2211. Count Collisions on a Road

Medium Graph String

Problem Description

In this problem, we are given a number `n` representing the number of cars on an infinitely long road. The cars are indexed from `0` to `n - 1`. Each car is at a distinct position and moves in either left, right, or stays stationary as indicated by the characters 'L', 'R', and 'S' in the given string `directions`.

Our task is to calculate the total number of collisions on this road given the rule that:

- When two cars moving in opposite directions collide, the collision count goes up by `2`.
- When a moving car hits a stationary car, the collision count increases by `1`.

A collision causes the involved cars to stop moving and stay at their collision point. Cars continue to move in their initial direction or stay still unless they collide.

We are asked to return the total number of collisions that will occur according to these rules.

Intuition

To solve this problem, the `rstrip('R')` and `lstrip('L')` methods come in handy because cars moving outwards towards the right or left indefinitely without facing each other will never collide with other cars. By stripping 'L' characters from the beginning and 'R' characters from the end of the string, we disregard those cases where no collisions will occur ever. What we are left with is the central part of the string where all potential collisions may happen.

Here's the intuition behind the solution:

- Cars that move out of the scene immediately (those moving to the right at the end and to the left at the start) won't be involved in any collisions.
- For the remaining cars (that might collide), each car will be involved in at least one collision except for those marked with 'S' which represent staying stationary. If a car is staying stationary, it will cause at most one collision (when hit by a moving car).

The provided solution counts the length of the stripped directions string (after removing the non-colliding 'L's at the start and 'R's at the end) to get the total number of cars that will be involved in collisions. From this, subtracting the count of 'S' characters gives us the total number of cars actually moving and therefore the total number of collisions since moving cars either hit another car or get hit.

Therefore, the total collision count is the number of actionable cars (moving cars) in the middle segment of the road since stationary cars will only add to the count when being hit, which the moving cars will guarantee. This solution ensures an efficient way to calculate the total number of collisions without having to simulate each car's movement or potential collisions explicitly.

Solution Approach

The solution to this problem is quite straightforward and does not require complex data structures or algorithms. The Python code provided leverages the language's built-in string manipulation methods to efficiently compute the result.

1. **String Trimming**: Using the `lstrip('L')` and `rstrip('R')` methods, we trim the 'L' characters from the starting of the `directions` string and 'R' characters from the ending. This represents removing the non-interacting cars that are moving indefinitely to the left from the start or to the right at the end without any potential for collision.
2. **Counting Moving Cars**: Once we have the trimmed string `d`, which now contains only cars that will definitely be involved in collisions or will stay stationary, we need to find out the total number of moving cars. This is because each moving car will eventually collide with another car or a stationary object, resulting in a collision.
3. **Calculating Collisions**: The collision count can be calculated by taking the length of the trimmed string `len(d)` (which represents the number of cars that are either moving or staying) and subtracting the number of 'S' characters `d.count('S')` from it. The reason for this subtraction is that 'S' represents stationary cars, which do not actively cause collisions but rather are the targets of collisions by moving cars. Each 'S' reduces the collision count by 1 because a stationary car paired with a moving car results in only one collision, not two as with two moving cars.

The final line `return len(d) - d.count('S')` gives the total number of collisions as required.

In summary, the solution approach is to exclude cars that will not participate in any collisions and then to calculate the number of collisions based on the reduced set of cars that have the potential to collide. This solution is efficient as it avoids unnecessary iteration and complex logic, instead relying on simple string operations to achieve the desired result.

Example Walkthrough

Let's consider a small example with `n = 7` cars and the `directions` string as `"LRSRLRR"`. We'll follow the steps outlined in the solution approach to calculate the total number of collisions.

1. String Trimming

By trimming the string, we remove any 'L' from the start and any 'R' from the end that won't be involved in collisions. Trimming `'LRSRLRR'` would result in `'RSRL'`.

Original string: `LRSRLRR` After trimming: `RSRL`

2. Counting Moving Cars

Now, we need to count how many moving cars there are in the trimmed string. We look at the string after trimming and see two 'R's and one 'L', making a total of three moving cars. The single 'S' represents a stationary car that will not initiate a collision.

Trimmed string: `RSRL` Moving cars: `R`, `R`, `L` (3 in total) Stationary cars: `S` (1 in total)

3. Calculating Collisions

To calculate the number of collisions, we take the length of the trimmed string, which is `4`, and subtract the number of 'S' characters, which is `1`. This leaves us with `4 - 1 = 3` collisions.

Length of trimmed string: `4` (`RSRL`) Count of 'S': `1` Total collisions: `4 - 1 = 3`

By following the above steps, we determine that there will be a total of `3` collisions according to the rules given in the problem description for our example string `"LRSRLRR"`.

