<https://leetcode.com/problems/minimum-cost-homecoming-of-a-robot-in-a-grid/description/>

There is an m x 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 = [startrow, startcol] indicates that **initially**, a **robot** is at the cell (startrow, startcol). You are also given an integer array homePos where homePos = [homerow, homecol] indicates that its **home** is at the cell (homerow, homecol).

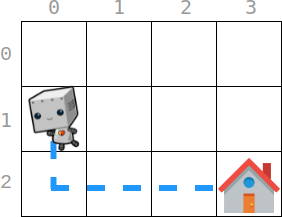
The robot needs to go to its home. It can move one cell in four directions: **left**, **right**, **up**, or **down**, and it can not move outside the boundary. Every move incurs some cost. You are further given two **0-indexed** integer arrays: rowCosts of length m and colCosts of length n.

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 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 <= 105

0 <= rowCosts[r], colCosts[c] <= 104

startPos.length == 2

homePos.length == 2

0 <= startrow, homerow < m

0 <= startcol, homecol < n

**Attempt 1: 2025-06-08**

**Solution 1: Greedy (10 min)**

class Solution {

    public int minCost(int[] startPos, int[] homePos, int[] rowCosts, int[] colCosts) {

        int totalCost = 0;

        int curRow = startPos[0];

        int curCol = startPos[1];

        int homeRow = homePos[0];

        int homeCol = homePos[1];

        while(curRow != homeRow) {

            if(curRow < homeRow) {

                curRow++;

            } else {

                curRow--;

            }

            totalCost += rowCosts[curRow];

        }

        while(curCol != homeCol) {

            if(curCol < homeCol) {

                curCol++;

            } else {

                curCol--;

            }

            totalCost += colCosts[curCol];

        }

        return totalCost;

    }

}

Time Complexity: O(m + n)

Space Complexity: O(1)

**Refer to**

**All shortest paths have the same cost**

<https://leetcode.com/problems/minimum-cost-homecoming-of-a-robot-in-a-grid/solutions/1598941/java-c-python-all-shortest-paths-have-the-same-cost/>

**Intuition**

It's a brain-teaser,

all shortest paths have the same cost.

**Prove**

We have 0 <= rowCosts[r], colCosts[c] <= 10^4,

that means we don't go duplicated path.

From the view of row index, the best path will be go directly from start x to home x

From the view of col index, the best path will be go directly from start y to home y

**Explanation**

Firstly move rows, from startPos[0] to homePos[0].

Secondly move cols, from startPos[1] to homePos[1].

Sum up the cost for every step.

**Complexity**

Time O(n+m)

Space O(1)

**Java**

public int minCost(int[] startPos, int[] homePos, int[] rowCosts, int[] colCosts) {

int res = 0, i = startPos[0], j = startPos[1], x = homePos[0], y = homePos[1];

while (i != x) {

i += (x - i) / Math.abs(x - i);

res += rowCosts[i];

}

while (j != y) {

j += (y - j) / Math.abs(y - j);

res += colCosts[j];

}

return res;

}

**Refer to Deepseek**

This problem requires calculating the minimum cost for a robot to return home from its starting position in a grid, moving only right or down, with specific row and column costs.

**Approach**

We'll use a greedy approach since:

The robot can only move right or down

The path is deterministic (must move toward home in both directions)

Costs are additive based on movement direction

**Solution Code**

class Solution {

public int minCost(int[] startPos, int[] homePos, int[] rowCosts, int[] colCosts) {

int totalCost = 0;

int currentRow = startPos[0];

int currentCol = startPos[1];

int homeRow = homePos[0];

int homeCol = homePos[1];

// Move vertically toward home row

while (currentRow != homeRow) {

if (currentRow < homeRow) {

currentRow++;

} else {

currentRow--;

}

totalCost += rowCosts[currentRow];

}

// Move horizontally toward home column

while (currentCol != homeCol) {

if (currentCol < homeCol) {

currentCol++;

} else {

currentCol--;

}

totalCost += colCosts[currentCol];

}

return totalCost;

}

}

**Explanation**

**Vertical Movement**:

Determine direction (up or down) based on start/home positions

Move one row at a time, accumulating row costs

Stop when reaching home row

**Horizontal Movement**:

Determine direction (left or right) based on start/home positions

Move one column at a time, accumulating column costs

Stop when reaching home column

**Cost Calculation**:

Only costs of passed rows/columns are added (start position cost excluded)

Order of movement doesn't affect total cost (can do horizontal first if preferred)

**Time and Space Complexity**

**Time Complexity**: O(m + n) where m = row distance, n = col distance

**Space Complexity**: O(1) - constant extra space used

This solution efficiently computes the minimum cost by leveraging the grid's structure and movement constraints. The greedy approach works perfectly here since there's only one optimal path given the movement restrictions.

