2120. Execution of All Suffix Instructions Staying in a Grid

Medium String Simulation Leetcode Link

Problem Description

In this problem, we are given a scenario where we have an `n x n` grid, and a robot is situated at a starting position on this grid. The robot's position is defined by the `startPos`, which is an array of two integers representing the row and column. Additionally, we are provided with a string `s` of length `m`, representing a sequence of instructions (`'L'`, `'R'`, `'U'`, `'D'`) that tell the robot how to move within the grid.

The task is to find out, for each instruction starting from the `ith` position in the string `s`, how many instructions the robot can execute before one of two conditions occurs:

1. The robot is instructed to move off the grid (out of bounds).
2. The robot reaches the end of the instruction string without moving off the grid.

The final output should be an array `answer` where each element `answer[i]` represents the number of instructions the robot can execute when starting from the `i`th instruction in `s`.

Intuition

The solution to this problem lies in simulating the path of the robot's movement on the grid from each possible starting point in the instruction string `s`.

For each instruction starting point `i`, we simulate the robot's movement by iterating over instructions from `s[i]` to `s[m - 1]` and tracking its current position. We use a dictionary `mp` to translate the instruction characters (`'L'`, `'R'`, `'U'`, `'D'`) into corresponding row and column movements.

With each step, we:

- Check if the next move stays within the grid boundaries.
- If it does, we update the robot's position and increment the counter for executed instructions.
- If the next move would take the robot out of bounds, we stop and consider the number of executed instructions as the result for the starting point `i`.

We repeat this process for every instruction as a potential starting point, obtaining the total number of instructions the robot can execute for each case. These counts are added to the `ans` list in the order of instruction starting points.

The approach relies on a brute-force simulation for each starting point, which simplifies the implementation and ensures correctness by considering every possible execution path from every starting point.

Solution Approach

The provided solution approach utilizes the simulation strategy, where we follow the robot's instructions from each possible start point in the instruction string `s` until it either moves off the grid or runs out of instructions. Here are the steps and the constructs used in the implementation:

Algorithm:

1. **Initialize an empty list (`ans`)** to collect the number of executable instructions for each starting point.
2. **Create a mapping (`mp`)** which translates instruction characters (`'L'`, `'R'`, `'U'`, `'D'`) to their corresponding movements on the grid as row (`x`) and column (`y`) changes.
3. **Loop through each instruction `i` as a start point** in the instruction string `s`. For each start point:

   a. **Initial Position**: Set the robot's current position (`x`, `y`) to the starting position (`startPos`).

   b. **Execution Counter (`t`)**: Initialize a counter to keep track of how many instructions have been executed from that starting point.
4. **Iterate over the instructions from the current start point `i` to the end of the string.** For each instruction `j`:

   a. Retrieve the corresponding movement from the `mp` dictionary.

   b. **Check if the movement is within grid bounds**:

   - If the resulting position (`x + a`, `y + b`) is within the grid (0 <=x < `n` and 0 <= `y` < `n`), apply the movement to the current position and increment the `t` counter.
   - If the move would go off-grid, **break out of the loop** since the robot cannot execute this instruction.

   c. **After the loop**, append the value of `t` (the number of instructions executed) to the `ans` list.
5. Once we have simulated starting from each instruction and collected the executable instruction counts, return `ans`.

Data Structures:

- **List**: To store the final count of executable instructions (`ans`) and the starting position (`startPos`).
- **Dictionary**: To translate instructions into actual x and y movements (`mp`).

Patterns:

- **Simulation**: This problem is directly addressed through simulatory execution of instructions, mimicking the exact behavior we are trying to query, thus avoiding the need for more complex algorithms or optimizations.

By using this brute-force simulation, the solution effectively handles any given `n x n` grid and instruction set `s`, ensuring a correct and complete answer.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach described above.

Suppose we have a grid of size `3 x 3`, and the robot starts at `startPos = [1,1]` (the center of the grid). Imagine our string of instructions, `s`, is `"RUL"`. We want to find out how many instructions the robot can execute starting at each character of the instruction string before moving off the grid.

Now let's simulate according to the algorithm:

1. **Initialize an empty list (`ans`)** to store results.
2. **Create a mapping (`mp`)** of `{'L': (0, -1), 'R': (0, 1), 'U': (-1, 0), 'D': (1, 0)}` to represent movements.
3. **Loop through each `i` in `s`** — here, `i` will take values `0` for `'R'`, `1` for `'U'`, and `2` for `'L'`.

