<https://leetcode.com/problems/count-collisions-on-a-road/description/>

There are n cars on an infinitely long road. The cars are numbered from 0 to n - 1 from left to right and each car is present at a unique point.

You are given a 0-indexed string directions of length n. directions[i] can be either 'L', 'R', or 'S' denoting whether the ith car is moving towards the left, towards the right, or staying at its current point respectively. Each moving car has the same speed.

The number of collisions can be calculated as follows:

When two cars moving in opposite directions collide with each other, the number of collisions increases by 2.

When a moving car collides with a stationary car, the number of collisions increases by 1.

After a collision, the cars involved can no longer move and will stay at the point where they collided. Other than that, cars cannot change their state or direction of motion.

Return the total number of collisions that will happen on the road.

**Example 1:**

**Input:** directions = "RLRSLL"

**Output:** 5

**Explanation:**

The collisions that will happen on the road are:

- Cars 0 and 1 will collide with each other. Since they are moving in opposite directions, the number of collisions becomes 0 + 2 = 2.

- Cars 2 and 3 will collide with each other. Since car 3 is stationary, the number of collisions becomes 2 + 1 = 3.

- Cars 3 and 4 will collide with each other. Since car 3 is stationary, the number of collisions becomes 3 + 1 = 4.

- Cars 4 and 5 will collide with each other. After car 4 collides with car 3, it will stay at the point of collision and get hit by car 5. The number of collisions becomes 4 + 1 = 5.

Thus, the total number of collisions that will happen on the road is 5.

**Example 2:**

**Input:** directions = "LLRR"

**Output:** 0

**Explanation:**

No cars will collide with each other. Thus, the total number of collisions that will happen on the road is 0.

**Constraints:**

1 <= directions.length <= 10^5

directions[i] is either 'L', 'R', or 'S'.

**Attempt 1: 2024-01-17**

**Solution 1: Brute Force (30 min)**

**Refer to**

<https://algo.monster/liteproblems/2211>

**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.

**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.

**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.

**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.

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

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)

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**

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;

}

}

**Time and Space Complexity**

**Time Complexity**

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

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.

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'.

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).

**More explain from chatGPT on below section:**

**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.

To prove the methodology, particularly around the logic for calculating collisions, we can break down the situation into manageable steps. Let’s review the key assumptions and logic behind your solution:

**Key Observations:**

Trimming: The cars at the far left ('L') and far right ('R') of the string that don't meet any other cars should not be involved in collisions, which is why trimming is valid.

All left-moving cars at the very start and right-moving cars at the very end of the string will not interact with any other cars. Hence, trimming them ensures we're left with only cars that will eventually collide.

Counting moving cars:

After trimming, the cars left in the string are the ones that may collide.

**Cars represented by 'S' are stationary and don’t move, meaning they can't cause collisions on their own. However, they can be hit by moving cars.**

**Collision Counting:**

Let’s break this down further with respect to each character:

'R' (right-moving): Can collide with stationary cars 'S' or left-moving cars 'L'.

'L' (left-moving): Can collide with stationary cars 'S' or right-moving cars 'R'.

'S' (stationary): Can only stop cars moving into them.

Thus, the essential logic for counting collisions is:

**Moving cars cause collisions: All cars that are either 'R' or 'L' (i.e., moving) will result in a collision.**

**'S' represents the stationary cars that only act as targets for collisions but don't "add" extra collisions by themselves.**

Now, proving the formula:

The length of the trimmed string **len(d)** represents the number of cars that are in potential conflict (cars that might collide or are already stationary due to previous collisions).

d.count('S') counts the number of stationary cars in the middle of the trimmed string. Each 'S' doesn't add to the collisions; it merely receives moving cars. Every moving car ('R' or 'L') interacting with an 'S' will result in one collision.

**Why Subtract d.count('S')?**

For every 'S' encountered, it **reduces the number of collisions by 1**, **because only one moving car can collide with an 'S' at a time. Thus, each 'S' effectively replaces one of the moving cars in a potential two-car collision. Therefore, by subtracting the number of 'S' characters from the total number of cars in the middle segment (len(d)), you count the actual number of collisions.**

**Proof by Cases:**

Let’s test this logic with some cases:

No **'S' (all moving cars)**:

Example: RRL → After trimming (no trimming here), all cars will collide. Total collisions: 2.Formula: len("RRL") - 0 = 3 - 0 = 3

With stationary cars **'S'**:

Example: RSSL → Trimmed to RSSL (no outer left 'L' or right 'R' to trim). The two 'S' characters receive one collision each from the 'R' cars and 'L' cars moving into them.

Collisions: The two moving cars ('R' and 'L') collide into the 'S'. Total collisions = 2.Formula: len("RSSL") - count('S') = 4 - 2 = 2.

**Conclusion:**

Each 'S' reduces the number of moving cars involved in collisions by 1. Therefore, the logic of subtracting d.count('S') from the total length len(d) holds valid. The result accurately reflects the total number of collisions by considering that stationary cars don’t contribute extra collisions.