657. Robot Return to Origin

Easy String Simulation

Leetcode Link

Problem Description

This LeetCode problem involves determining whether a robot, which starts at the origin (0, 0) on a 2D plane, returns to its starting point after executing a series of moves. The series of moves is given as a string where each character corresponds to a move in one of four possible directions: 'R' (right), 'L' (left), 'U' (up), and 'D' (down). The task is to analyze this string and return `true` if the robot ends up at the origin after all the moves or `false` otherwise. The key point is to keep track of the robot's position relative to the origin, regardless of how it's facing, and verify if it returns to (0, 0).

Intuition

The solution is based on the simple idea of simulating the robot's moves by keeping track of its position on the 2D plane with a pair of variables, one for the horizontal axis (x) and the other for the vertical axis (y). Initially, the robot is at (0, 0). For each move, depending on the direction, we increment or decrement the corresponding axis value - 'R' increases x, 'L' decreases x, 'U' increases y, and 'D' decreases y. After processing all moves, if the robot's position is still (0, 0), it means the robot has returned to the starting point, and we return `true`; otherwise, `false`.

This approach relies on the observation that for the robot to return to the origin, it must have performed equal numbers of 'L' and 'R' moves as well as equal numbers of 'U' and 'D' moves. This pair of equalities ensures that the robot has undone all horizontal and vertical displacements, respectively.

Solution Approach

The implementation of the solution uses a straightforward, iterative approach to simulate the moves of the robot. Here is a step-by-step explanation of the algorithm and the data structures used:

1. Initialize two variables, `x` and `y`, both set to `0`. These variables represent the position of the robot on the x-axis (horizontal) and y-axis (vertical) respectively.
2. Loop through each character in the `moves` string. This string is the sequence of moves the robot is to perform.
3. For each character (representing a move), compare it to the possible moves 'R', 'L', 'U', 'D':
   - If the move is 'R' (right), increment the `x` variable to simulate a move to the right.
   - If the move is 'L' (left), decrement the `x` variable to simulate a move to the left.
   - If the move is 'U' (up), increment the `y` variable to simulate a move upward.
   - If the move is 'D' (down), decrement the `y` variable to simulate a move downward.
4. After the loop completes, all the moves have been simulated, and the robot's final position has been updated accordingly.
5. Finally, check if both `x` and `y` are `0`. If they are, it means the robot returned to the origin after all its moves. Therefore, return `true`. If either `x` or `y` is not `0`, return `false`.

The algorithm uses constant extra space, only needing two integer variables, regardless of the length of the `moves` string. The time complexity is linear with respect to the length of the `moves` string since it has to check each move exactly once.

No additional data structures or complex patterns are needed. The solution is optimal in terms of both time and space complexity, which are O(n) and O(1), respectively, with `n` being the length of the moves string.

Example Walkthrough

Let's consider an example where the moves string given is `"UDLR"`. According to the problem description, we need to simulate these moves and determine if the robot returns to the origin `(0, 0)` afterwards.

1. We initialize two variables, `x` and `y`, to `0`. These represent the robot's current position on the x and y axes, respectively: `(x, y) = (0, 0)`.
2. We start looping through the `moves` string.
3. The first character is `'U'`. This indicates a move up. According to our algorithm, this means we increment `y`.
   - New position: `(x, y) = (0, 1)`
4. The next character is `'D'`. This move is down, so we decrement `y`.
   - New position: `(x, y) = (0, 0)`. The robot has returned to the origin, but we continue to process all moves.
5. The third character is `'L'`. The robot moves left, resulting in the decrement of `x`.
   - New position: `(x, y) = (-1, 0)`
6. The last character is `'R'`. This is a move to the right, which means we increment `x`.
   - Final position: `(x, y) = (0, 0)`
7. We have completed the loop and processed all the moves. The robot's final position is at the origin, where it started.
8. Since both `x` and `y` are `0`, our algorithm returns `true`. This indicates that the robot indeed returned to its starting point after executing the given series of moves.

In this walk-through, we can see how the algorithm effectively uses the position variables to keep track of the robot's location. It's clear that for every move away from the origin, there is a counter move that brings the robot back to the origin, thus restoring both `x` and `y` to `0`.

