
 * 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 minCost(startPos: number[], homePos: number[], rowCosts: number[],
colCosts: number[]): number {
    };

```

### C# Solution:

```

/*
 * Problem: Minimum Cost Homecoming of a Robot in a Grid
 * Difficulty: Medium
 * 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 MinCost(int[] startPos, int[] homePos, int[] rowCosts, int[]
colCosts) {
        }
    }

```

### C Solution:

```

/*
 * Problem: Minimum Cost Homecoming of a Robot in a Grid
 * Difficulty: Medium
 * 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 minCost(int* startPos, int startPosSize, int* homePos, int homePosSize,
int* rowCosts, int rowCostsSize, int* colCosts, int colCostsSize) {

```

```
}
```

### Go Solution:

```
// Problem: Minimum Cost Homecoming of a Robot in a Grid
// Difficulty: Medium
// 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 minCost(startPos []int, homePos []int, rowCosts []int, colCosts []int)
int {

}
```

### Kotlin Solution:

```
class Solution {
    fun minCost(startPos: IntArray, homePos: IntArray, rowCosts: IntArray,
    colCosts: IntArray): Int {

    }
}
```

### Swift Solution:

```
class Solution {
    func minCost(_ startPos: [Int], _ homePos: [Int], _ rowCosts: [Int], _ colCosts: [Int]) -> Int {

    }
}
```

### Rust Solution:

```
// Problem: Minimum Cost Homecoming of a Robot in a Grid
// Difficulty: Medium
// 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 min_cost(start_pos: Vec<i32>, home_pos: Vec<i32>, row_costs: Vec<i32>,
                    col_costs: Vec<i32>) -> i32 {
        }

    }
}

```

### Ruby Solution:

```

# @param {Integer[]} start_pos
# @param {Integer[]} home_pos
# @param {Integer[]} row_costs
# @param {Integer[]} col_costs
# @return {Integer}
def min_cost(start_pos, home_pos, row_costs, col_costs)

end

```

### PHP Solution:

```

class Solution {

    /**
     * @param Integer[] $startPos
     * @param Integer[] $homePos
     * @param Integer[] $rowCosts
     * @param Integer[] $colCosts
     * @return Integer
     */
    function minCost($startPos, $homePos, $rowCosts, $colCosts) {

    }
}

```

### Dart Solution:

```

class Solution {
    int minCost(List<int> startPos, List<int> homePos, List<int> rowCosts,
    List<int> colCosts) {
        }
    }
}

```

### Scala Solution:

```

object Solution {
    def minCost(startPos: Array[Int], homePos: Array[Int], rowCosts: Array[Int],
    colCosts: Array[Int]): Int = {
        }
    }
}

```

### Elixir Solution:

```

defmodule Solution do
  @spec min_cost(start_pos :: [integer], home_pos :: [integer], row_costs :: [integer], col_costs :: [integer]) :: integer
  def min_cost(start_pos, home_pos, row_costs, col_costs) do
    end
  end
end

```

### Erlang Solution:

```

-spec min_cost(StartPos :: [integer()], HomePos :: [integer()], RowCosts :: [integer()],
  ColCosts :: [integer()]) -> integer().
min_cost(StartPos, HomePos, RowCosts, ColCosts) ->
  .

```

### Racket Solution:

```

(define/contract (min-cost startPos homePos rowCosts colCosts)
  (-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?)
    (listof exact-integer?) exact-integer?))

```