2087. Minimum Cost Homecoming of a Robot in a Grid

Problem Description

In this LeetCode problem, we are given a grid of size `m x n`, with the top-left cell at `(0, 0)` and the bottom-right cell at `(m - 1, n - 1)`. A robot is initially located at the cell `startPos = [start_row, start_col]`, and its goal is to reach its home located at `homePos = [home_row, home_col]`. The robot can move in four directions from any cell - left, right, up, or down - while staying inside the grid boundaries.

The crux of the problem is to calculate the minimum total cost for the robot to return home. The costs of moving through the grid are given by two arrays: `rowCosts` and `colCosts`. Moving to a cell in a different row incurs a cost from `rowCosts` equal to the value of the destination row, and moving to a cell in a different column incurs a cost from `colCosts` equal to the value of the destination column.

The task is to find and return this minimum cost.

Intuition

The key insight to solve this problem is realizing that the robot can move in a straight line to its destination, without any detours, because there are no obstacles or restrictions on its path other than the grid boundaries. The cost incurred by the robot depends only on the cells it passes through, particularly their row and column indices.

Hence, determining the minimum total cost can be done by separately calculating the cost for moving in the vertical direction (up or down, whichever is required to reach `home_row`) and the cost for moving in the horizontal direction (left or right, to reach `home_col`).

For the vertical movement, if `startPos[0]` (the start row) is less than `homePos[0]` (the home row), the robot needs to move down, incurring the sum of `rowCosts` between these two rows. Conversely, if it is greater, the robot moves up, summing up the corresponding `rowCosts` in reverse order.

Similarly, for the horizontal movement, we check if `startPos[1]` (the start column) is less than `homePos[1]` (the home column), indicating a move to the right with the sum of `colCosts` between these columns. If greater, the robot moves left, summing the `colCosts` from home to start.

The solution approach consists of two sums in each necessary direction – one for row movement and one for column movement. Finally, adding both sums gives us the total minimum cost required by the robot to reach its home.

Solution Approach

The provided solution approach is straightforward and directly translates the intuition into code. It utilizes simple algorithmic concepts, relying on direct access to array elements and summing up slices of the arrays based on certain conditions. No complex data structures or patterns are needed for this approach, making it a perfect example of an efficient brute-force solution. Here's a step-by-step explanation of the code:

1. First, we destructure the starting and home positions into their respective row and column indices: `i, j` for starting and `x, y` for home positions.
2. We initialize `ans` to zero to accumulate the total cost.
3. For vertical movement:
   - If the robot is below its home row (`start_row < home_row`), we sum `rowCosts` from the row just below the start row up to and including the home row, as the robot needs to move down.
   - Otherwise (`start_row >= home_row`), we sum `rowCosts` from the home row up to but not including the start row, as the robot moves up.
4. For horizontal movement:
   - If the robot is to the left of its home column (`start_col < home_col`), we sum `colCosts` from the column just to the right of the start column up to and including the home column, as the robot needs to move right.
   - Conversely, if the robot is to the right (`start_col >= home_col`), we sum `colCosts` from the home column up to but not including the start column, as the robot moves left.
5. We add the two sums from step 3 and step 4 to get the total cost, which we assign to `ans`.
6. We return the value computed in `ans` as this is the minimum cost for the robot to reach its home position.

The essential algorithmic concepts used here are conditionals to determine the direction of the robot's movement and array slicing with the built-in `sum()` function to calculate the movement's cost.

```
ans = 0
if i < x:
    ans += sum(rowCosts[i + 1 : x + 1])
else:
    ans += sum(rowCosts[x:i])
if j < y:
    ans += sum(colCosts[j + 1 : y + 1])
else:
    ans += sum(colCosts[y:j])
```

This block of code succinctly captures the logic required to solve the problem. The use of array slicing in Python makes for an elegant solution that is not only efficient but also easy to read and understand.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Assume we have a grid represented by its size `m x n`, and for our example, let's take `m = 3` and `n = 3`, so we have a 3×3 grid. Let's say the `startPos` is `[1, 1]`, and the `homePos` is `[2, 2]`. Also, let's say `rowCosts = [1, 2, 3]` and `colCosts = [4, 5, 6]`.

