2833. Furthest Point From Origin

Easy | Array | Counting

Problem Description

In this problem, we are given a string `moves` that represents a sequence of movements along a number line. This string is composed of only three types of characters: `'L'`, `'R'`, and `'_'`. Each character in the string represents a potential movement in a specific direction:

- `'L'` represents a move to the left.
- `'R'` represents a move to the right.
- `'_'` represents a move that can be chosen to be either left or right.

We are asked to determine the furthest distance from the origin (point 0 on the number line) we can reach after performing all the moves. The furthest distance is calculated as how far one could be from the origin in either direction, so we need to consider the greatest possible deviation from the starting point.

Intuition

The problem asks for the furthest distance possible from the origin rather than the final position after all moves. Intuitively, the furthest one can be from the origin is by combining the absolute difference between the moves to the left `'L'` and the moves to the right `'R'` along with the total number of `'_'` moves. This is because `'_'` moves provide us with flexibility, and we can assume they can be used to move in the direction that increases this difference.

Here is the reasoning broken down by each type of move:

- For each `'L'` move, the position moves one unit to the left.
- For each `'R'` move, the position moves one unit to the right.
- For each `'_'` move, we do not know whether it should be left or right for maximum distance, so we can count it separately.

At the end, the furthest distance from the origin can be calculated as:

- The absolute difference between the count of `'L'` and `'R'` which gives us how far we could be if the `'_'` characters did not contribute to the distance. This models the rigid moves.
- Plus the count of `'_'` because they can be used to extend the distance further in the direction that we have more moves (`'L'` or `'R'`).

Solution Approach

The implementation uses Python's built-in method `count()` to calculate the number of moves for each type: `'L'` for left moves, `'R'` for right moves, and `'_'` for the flexible moves. Using these counts, the solution takes advantage of the fact that the final distance can be determined by two components:

1. The absolute difference between left and right moves:
   - This tells us the net movement in one specific direction. If there are more `'L'` than `'R'` moves, this will be a negative number representing the distance to the left of the origin, and vice versa. The `abs()` function is applied to this difference to get the positive equivalent of this net movement because we're only interested in distance, not direction.
2. The total number of '_' moves:
   - Since these moves are flexible, they can be added directly to the difference calculated in step 1. This accounts for the maximum possible extension to the distance we can reach from the origin. Here we assume that all `'_'` moves are used optimally to move further in the direction that already has a net movement.

The formula for the *"furthest distance from the origin"* is thus:

```
furthest_distance = abs(count('L') - count('R')) + count('_')
```

Following this approach, the solution does not need to iterate through the string for each move or calculate the position after each move. Instead, it treats the problem comprehensively, counting the moves of each type and then combining them to immediately return the furthest possible distance.

Here is the Python function for reference:

```python
class Solution:
    def furthestDistanceFromOrigin(self, moves: str) -> int:
        # Calculating the net number of moves to the left or right by finding the absolute difference
        # between the number of 'L' and 'R' in the given moves string.
        net_directional_moves = abs(moves.count("L") - moves.count("R"))

        # Adding the count of '_' to the net directional moves to get the furthest distance possible,
        # since '_' can represent a move in either direction.
        furthest_distance = net_directional_moves + moves.count("_")

        return furthest_distance
```

This code returns the maximum distance achievable after all `moves`. Because it performs a fixed set of operations unrelated to the size of the input string, this algorithm runs in O(n) time, where n is the length of the `moves` string, due to the underlying implementation of the `count()` function.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we are given the `moves` string `"L_LRR_"`.

Here's how we walk through this string according to the solution approach:

1. Count the number of each type of move:
   - Number of `'L'` moves: 2 (the first and third characters).
   - Number of `'R'` moves: 2 (the fourth and fifth characters).
   - Number of `'_'` moves: 2 (the second and sixth characters).
2. Compute the absolute difference between the number of `'L'` and `'R'` moves:
   - The difference is `abs(2 - 2)` which equals `0`. This shows that the net movement to the left or to the right, without considering the `'_'` moves, would leave us at the origin.
3. Add the number of `'_'` moves to this difference:
   - Since the difference was `0`, adding the 2 `'_'` moves would give us `0 + 2 = 2`.

Therefore, the furthest distance from the origin that can be achieved with the given `moves` string `"L_LRR_"` is `2`. We can interpret this as having the flexibility to move 2 additional units in either direction, on top of the net movement calculated between `'L'` and `'R'`, resulting in a maximum possible deviation of 2 units from the origin.

The Python function would execute as follows:

```python
solution = Solution()
print(solution.furthestDistanceFromOrigin("L_LRR_"))  # Output: 2
```

This example confirms that the given solution approach effectively calculates the furthest possible distance from the origin by counting and comparing the moves without needing to simulate each move step by step.