Solution Implementation

Python

```python
class Solution:
    def countCollisions(self, directions: str) -> int:
        # Strip 'L' from the left end and 'R' from the right end of the directions string
        # because 'L' at the start or 'R' at the end won't cause any collisions.
        sanitized_directions = directions.lstrip('L').rstrip('R')

        # Count the number of collisions:
        # Total number of cars that can collide is the length of the sanitized directions
        # subtracted by the number of 'S' (stationary) cars since 'S' cars do not move.
        collisions = len(sanitized_directions) - sanitized_directions.count('S')

        return collisions
```

Java

```java
class Solution {

    public int countCollisions(String directions) {
        // Convert the input string to a character array for easier processing.
        char[] directionChars = directions.toCharArray();

        // Get the length of the directionChars array.
        int length = directionChars.length;

        // Initialize pointers for left and right.
        int leftPointer = 0;
        int rightPointer = length - 1;

        // Skip all the 'L' cars from the start as they do not contribute to collisions.
        while (leftPointer < length && directionChars[leftPointer] == 'L') {
            leftPointer++;
        }

        // Skip all the 'R' cars from the end as they do not contribute to collisions.
        while (rightPointer >= 0 && directionChars[rightPointer] == 'R') {
            rightPointer--;
        }

        // Initialize a counter for collisions to zero.
        int collisionsCount = 0;

        // Iterate over the remaining cars between leftPointer and rightPointer.
        for (int i = leftPointer; i <= rightPointer; ++i) {
            // Count only the cars that are not 'S' (since 'S' means stopped and will not collide).
            if (directionChars[i] != 'S') {
                collisionsCount++;
            }
        }

        // Return the total count of collisions.
        return collisionsCount;
    }
}
```

C++

```cpp
class Solution {
public:
    int countCollisions(string directions) {
        // Variables to keep track of the left and right pointers
        int leftIndex = 0, rightIndex = directions.size() - 1;

        // Variable to count the number of collisions
        int collisionCount = 0;

        // Skip all cars moving out of bounds to the left at the beginning
        while (leftIndex <= rightIndex && directions[leftIndex] == 'L') {
            leftIndex++;
        }

        // Skip all cars moving out of bounds to the right at the end
        while (leftIndex <= rightIndex && directions[rightIndex] == 'R') {
            rightIndex--;
        }

        // Count collisions - all cars in the middle will collide, except those stationary ('S')
        for (int i = leftIndex; i <= rightIndex; i++) {
            if (directions[i] != 'S') {
                collisionCount++;
            }
        }

        // Return the total collision count
        return collisionCount;
    }
};
```

TypeScript

```typescript
function countCollisions(directions: string): number {
  // Determine the length of the directions string.
  const directionsLength: number = directions.length;

  // Initialize pointers to the start and end of the string.
  let leftPointer: number = 0,
      rightPointer: number = directionsLength - 1;

  // Skip all 'L' from the start as they do not contribute to collisions.
  while (leftPointer < directionsLength && directions[leftPointer] === 'L') {
    leftPointer++;
  }

  // Skip all 'R' from the end as they do not contribute to collisions.
  while (rightPointer >= 0 && directions[rightPointer] === 'R') {
    rightPointer--;
  }

  // Initialize the collision count.
  let collisionCount: number = 0;

  // Check the remaining part of the string for 'L' or 'R' which will collide.
  for (let index = leftPointer; index <= rightPointer; index++) {
    if (directions[index] !== 'S') {
      // Increment collision count for each 'L' or 'R' as they result in a collision.
      collisionCount++;
    }
  }

  // Return the total number of collisions.
  return collisionCount;
}
```

```python
class Solution:
    def countCollisions(self, directions: str) -> int:
        # Strip 'L' from the left end and 'R' from the right end of the directions string
        # because 'L' at the start or 'R' at the end won't cause any collisions.
        sanitized_directions = directions.lstrip('L').rstrip('R')

        # Count the number of collisions:
        # Total number of cars that can collide is the length of the sanitized directions
        # subtracted by the number of 'S' (stationary) cars since 'S' cars do not move.
        collisions = len(sanitized_directions) - sanitized_directions.count('S')

        return collisions
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily determined by three operations:

1. `lstrip('L')`: This must check each character from the left until a non-L character is found. In the worst case, all characters are 'L', having a complexity of $O(n)$ where n is the total number of characters in the string.
2. `rstrip('R')`: Similarly, this function must check each character from the right until a non-R character is encountered. This also has a worst-case complexity of $O(n)$ when all characters are 'R'.
3. `count('S')`: This operation counts the number of 'S' characters in the modified string. This takes $O(m)$ time where m is the length of the modified string. However, since $m <= n$, we also consider it $O(n)$ for the worst case.

When these operations are added together, despite being sequential and not nested, the complexity is still governed by the longest operation which is $O(n)$.

So, the overall time complexity of the code is $O(n)$.

Space Complexity

The space complexity of the code is determined by the storage required for the modified string `d`.

- `d` is a substring of the original input `directions`. However, it does not require additional space proportional to the input size; it uses the slices (which are views in Python) to reference parts of the original string without creating a new copy.
- Thus, the extra space used is for a fixed number of variables which do not grow with the size of the input.

Hence, the space complexity is $O(1)$.