1041. Robot Bounded In Circle

Medium Math String Simulation

Problem Description

The problem involves a robot that operates on an infinite plane and can only make a limited set of movements. It starts at the coordinates `(0, 0)` and faces north. The robot can:

- Move forward one unit ('G').
- Make a 90-degree left turn ('L').
- Make a 90-degree right turn ('R').

The robot repeats a sequence of these instructions infinitely. The main question is to determine whether the robot's movements are bounded within a circle - meaning that no matter how many times it repeats the instructions, it will never venture infinitely far away from the starting point.

Intuition

To solve this problem, the intuition lies in recognizing the pattern of the robot's movement after it completes the sequence of instructions once. If the robot returns to the `(0, 0)` point facing north, it will repeat the path exactly and is thus bounded within a circle.

However, if the robot does not return to the `(0, 0)` point but ends facing north, it will move further away with each repetition. Thus, it's not bounded by a circle.

If it ends up in any direction other than north, we know that after a maximum of 4 sequences, it will be back facing north (since there are 4 possible directions and a turn changes the direction by one). By this time, if it has not returned to the starting point, the path it takes would form a loop that repeats, hence it is bounded.

The solution code creates a direction variable `k` that indicates the current facing direction of the robot and an array `dist` that keeps the net distance moved in each direction (north, east, south, and west). After processing each instruction, there are two possible cases that define if the robot is bounded:

1. `dist[0] == dist[2] and dist[1] == dist[3]`: This means the net distance moved north equals the net distance moved south, and the net distance moved east equals the net distance moved west, indicating it's back at the starting point facing north.
2. `k != 0`: This indicates that the robot is facing a different direction than it started with, after completing the sequence. Hence, in this case, it's guaranteed to be bounded by a circle after enough repetitions.

Solution Approach

The implementation of the solution involves a step-by-step process to determine the robot's status after executing one sequence of instructions:

1. **Initialize Direction and Distance:** A variable `k` is set to 0, representing the robot's direction (0 for north, 1 for east, 2 for south, 3 for west), and a list `dist` is initialized with four zeros to track the net distance moved in each of the four directions.
2. **Process Instructions:** The function iterates through each character in the `instructions` string:
   - If it encounters 'L', it turns the robot left by incrementing `k` and then taking modulo 4. This ensures `k` remains in the range `[0, 3]`.
   - If it encounters 'R', it turns the robot right by decrementing `k` (add 3 before modulo to simulate the rotation right), then taking modulo 4.
   - If it encounters 'G', it increments the distance by 1 in the direction the robot is currently facing, which is updated in the `dist` list.
3. **Check for Boundedness:** After processing all instructions, the robot's boundedness is determined using two checks:
   - `dist[0] == dist[2] and dist[1] == dist[3]`: This check compares the net north-south and east-west movements. If these are equal, it means the robot has returned to the initial point `(0, 0)` and is therefore bounded.
   - `k != 0`: This check sees if the robot is facing a different direction than the initial one. If it is not facing north, it implies the robot is on a path that will repeat and form a loop, hence it is bounded.

By using an array to track the net movement in each cardinal direction and a variable for direction, the solution leverages simple mathematical properties and modular arithmetic to determine the robot's path characteristics in an efficient manner. This approach ensures that we are only concerned with the state of the robot after one sequence, greatly simplifying the problem which otherwise might seem to require simulating an infinite number of instruction sequences.

In short, the solution leverages the following algorithms and data structures:

- A loop to iterate over the instructions.
- An array to maintain net distances in four cardinal directions.
- Modular arithmetic for direction updates ensuring the direction stays within bounds.
- Conditional logic to assert the robot's boundedness.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider that the instructions string given to our robot is `"GGLLGG"`.