The robot starts at position `(1, 1)`, which corresponds to the second row and second column of the grid (since the grid indices start at 0), and wants to move to its home at `(2, 2)`.

Following the solution approach:

1. We destructure the positions into their indices:
   - `i, j` for start position: i = 1, j = 1
   - `x, y` for home position: x = 2, y = 2
2. Initialize `ans` to zero.
3. For vertical movement:
   - `i < x`: This is true (1 < 2), so the robot moves down.
   - We sum up the `rowCosts` from the row just below the start row up to the home row, which is `rowCosts[2]` since we don't need to sum a range here.
   - `ans += rowCosts[2]`, which is `ans += 3`.
4. For horizontal movement:
   - `j < y`: This is also true (1 < 2), which means the robot moves to the right.
   - We sum up the `colCosts` from the column just to the right of the start column up to the home column, which again, is `colCosts[2]`, with no range to sum.
   - `ans += colCosts[2]`, which is `ans += 6`.
5. Now, we add the two sums to get `ans = 3 (from rowCosts) + 6 (from colCosts) = 9`.
6. We return this `ans` value, which is the minimum total cost for the robot to move from its starting position to its home position on this grid. The minimum cost is 9.

In conclusion, the robot would incur a cost of 9 to move from `[1, 1]` to `[2, 2]`, with a `rowCosts` of `[1, 2, 3]` and `colCosts` of `[4, 5, 6]`. The simplicity of this algorithm lies in its straightforward calculation of moving costs: we only consider the costs along the robot's path to its destination.

Python Solution

```
from typing import List

class Solution:
    def minCost(self,
                start_pos: List[int],
                home_pos: List[int],
                row_costs: List[int],
                col_costs: List[int]) -> int:
        # Initialize start position (i, j) and target home position (x, y)
        i, j = start_pos
        x, y = home_pos
        total_cost = 0  # Variable to store the total cost

        # Calculate the row movement cost
        if i < x:
            # If start row is less than home row, sum the costs from next of start row to home row
            total_cost += sum(row_costs[i + 1 : x + 1])
        else:
            # If start row is greater than or equal to home row, sum the costs from home row to the row before start row
            total_cost += sum(row_costs[x:i])

        # Calculate the column movement cost
        if j < y:
            # If start column is less than home column, sum the costs from next of start column to home column
            total_cost += sum(col_costs[j + 1 : y + 1])
        else:
            # If start column is greater than or equal to home column, sum the costs from home column to the column before start column
            total_cost += sum(col_costs[y:j])

        return total_cost  # Return the calculated total cost
```

Java Solution

```
class Solution {
    public int minCost(int[] startPos, int[] homePos, int[] rowCosts, int[] colCosts) {
        // Initialize variables with starting positions.
        int currentRow = startPos[0], currentCol = startPos[1];
        // Destination positions.
        int targetRow = homePos[0], targetCol = homePos[1];
        // Variable to keep track of the total minimum cost.
        int totalCost = 0;

        // If currentRow is less than targetRow, move down.
        if (currentRow < targetRow) {
            for (int row = currentRow + 1; row <= targetRow; ++row) {
                totalCost += rowCosts[row]; // Add cost for each row moved.
            }
        } else {
            // If currentRow is more than targetRow, move up.
            for (int row = targetRow; row < currentRow; ++row) {
                totalCost += rowCosts[row]; // Add cost for each row moved.
            }
        }

        // If currentCol is less than targetCol, move right.
        if (currentCol < targetCol) {
            for (int col = currentCol + 1; col <= targetCol; ++col) {
                totalCost += colCosts[col]; // Add cost for each column moved.
            }
        } else {
            // If currentCol is more than targetCol, move left.
            for (int col = targetCol; col < currentCol; ++col) {
                totalCost += colCosts[col]; // Add cost for each column moved.
            }
        }

        // Return the accumulated total cost.
        return totalCost;
    }
}
```