Python Solution

```python
class Solution:
    def judgeCircle(self, moves: str) -> bool:
        # Initialize coordinates at the origin point (0,0)
        horizontal_position = 0
        vertical_position = 0

        # Iterate through each move in the string
        for move in moves:
            # Move right: increase horizontal position
            if move == 'R':
                horizontal_position += 1
            # Move left: decrease horizontal position
            elif move == 'L':
                horizontal_position -= 1
            # Move up: increase vertical position
            elif move == 'U':
                vertical_position += 1
            # Move down: decrease vertical position
            elif move == 'D':
                vertical_position -= 1

        # If both positions are back to 0, return True (circle complete)
        return horizontal_position == 0 and vertical_position == 0

# The function 'judgeCircle' analyzes a sequence of moves (R, L, U, D)
# and determines if they form a circle leading back to the origin (0,0).

```

Java Solution

```java
class Solution {
    // Method to judge whether the robot returns to the origin after executing a sequence of moves
    public boolean judgeCircle(String moves) {
        // Initial position of the robot
        int x = 0; // Horizontal axis displacement
        int y = 0; // Vertical axis displacement

        // Loop through the moves string to process each move
        for (int i = 0; i < moves.length(); ++i) {
            // Get the current move
            char move = moves.charAt(i);

            // Process the move to calculate the displacement
            if (move == 'R') { // Right move: increase x-coordinate
                x++;
            } else if (move == 'L') { // Left move: decrease x-coordinate
                x--;
            } else if (move == 'U') { // Up move: increase y-coordinate
                y++;
            } else if (move == 'D') { // Down move: decrease y-coordinate
                y--;
            }
            // Note: No 'else' case since we assume all characters in 'moves' are valid (only 'R', 'L', 'U', 'D')
        }

        // Check if the robot returned to the origin (0,0)
        return x == 0 && y == 0;
    }
}

```

C++ Solution

```cpp
#include <string>
#include <unordered_map>

// Function to determine if a series of moves for a robot
// in a plane starting at the origin (0,0) will result in the robot
// returning to the original position
bool judgeCircle(std::string moves) {
    // Variables to hold the robot's x and y coordinates,
    // initialized to the start position (0, 0)
    int xCoordinate = 0;
    int yCoordinate = 0;

    // Creating a map for defining the movements associated
    // with each possible direction
    const std::unordered_map<char, std::pair<int, int>> directions {
        {'R', {1, 0}},  // Right move increases x coordinate by 1
        {'L', {-1, 0}}, // Left move decreases x coordinate by 1
        {'U', {0, 1}},  // Up move increases y coordinate by 1
        {'D', {0, -1}}  // Down move decreases y coordinate by 1
    };

    // Iterate over each move in the string
    for (char move : moves) {
        // Look up the movement associated with the direction
        const auto& delta = directions.at(move);
        // Update the robot's position coordinates
        xCoordinate += delta.first;
        yCoordinate += delta.second;
    }

    // Check if the robot is back at the origin (0,0)
    // If so, return true, otherwise return false
    return xCoordinate == 0 && yCoordinate == 0;
}

```

Typescript Solution

```typescript
// TypeScript function to determine if a series of moves for a robot
// in a plane starting at the origin (0,0) will result in the robot
// returning to the original position
function judgeCircle(moves: string): boolean {
    // Variables to hold the robot's position coordinates,
    // initialized to the start position (0,0)
    let xCoordinate = 0,
        yCoordinate = 0;

    // Object representing the possible directions and their respective movements
    const directions = {
        'R': [1, 0],   // Right move (x coordinate +1)
        'L': [-1, 0],  // Left move (x coordinate -1)
        'U': [0, 1],   // Up move (y coordinate +1)
        'D': [0, -1],  // Down move (y coordinate -1)
    };

    // Iterate over each move in the input string
    for (let move of moves) {
        // Get the directional movements for the current move
        const [deltaX, deltaY] = directions[move];
        // Update the robot's position coordinates
        xCoordinate += deltaX;
        yCoordinate += deltaY;
    }

    // Return true if the robot is back at the origin (0,0), false otherwise
    return xCoordinate === 0 && yCoordinate === 0;
}

```

Time and Space Complexity

The time complexity of the given code can be analyzed as follows: The function `judgeCircle` iterates through each character in the input string `moves`. The number of operations per character is constant (either increment or decrement operations). Therefore, the time complexity is directly proportional to the length of `moves`, which gives us a time complexity of `O(n)`, where `n` is the number of moves.

The space complexity of the code is also easy to determine: the only extra space used are the two integer variables `x` and `y`, which store the coordinates. These variables use a constant amount of space regardless of the input size, thus the space complexity is `O(1)`, denoting constant space usage.