1. **Initialize Direction and Distance:** Let's initialize the robot's direction `k` to 0 (facing north) and the list `dist` to `[0, 0, 0, 0]`, tracking the net distance moved in each cardinal direction: north, east, south, and west, respectively.
2. **Process Instructions:**
   - First, the robot sees 'G' and moves forward one unit north. The `dist` list becomes `[1, 0, 0, 0]`.
   - Then, another 'G' means it moves one more unit north. The `dist` list is now `[2, 0, 0, 0]`.
   - Next, the instruction 'L' tells the robot to turn left (west), so we increment `k` by 1 and take modulo 4, giving us `k = 1`.
   - Another 'L' instruction turns the robot left from west to south, updating `k` to 2 after incrementation and modulo.
   - The robot encounters 'G', moving one unit south. The `dist` array updates to `[2, 0, 1, 0]`.
   - The final 'G' means another move south, resulting in `dist` being `[2, 0, 2, 0]`.
3. **Check for Boundedness:**
   - We check if `dist[0] == dist[2] and dist[1] == dist[3]`. Indeed, `2 == 2 and 0 == 0`, which means the robot has returned to the initial point `(0, 0)` facing south (`k = 2`).
   - We also check if `k != 0`. Since `k` is `2`, it indeed is not equal to `0`, which implies that the robot did not end facing north.

From the checks, we conclude that the robot does not keep moving away infinitely from the starting point. Even if the instructions are repeated infinitely, the robot will always return to `(0, 0)` after each set of instructions, creating a looped path. Thus, it is bounded within an invisible circle defined by its path.

Solution Implementation

Python

```python
class Solution:
    def isRobotBounded(self, instructions: str) -> bool:
        # Initialize the direction index, 0 = North, 1 = West, 2 = South, 3 = East
        direction_index = 0

        # Initialize distances for all four directions.
        # index corresponds to the direction: 0 = North, 1 = West, 2 = South, 3 = East
        distances = [0, 0, 0, 0]

        # Loop through each character in the instructions string
        for command in instructions:
            if command == 'L':
                # Turn left: Update direction index counter-clockwise
                direction_index = (direction_index + 1) % 4
            elif command == 'R':
                # Turn right: Update direction index clockwise
                direction_index = (direction_index + 3) % 4
            else:  # command == 'G'
                # Move forward: Increment distance in the current facing direction
                distances[direction_index] += 1

        # Check if the robot returns to the original position (circular) OR
        # is facing in a different direction other than North after one cycle of instructions
        # This ensures that repeating the instructions will eventually bring it back to the origin.
        is_circular = (distances[0] == distances[2] and distances[1] == distances[3]) or direction_index != 0

        return is_circular
```

Java

```java
class Solution {
    public boolean isRobotBounded(String instructions) {
        // 'directionIndex' represents the current direction of the robot:
        // 0-Up, 1-Left, 2-Down, 3-Right
        int directionIndex = 0;

        // 'distance' array stores the net distance moved in all 4 directions.
        int[] distance = new int[4];

        // Iterate over each instruction and update the direction or distance accordingly.
        for (int i = 0; i < instructions.length(); ++i) {
            char currentInstruction = instructions.charAt(i);

            // If the instruction is 'L', it turns the robot 90 degrees to the left.
            if (currentInstruction == 'L') {
                directionIndex = (directionIndex + 1) % 4;
            }
            // If the instruction is 'R', it turns the robot 90 degrees to the right.
            else if (currentInstruction == 'R') {
                directionIndex = (directionIndex + 3) % 4;
            }
            // If the instruction is 'G', the robot moves forward one unit in the current direction.
            else {
                distance[directionIndex]++;
            }
        }

        // The robot is bounded if:
        // 1. It returns to the initial position (distance moved up equals down and left equals right).
        // 2. It is not facing north after processing all instructions.
        return (distance[0] == distance[2] && distance[1] == distance[3]) || (directionIndex != 0);
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to determine if the robot is bounded in a circle.
    bool isRobotBounded(string instructions) {
        // Define an array to store the distance moved in each of the four directions (North, East, South, West).
        int distance[4] = {0};
        // 'direction' represents the current direction of the robot: 0 - North, 1 - East, 2 - South, 3 - West.
        int direction = 0;

        // Iterate over the given instruction string.
        for (char& instruction : instructions) {
            // Change direction based on the current instruction.
            if (instruction == 'L') {
                direction = (direction + 1) % 4; // Turning left.
            } else if (instruction == 'R') {
                direction = (direction + 3) % 4; // Turning right.
            } else { // 'instruction' must be 'G' at this point, which means go straight.
                // Increment the distance in the current direction.
                ++distance[direction];
            }
        }

        // The robot is bounded in a circle if it returns to the origin,
        // or if it doesn't face north at the end of the sequence.
        // Returning to the origin means having the same number of moves in opposite directions.
        return (distance[0] == distance[2] && distance[1] == distance[3]) || direction != 0;
    }
};
```