C++ Solution

```
#include <vector>
#include <numeric> // include necessary library for std::accumulate

class Solution {
public:
    // Method to calculate the minimum cost to move from the start position to the home position.
    int minCost(std::vector<int>& startPos, std::vector<int>& homePos, std::vector<int>& rowCosts, std::vector<int>& colCosts) {
        // Extract start and home positions into readable variables
        int currentRow = startPos[0], currentCol = startPos[1];
        int targetRow = homePos[0], targetCol = homePos[1];
        int totalCost = 0; // Initialize total cost to be accumulated

        // Move vertically and accumulate row costs
        if (currentRow < targetRow) {
            // Moving down: sum the costs from the row just below the current row to the target row (inclusive)
            totalCost += std::accumulate(rowCosts.begin() + currentRow + 1, rowCosts.begin() + targetRow + 1, 0);
        } else {
            // Moving up: sum the costs from the target row to the row just above the current row (exclusive)
            totalCost += std::accumulate(rowCosts.begin() + targetRow, rowCosts.begin() + currentRow, 0);
        }

        // Move horizontally and accumulate column costs
        if (currentCol < targetCol) {
            // Moving right: sum the costs from the column just right of the current column to the target column (inclusive)
            totalCost += std::accumulate(colCosts.begin() + currentCol + 1, colCosts.begin() + targetCol + 1, 0);
        } else {
            // Moving left: sum the costs from the target column to the column just left of the current column (exclusive)
            totalCost += std::accumulate(colCosts.begin() + targetCol, colCosts.begin() + currentCol, 0);
        }

        // Return the total calculated cost
        return totalCost;
    }
};
```

Typescript Solution

```
// Import array manipulation functions, since std is not available in TypeScript
// We would typically rely on native array methods in TypeScript

// Function to calculate the minimum cost to move from the start position to the home position
function minCost(startPos: number[], homePos: number[], rowCosts: number[], colCosts: number[]): number {
    // Extract start and home positions into readable variables
    const startRow = startPos[0];
    const startColumn = startPos[1];
    const targetRow = homePos[0];
    const targetColumn = homePos[1];
    let totalCost = 0; // Initialize total cost to be accumulated

    // Function to sum the elements of an array within a specified range
    const sumRange = (costs: number[], start: number, end: number): number => {
        let sum = 0;
        for (let i = start; i <= end; i++) {
            sum += costs[i];
        }
        return sum;
    };

    // Move vertically and accumulate row costs
    if (startRow < targetRow) {
        // Moving down: sum the costs from the row just below the start row to the target row (inclusive)
        totalCost += sumRange(rowCosts, startRow + 1, targetRow);
    } else {
        // Moving up: sum the costs from the target row to the row just above the start row (exclusive)
        totalCost += sumRange(rowCosts, targetRow, startRow - 1);
    }

    // Move horizontally and accumulate column costs
    if (startColumn < targetColumn) {
        // Moving right: sum the costs from the column just right of the start column to the target column (inclusive)
        totalCost += sumRange(colCosts, startColumn + 1, targetColumn);
    } else {
        // Moving left: sum the costs from the target column to the column just left of the start column (exclusive)
        totalCost += sumRange(colCosts, targetColumn, startColumn - 1);
    }

    // Return the total calculated cost
    return totalCost;
}

// Usage of the minCost function would involve calling it with appropriate arguments:
// const cost = minCost([startX, startY], [homeX, homeY], rowCostsArray, colCostsArray)
```

Time and Space Complexity

The given Python function computes the minimum cost to move from a starting position to a home position on a grid, given the costs of moving through each row and column.

Time Complexity

The time complexity of the code is determined primarily by the `sum` calls for row and column movements.

- The first `sum` operation to calculate the row costs is $O(n)$ if $x > i$ or $O(m)$ if $x < i$, where n is the number of rows from `i + 1` to `x + 1` and m is the number of rows from `x` to `i`.
- The second `sum` operation to calculate the column costs is $O(p)$ if $y > j$ or $O(q)$ if $y < j$, where p is the number of columns from `j + 1` to `y + 1` and q is the number of columns from `y` to `j`.

However, since each row and each column is found in the `rowCosts` and `colCosts` lists respectively only once, at most, we would perform a single pass through each list. Consequently, the overall time complexity is $O(R + C)$, where R is the number of rows (length of `rowCosts`) and C is the number of columns (length of `colCosts`).

Space Complexity

The space complexity of the function is $O(1)$ (or constant space) because the space usage does not grow with the size of the input. The function only uses a fixed number of integer variables to compute the result, and there are no data structures that grow with the input size.