For `i = 0` ('R'):

- Start from `startPos` (1,1).
- Move right `(0, 1)` to (1,2). Grid boundary not exceeded, increment `t` to 1.
- Next is `'U'`, move up `(-1, 0)` to (0,2). Grid boundary not exceeded, increment `t` to 2.
- Last is `'L'`, move left `(0, -1)` to (0,1). Grid boundary not exceeded, increment `t` to 3.
- Add `t = 3` to `ans`.

For `i = 1` ('U'):

- Start from `startPos` (1, 1).
- Move up `(-1, 0)` to (0, 1). Grid boundary not exceeded, increment `t` to 1.
- Next is `'L'`, move left `(0, -1)` to (0,0). Grid boundary not exceeded, increment `t` to 2.
- There are no more instructions, so add `t = 2` to `ans`.

For `i = 2` ('L'):

- Start from `startPos` (1, 1).
- Move left `(0, -1)` to (1,0). Grid boundary not exceeded, increment `t` to 1.
- No more instructions, so add `t = 1` to `ans`.

After the loop, we have: `ans = [3, 2, 1]`.

4. **Return `ans`**, which gives us the count of executable instructions for the robot starting from each position in the string `s`.

So, the output would be `[3, 2, 1]` for our example, indicating that starting from the first instruction 'R', the robot can execute 3 instructions; from 'U', 2 instructions; and from 'L', 1 instruction before going off the grid or running out of instructions.

Python Solution

```python
from typing import List

class Solution:
    def execute_instructions(self, n: int, start_pos: List[int], instructions: str) -> List[int]:
        # Initialize the result list
        results = []

        # Length of the instruction string
        num_instructions = len(instructions)

        # Mapping for each instruction to its corresponding row and column changes
        direction_map = {"L": [0, -1], "R": [0, 1], "U": [-1, 0], "D": [1, 0]}

        # Iterate over each instruction starting position
        for i in range(num_instructions):
            # Starting position for this sequence of instructions
            x, y = start_pos

            # Count of valid moves from the current starting instruction
            valid_moves = 0

            # Execute instructions starting at index i and onwards
            for j in range(i, num_instructions):
                # Get the row and column changes for the current instruction
                row_change, col_change = direction_map[instructions[j]]

                # Update the current position based on the instruction
                x += row_change
                y += col_change

                # Check if the new position is still within bounds
                if 0 <= x < n and 0 <= y < n:
                    # Increase the count of valid moves
                    valid_moves += 1
                else:
                    # If out of bounds, no more valid moves, break the loop
                    break

            # Add the count of valid moves for this sequence to the results list
            results.append(valid_moves)

        # Return the list of valid moves for each starting point
        return results

```

Java Solution

```java
class Solution {
    public int[] executeInstructions(int n, int[] startPos, String instructions) {
        int instructionsLength = instructions.length(); // Length of the instructions string
        int[] results = new int[instructionsLength]; // Array to store the result for each position
        // A map to associate directions with their respective 2D coordinate changes
        Map<Character, int[]> directionMap = new HashMap<>(4);
        directionMap.put('L', new int[] {0, -1});// Left move
        directionMap.put('R', new int[] {0, 1}); // Right move
        directionMap.put('U', new int[] {-1, 0});// Up move
        directionMap.put('D', new int[] {1, 0}); // Down move

        // Loop through each position in the instructions as a starting point
        for (int start = 0; start < instructionsLength; ++start) {
            int x = startPos[0], y = startPos[1]; // Current position to check from
            int validMoves = 0; // Counter to count the valid moves from the current starting position

            // Loop through the instructions starting from the current position
            for (int current = start; current < instructionsLength; ++current) {
                char currentInstruction = instructions.charAt(current); // Get the current instruction
                // Retrieve the 2D coordinate change for the current direction
                int deltaX = directionMap.get(currentInstruction)[0], deltaY = directionMap.get(currentInstruction)[1];

                // Check if the move within the bounds of the grid
                if (0 <= x + deltaX && x + deltaX < n && 0 <= y + deltaY && y + deltaY < n) {
                    x += deltaX; // Update the current x-coordinate
                    y += deltaY; // Update the current y-coordinate
                    ++validMoves; // Increment the valid move counter
                } else {
                    // A move took us out of the grid bounds, stop checking further instructions
                    break;
                }
            }
            results[start] = validMoves; // Store the valid moves for this start position
        }
        return results; // Return the array containing valid moves for each start position
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_map>
using namespace std;

class Solution {
public:
    vector<int> executeInstructions(int n, vector<int>& startPos, string s) {
        int instructionCount = s.size(); // number of instructions to execute
        vector<int> answer(instructionCount); // vector to hold the number of valid instructions for each index
        unordered_map<char, vector<int>> directionMap; // map to hold the direction vectors
        // Direction vectors for L, R, U, D
        directionMap['L'] = {0, -1};
        directionMap['R'] = {0, 1};
        directionMap['U'] = {-1, 0};
        directionMap['D'] = {1, 0};

        // Iterate over each instruction to see how many steps we can take starting from that instruction
        for (int i = 0; i < instructionCount; ++i) {
            int posX = startPos[0]; // current X position
            int posY = startPos[1]; // current Y position
            int validSteps = 0; // counter for the number of valid steps we can take for current instruction

            // Starting from the i-th instruction, execute instructions until out of bounds
            for (int j = i; j < instructionCount; ++j) {
                // Get the move's direction vector (dx, dy) based on the j-th instruction
                int dx = directionMap[s[j]][0];
                int dy = directionMap[s[j]][1];

                // Check if the new position after the move is within bounds
                if (0 <= posX + dx && posX + dx < n && 0 <= posY + dy && posY + dy < n) {
                    // Update position and increment valid step count
                    posX += dx;
                    posY += dy;
                    ++validSteps;
                } else {
                    // If the move is out of bounds, break the loop
                    break;
                }
            }
            // Store the number of valid steps for the i-th starting instruction
            answer[i] = validSteps;
        }
        return answer; // return the vector with the results
    }
};