TypeScript

```typescript
function isRobotBounded(instructions: string): boolean {
    // Array to keep track of distance moved in 4 directions:
    // north (index 0), east (index 1), south (index 2), and west (index 3).
    const distances: number[] = new Array(4).fill(0);

    // Variable 'direction' to keep track of current direction:
    // 0 = north, 1 = east, 2 = south, 3 = west.
    let direction = 0;

    // Loop through each instruction character
    for (const instruction of instructions) {
        if (instruction === 'L') {
            // Turn left from current direction
            direction = (direction + 1) % 4;
        } else if (instruction === 'R') {
            // Turn right from current direction (3 left turns)
            direction = (direction + 3) % 4;
        } else {
            // Move one unit in the current direction
            ++distances[direction];
        }
    }

    // Check if robot is back to the original position (x = 0 and y = 0)
    // or it is not facing north. Facing not north means the loop will return to the origin
    // in another one or more sequences of instructions.
    return (distances[0] === distances[2] && distances[1] === distances[3]) || direction !== 0;
}
```

```typescript
class Solution:
    def isRobotBounded(self, instructions: str) -> bool:
        # Initialize the direction index, 0 = North, 1 = West, 2 = South, 3 = East
        direction_index = 0

        # Initialize distances for all four directions.
        # index corresponds to the direction: 0 = North, 1 = West, 2 = South, 3 = East
        distances = [0, 0, 0, 0]

        # Loop through each character in the instructions string
        for command in instructions:
            if command == 'L':
                # Turn left: Update direction index counter-clockwise
                direction_index = (direction_index + 1) % 4
            elif command == 'R':
                # Turn right: Update direction index clockwise
                direction_index = (direction_index + 3) % 4
            else:  # command == 'G'
                # Move forward: Increment distance in the current facing direction
                distances[direction_index] += 1

        # Check if the robot returns to the original position (circular) OR
        # is facing in a different direction other than North after one cycle of instructions
        # This ensures that repeating the instructions will eventually bring it back to the origin.
        is_circular = (distances[0] == distances[2] and distances[1] == distances[3]) or direction_index != 0

        return is_circular
```

Time and Space Complexity

The code snippet presented is designed to determine if a set of instructions will cause a robot to end up in a circle or not. Here is the complexity analysis:

Time Complexity

The time complexity of the algorithm is determined by the number of instructions that need to be processed. The for-loop iterates through each character in the `instructions` string once.

- The for-loop runs for n iterations, where n is the length of the `instructions`.
- Each operation within the for-loop (checking the character and updating `k` or `dist`) is a constant time operation.
- Therefore, the time complexity is $O(n)$, where n is the length of the `instructions` string.

Space Complexity

The space complexity of the algorithm is determined by the amount of extra space used irrespective of the input size.

- A fixed-size list `dist` of 4 integers is used to keep track of the distances moved in each of the four directions.
- Variable `k` is used to keep track of the current direction, which also uses a constant amount of space.
- As there is no dynamic data structure that grows with the input size and only a fixed amount of extra space is used, the space complexity is $O(1)$, which means it is constant.