Solution Implementation

Python

```python
class Solution:
    def furthestDistanceFromOrigin(self, moves: str) -> int:
        # Count the number of 'L' moves
        left_moves = moves.count("L")
        # Count the number of 'R' moves
        right_moves = moves.count("R")
        # Count the number of '_' (stand still) moves
        stand_still_moves = moves.count("_")

        # Calculate the horizontal distance from the origin
        # by finding the absolute difference between left and right moves.
        horizontal_distance = abs(left_moves - right_moves)

        # The furthest distance from the origin is the sum of the absolute
        # horizontal distance and the number of stand still moves.
        # Stand still moves do not change the horizontal or vertical position,
        # so they are added directly to the furthest distance.
        furthest_distance = horizontal_distance + stand_still_moves

        return furthest_distance

# Example usage:
# solution = Solution()
# moves = "L_R__R_L"
# print(solution.furthestDistanceFromOrigin(moves))  # Output will be 3
```

Java

```java
class Solution {

    /**
     * Calculates the furthest distance from the origin based on a set of moves.
     * Assumes moves only in the left ('L'), right ('R'), and up ('_') directions.
     * Left and right movements will cancel each other out, while up moves will always
     * increase the distance from the origin.
     *
     * @param moves String representing a sequence of moves.
     * @return The furthest distance from the origin following the moves.
     */
    public int furthestDistanceFromOrigin(String moves) {
        // Calculate the net horizontal distance by finding the difference
        // between left (L) and right (R) moves.
        int horizontalDistance = Math.abs(countOccurrences(moves, 'L') - countOccurrences(moves, 'R'));

        // Add the total vertical ('_') moves to the net horizontal distance
        // to find the furthest distance from the origin.
        return horizontalDistance + countOccurrences(moves, '_');
    }

    /**
     * Counts the occurrences of a specified character in a given string.
     *
     * @param str The string to search.
     * @param ch The character to count.
     * @return The number of times the character appears in the string.
     */
    private int countOccurrences(String str, char ch) {
        // Initialize count to 0.
        int count = 0;

        // Iterate over each character in the string.
        for (int i = 0; i < str.length(); i++) {
            // If the current character matches the target character, increment the count.
            if (str.charAt(i) == ch) {
                count++;
            }
        }

        // Return the total count of the target character in the string.
        return count;
    }
}
```

C++

```cpp
#include <algorithm> // Include algorithm for std::count
#include <string>    // Include string for std::string
#include <cmath>     // Include cmath for std::abs

class Solution {
public:
    // Calculates the furthest distance from the origin, given a string of moves.
    int furthestDistanceFromOrigin(std::string moves) {
        // Helper lambda to count occurrences of a character in the moves string.
        auto countCharacter = [&](char character) {
            return std::count(moves.begin(), moves.end(), character);
        };

        // The net horizontal displacement is the difference between the counts
        // of 'L' and 'R' moves since 'L' moves left and 'R' moves right.
        // The '_' represents a pause or stay at the current position, so it adds
        // no displacement but still counts towards the furthest distance reached.
        // The std::abs function is used to ensure we get a non-negative distance.
        return std::abs(countCharacter('L') - countCharacter('R')) + countCharacter('_');
    }
};
```

TypeScript

```typescript
function furthestDistanceFromOrigin(moves: string): number {
    // Counts the occurrences of a character within the 'moves' string.
    const countOccurrences = (character: string): number => {
        return moves.split('').filter(x => x === character).length;
    };

    // Calculate the horizontal distance by finding the
    // absolute difference between 'L' and 'R' occurrences.
    const horizontalDistance: number = Math.abs(
        countOccurrences('L') - countOccurrences('R')
    );

    // Vertical movement is not accounted for in the calculation,
    // so an underscore represents an unknown move that doesn't
    // change the horizontal distance.

    const unknownMoves: number = countOccurrences('_'); // Counts the '_' occurrences.

    // Return the furthest horizontal distance from the origin,
    // considering the unknown moves as they could all be in one direction.
    return horizontalDistance + unknownMoves;
}
```

```python
class Solution:
    def furthestDistanceFromOrigin(self, moves: str) -> int:
        # Count the number of 'L' moves
        left_moves = moves.count("L")
        # Count the number of 'R' moves
        right_moves = moves.count("R")
        # Count the number of '_' (stand still) moves
        stand_still_moves = moves.count("_")

        # Calculate the horizontal distance from the origin
        # by finding the absolute difference between left and right moves.
        horizontal_distance = abs(left_moves - right_moves)

        # The furthest distance from the origin is the sum of the absolute
        # horizontal distance and the number of stand still moves.
        # Stand still moves do not change the horizontal or vertical position,
        # so they are added directly to the furthest distance.
        furthest_distance = horizontal_distance + stand_still_moves

        return furthest_distance

# Example usage:
# solution = Solution()
# moves = "L_R__R_L"
# print(solution.furthestDistanceFromOrigin(moves))  # Output will be 3
```

Time and Space Complexity

The time complexity of the code is `O(n)` where `n` is the length of the input string `moves`. This is because the `count` method on a string traverses the entire string once for each count operation to compute the number of occurrences of the given character. There are 3 count operations, each of which is `O(n)`, but, since they are not nested, the overall time complexity remains `O(n)`.

The space complexity of the code is `O(1)`. The space used does not depend on the size of the input string because the number of variables used (`moves`) is constant regardless of the length of the string. The count operations do not use any additional space that scales with the input size, as they simply return an integer value.