```

Typescript Solution

```typescript
// This function takes a grid size (n), a starting position (startPos),
// and a string of instructions (s) and calculates the number of valid positions
// the robot can move to following the instructions starting at each position within the string.
function executeInstructions(gridSize: number, startPos: number[], instructions: string): number[] {
    const instructionsLength = instructions.length;
    const answerArray = new Array(instructionsLength);

    // Loop through each instruction in the string.
    for (let i = 0; i < instructionsLength; i++) {
        // Initialize the robot's current position to the starting position.
        let [currentRow, currentCol] = startPos;
        let stepCount: number;

        // Iterate over the remaining instructions from the current position.
        for (stepCount = i; stepCount < instructionsLength; stepCount++) {
            const currentInstruction = instructions[stepCount];

            // Update the current position based on the instruction.
            if (currentInstruction === 'U') {
                currentRow--;
            } else if (currentInstruction === 'D') {
                currentRow++;
            } else if (currentInstruction === 'L') {
                currentCol--;
            } else {
                currentCol++;
            }

            // If the new position is out of bounds, stop processing further instructions.
            if (currentRow === -1 || currentRow === gridSize || currentCol === -1 || currentCol === gridSize) {
                break;
            }
        }

        // The number of valid steps taken is the difference between the current
        // step count and the starting step index for this round of instructions.
        answerArray[i] = stepCount - i;
    }

    // Return the array containing the number of valid steps for each starting position in the string.
    return answerArray;
}

```

Time and Space Complexity

The provided Python code implements a function that calculates the number of valid instructions from each starting point in the string `s`. For calculation, it simulates the movement on an `n x n` grid based on the instructions given in `s`.

- Time Complexity: The outer loop runs `m` times, where `m` is the length of the string `s`. An inner loop runs for each character in `s` starting at the position `i`, where `i` is the index of the outer loop. In the worst case, the inner loop runs `m` times (when `i` is 0), and as `i` increases, the number of iterations decreases. Hence, the number of total operations can be approximated by the sum of the first `m` natural numbers. This sum is (m*(m+1))/2 which has a complexity of `O(m^2)`.

  The precise sum of operations for the inner loop is:

  ```
  m + (m - 1) + (m - 2) + ... + 2 + 1 = m*(m+1)/2
  ```

  Therefore, the time complexity is `O(m^2)`.

- Space Complexity: The space complexity is determined by the storage used by the algorithm besides the input. Here, the variables used (`ans`, `mp`, `x`, `y`, `a`, `b`, `t`) do not grow with the size of the input `n` or the length of the string `s`; they either are constants or only store single values. The `ans` list, however, grows with the length of `s` since an entry is added for each character in `s`.

  Thus, space complexity is `O(m)` because the only variable that scales with input size is the `ans` list, with `m` being the length of the input string `s`.