**Native DFS Solution and Memoization Solution for Robot Homecoming Problem**

While the greedy approach is optimal for this problem, here's how a native DFS solution would look for completeness:

**Void return Native DFS Solution (TLE 14/69)**

class Solution {

private int minCost = Integer.MAX\_VALUE;

public int minCost(int[] startPos, int[] homePos, int[] rowCosts, int[] colCosts) {

// No cost if already at home

if (startPos[0] == homePos[0] && startPos[1] == homePos[1]) {

return 0;

}

dfs(startPos[0], startPos[1], homePos, rowCosts, colCosts, 0);

return minCost;

}

private void dfs(int row, int col, int[] homePos, int[] rowCosts, int[] colCosts, int currentCost) {

// Base case: reached home

if (row == homePos[0] && col == homePos[1]) {

minCost = Math.min(minCost, currentCost);

return;

}

// Prune if current path is already worse than best found

if (currentCost >= minCost) {

return;

}

// Try moving right (if not at or beyond home column)

if (col < homePos[1]) {

dfs(row, col + 1, homePos, rowCosts, colCosts, currentCost + colCosts[col + 1]);

}

// Try moving left (if beyond home column)

else if (col > homePos[1]) {

dfs(row, col - 1, homePos, rowCosts, colCosts, currentCost + colCosts[col - 1]);

}

// Try moving down (if not at or beyond home row)

if (row < homePos[0]) {

dfs(row + 1, col, homePos, rowCosts, colCosts, currentCost + rowCosts[row + 1]);

}

// Try moving up (if beyond home row)

else if (row > homePos[0]) {

dfs(row - 1, col, homePos, rowCosts, colCosts, currentCost + rowCosts[row - 1]);

}

}

}

Time Complexity: is exponential O(2^(m+n)) where m and n are distances to home

**Key Problems with DFS Approach**

**Inefficiency**:

**Time complexity is exponential O(2^(m+n)) where m and n are distances to home**

Will time out on medium/large grids

**Redundant Paths**:

Explores many suboptimal paths that could be pruned

No memoization benefit since each path is unique

**Unnecessary Complexity**:

The problem has a deterministic optimal path (greedy solution works perfectly)

DFS overcomplicates a simple problem

**Why Greedy is Better**

**Optimal Path**:

There's exactly one shortest path (moving directly toward home)

No need to explore alternatives

**Efficiency**:

O(m+n) time complexity

Only computes the necessary moves

**Simplicity**:

Straightforward implementation

Easy to understand and maintain

**Recommendation**

For this specific problem, always prefer the greedy approach. The DFS solution is provided only to demonstrate how it would look, but it's not practical for actual submission due to performance constraints.

The greedy solution is:

Correct - always finds the minimal cost path

Optimal - has the best possible time complexity

Simple - easy to implement and understand

**Integer-Return Native DFS Solution for Robot Homecoming (TLE 14/69)**

Here's a corrected integer-return DFS solution that properly handles all movement directions and avoids array index issues:

class Solution {

public int minCost(int[] startPos, int[] homePos, int[] rowCosts, int[] colCosts) {

// Base case: already at home

if (startPos[0] == homePos[0] && startPos[1] == homePos[1]) {

return 0;

}

return dfs(startPos[0], startPos[1], homePos, rowCosts, colCosts);

}

private int dfs(int row, int col, int[] homePos, int[] rowCosts, int[] colCosts) {

// Base case: reached home

if (row == homePos[0] && col == homePos[1]) {

return 0;

}

int minCost = Integer.MAX\_VALUE;

// Try moving right if needed

if (col < homePos[1]) {

int rightCost = colCosts[col + 1] + dfs(row, col + 1, homePos, rowCosts, colCosts);

minCost = Math.min(minCost, rightCost);

}

// Try moving left if needed

else if (col > homePos[1]) {

int leftCost = colCosts[col - 1] + dfs(row, col - 1, homePos, rowCosts, colCosts);

minCost = Math.min(minCost, leftCost);

}

// Try moving down if needed

if (row < homePos[0]) {

int downCost = rowCosts[row + 1] + dfs(row + 1, col, homePos, rowCosts, colCosts);

minCost = Math.min(minCost, downCost);

}

// Try moving up if needed

else if (row > homePos[0]) {

int upCost = rowCosts[row - 1] + dfs(row - 1, col, homePos, rowCosts, colCosts);

minCost = Math.min(minCost, upCost);

}

return minCost;

}

}

Time Complexity: is exponential O(2^(m+n)) where m and n are distances to home

**Key Improvements:**

**Proper Integer Return**:

Returns the actual cost instead of void

More natural recursive structure

**Correct Movement Logic**:

Properly separates row and column comparisons

Uses homePos[0] for rows and homePos[1] for columns

**Base Cases**:

Handles case when already at home position

Returns 0 when reaching home

**Bounds Safety**:

Only makes valid moves that don't go out of bounds

**Performance Considerations:**

**Exponential Complexity**:

Still O(2^(m+n)) where m and n are distances to home

Will time out on medium/large grids

**Memoization Potential**:

Could be optimized with memoization (though greedy is still better)

**Redundant Calculations**:

Recomputes the same subproblems repeatedly

**Memoized version for Integer-Return Native DFS Solution for Robot Homecoming (TLE 54/69)**

class Solution {

public int minCost(int[] startPos, int[] homePos, int[] rowCosts, int[] colCosts) {

int m = rowCosts.length;

int n = colCosts.length;

Integer[][] memo = new Integer[m][n];

return dfs(startPos[0], startPos[1], homePos, rowCosts, colCosts, memo);

}

private int dfs(int row, int col, int[] homePos, int[] rowCosts, int[] colCosts, Integer[][] memo) {

if (row == homePos[0] && col == homePos[1]) {

return 0;

}

if (memo[row][col] != null) {

return memo[row][col];

}

int minCost = Integer.MAX\_VALUE;

if (col < homePos[1]) {

int cost = colCosts[col + 1] + dfs(row, col + 1, homePos, rowCosts, colCosts, memo);

minCost = Math.min(minCost, cost);

}

else if (col > homePos[1]) {

int cost = colCosts[col - 1] + dfs(row, col - 1, homePos, rowCosts, colCosts, memo);

minCost = Math.min(minCost, cost);

}

if (row < homePos[0]) {

int cost = rowCosts[row + 1] + dfs(row + 1, col, homePos, rowCosts, colCosts, memo);

minCost = Math.min(minCost, cost);

}

else if (row > homePos[0]) {

int cost = rowCosts[row - 1] + dfs(row - 1, col, homePos, rowCosts, colCosts, memo);

minCost = Math.min(minCost, cost);

}

memo[row][col] = minCost;

return minCost;

}

}

Time Complexity: O(m \* n) where m and n are distances to home

Space Complexity: O(m \* n)

**Recommendation:**

While these DFS solutions work, for this specific problem the greedy approach remains the best choice due to its O(m+n) time complexity and simplicity. The DFS solutions are provided mainly for educational purposes to demonstrate recursive thinking.

**Time Complexity Analysis of Memoized DFS Solution**

For the memoized DFS solution to the Robot Homecoming problem, the time complexity is: **O(m × n)** *(where m = number of rows, n = number of columns in the grid)*

**Why?**

**Memoization ensures each cell is computed only once**

The memo[row][col] table stores results for each (row, col) pair.

Once computed, subsequent calls return the stored value in **O(1)** time.

**Each cell is visited at most once**

The robot moves only **right/down** (or left/up if needed), meaning it doesn’t revisit cells in a way that would cause recomputation.

**Work per cell is constant (O(1))**

Each DFS call performs a few comparisons and arithmetic operations.

The recursive calls are bounded by the memo table.

**Comparison with Unmemoized DFS**

|  |  |  |
| --- | --- | --- |
| Approach | Time Complexity | Space Complexity |
| **Naive DFS** | O(2^(m+n)) | O(m + n) (recursion stack) |
| **Memoized DFS** | **O(m × n)** | O(m × n) (memo table) |
| **Greedy**  (Optimal) | O(m + n) | O(1) |

**When Memoization Helps**

**Worst-case:** The robot must traverse all m rows and n columns, but memoization ensures no redundant work.

**Best-case:** If the robot is already at homePos, it returns in **O(1)**.

**Why Greedy is Still Better**

The **memoized DFS** is **O(m × n)** time and space.

The **greedy solution** is **O(m + n)** time and **O(1)** space (just loops through rows and columns).

Since the robot’s path is **deterministic**, the greedy approach is optimal.

**Final Answer**

The **memoized DFS solution has a time complexity of O(m × n)** due to filling the m × n memo table, where each cell is computed only once. However, the **greedy solution remains superior** with **O(m + n)** time and **O(1)** space.

**Use memoized DFS** only if exploring DFS concepts—otherwise, **prefer the greedy approach** for efficiency.

**Refer to**

[L62.Unique Paths (Ref.L63,L64,L174,L2087,L2304,L2400)](note://E33F0B728ABE4CCF8B7E554A219CF1EA)