2731. Movement of Robots

positions based on their intended direction.

problem solvable within the given constraints.

Brainteaser Array Prefix Sum Sorting

# **Problem Description**

Medium

direction each robot will head towards once they're commanded to move. The directions 'L' and 'R' stand for left (toward negative infinity) and right (toward positive infinity), respectively. Robots will move at a rate of one unit per second. A key point in the problem is that if two robots meet at the same point at the same time, they instantaneously switch directions but continue moving without any delay. This collision means that we could consider the robots as if they pass through each other and continue in their initial directions for the purpose of calculation.

In this problem, we are provided with an array nums which represents the initial positions of a group of robots on an infinite

number line, where each element is a unique robot's starting position. Along with this, we have a string s that contains the

The task is to calculate the sum of all distances between pairs of robots after a given number of seconds d and to return this sum modulo 10^9 + 7 due to the potential for large numbers.

At first glance, calculating the distances between robots after an arbitrary number of seconds seems complicated, especially

## when considering potential collisions. However, the key realization that simplifies this problem is to understand that we don't actually have to simulate the movement of robots and handle their collisions.

Because robots change direction instantaneously upon collision and there is no delay, we can imagine that the robots simply pass through each other and continue in their original direction. This means that after d seconds, regardless of any collisions that might have happened, we can calculate the new position of each robot by simply adding or subtracting d from their initial

After determining the theoretical new positions, we can sort these positions in ascending order. We then iterate through this sorted list to calculate the running sum of distances between each robot and all that come before it. This is effectively the cumulative distance from each of the robots to the start. Summing up the distances while iterating through the sorted list provides the total sum of distances between all pairs of robots after d seconds. We take care to apply the modulo operation as required to handle the potential for very large numbers.

This approach negates the need to handle the complexity of collisions during the movement phase, which is what makes the

Solution Approach

The implementation of this problem's solution uses simple yet elegant logic and a few well-established Python techniques. Firstly, we manipulate the initial positions of the robots according to the directions specified by the s string. This is carried out by

a combination of enumeration and conditional addition or subtraction. Depending on whether the direction at index i in s is 'R'

collisions further.

statement.

**Example Walkthrough** 

The updated positions array structure is then sorted. Sorting here uses the default O(n log n) sorting algorithm, which is common practice in Python. Sorting is crucial as it prepares us to compute distances without having to track individual movements or

(right) or 'L' (left), we add or subtract the duration d from the corresponding starting position in nums.

were i units apart, while subtracting s corrects for their actual positions.

Next, an iterative approach is used. The loop calculates the running sum of distances by traversing through the sorted array of new positions. To achieve this, two accumulator variables are used: ans, which accumulates the sum of distances, and s, which serves as the running sum of robot positions. The distance from a given robot to all robots before it in the sorted list is i \* x - sfor each robot x at index i. The product i \* x gives the sum of distances from all previous robots to the current robot if they

Finally, the solution modulo 10^9 + 7 is applied to the cumulative distance sum to obtain the answer required by the problem

The Python code implementation follows a clear sequence and is succinct, relying on a simple enumeration pattern and basic

arithmetic to calculate the cumulative distances in the sorted array of robot positions. It does not employ complex data structures or algorithms but uses familiar constructs such as loops and sort functions effectively to derive the solution.

Suppose we have the array nums = [4, 1, 5] representing the starting positions of robots and a string s = "RLR" that indicates

We process each position according to the direction provided. After d seconds, the new positions will be calculated as

 For the first robot at nums[0] = 4 with direction s[0] = 'R', the new position will be 4 + 2 = 6.  $\circ$  The second robot starts at nums[1] = 1 with direction s[1] = 'L', giving us a new position of 1 - 2 = -1. ○ The third robot starts at nums [2] = 5 with direction s[2] = 'R', making the new position 5 + 2 = 7.

## The updated array of positions then becomes [6, -1, 7].

problem.

Solution Implementation

follows:

To find the sum of all distances between pairs of robots after moving, we iterate through the sorted list and calculate the

 $\circ$  The distance from 6 to 7 is 7 - 6 = 1.

We sort this array to get the order after d seconds: [-1, 6, 7].

• For the second robot (6), we have (1 \* 6) - (-1) = 6 + 1 = 7.

Let's illustrate the solution approach with a small example:

cumulative distance:  $\circ$  The distance from -1 to 6 is 6 - (-1) = 7.

- Now we calculate the sum of all distances: For the first robot (-1), the cumulative sum is ∅.
- $\circ$  For the final robot (7), we have (2 \* 7) (6 1) = 14 5 = 9. The total sum of distances is therefore 0 + 7 + 9 = 16.

# Update each number in the list according to the corresponding direction character

# If character is not "R", subtract the distance value (implying "L" is encountered)

the direction each robot will move. We are also given d = 2 seconds to simulate the movement.

Applying the solution modulo  $10^9 + 7$ , the final answer remains 16, since 16 is much smaller than  $10^9 + 7$ . Using this walkthrough, we gain a clear understanding of how to approach and solve the problem as outlined in the solution

approach without directly considering robot collisions. This simplifies the calculations and allows for an efficient solution to the

**Python** class Solution: def sumDistance(self, numbers: List[int], directions: str, distance: int) -> int: # Define the modulo value for large numbers to prevent overflow  $modulo_value = 10**9 + 7$ 

for index, direction char in enumerate(directions):

# Sort the numbers to calculate total sum distance

# If character is "R", add the distance value

# Initialize variables for the answer and the cumulative sum

# minus the cumulative sum of all previous elements

# Update the cumulative sum with the current number

total\_sum\_distance += index \* number - cumulative\_sum

# Return the total sum distance modulo the defined modulo\_value

if direction char == "R":

cumulative\_sum += number

long result = 0, cumulativeSum = 0;

final int modulo = (int) 1e9 + 7;

for (int i = 0; i < n; ++i) {

#include <algorithm> // Required for std::sort

for (int i = 0; i < n; ++i) {

#include <vector> // Required for std::vector

return (int) result;

// Define modulo constant for large number handling

cumulativeSum += adjustedDistances[i];

// Required for std::string

int sumDistance(vector<int>& nums, string s, int d) {

// Function to calculate the sum of distances based on conditions.

sort(adjustedPositions.begin(), adjustedPositions.end());

long long answer = 0; // Initialize the sum of distances as 0.

int n = nums.size(); // Get the size of the input vector 'nums'.

// Calculate adjusted positions based on 'L' or 'R' in string 's'.

adjustedPositions[i] = 1LL \* nums[i] + (s[i] == 'L' ? -d : d);

// Sort the adjusted positions to access them in non-decreasing order.

def sumDistance(self, numbers: List[int], directions: str, distance: int) -> int:

# The current element's contribution is its value times its index

# Update each number in the list according to the corresponding direction character

# If character is not "R", subtract the distance value (implying "L" is encountered)

# Define the modulo value for large numbers to prevent overflow

# If character is "R", add the distance value

# Initialize variables for the answer and the cumulative sum

# minus the cumulative sum of all previous elements

# Update the cumulative sum with the current number

total\_sum\_distance += index \* number - cumulative\_sum

for index, direction char in enumerate(directions):

# Sort the numbers to calculate total sum distance

# Calculate the sum-distance (sum of all |Pj - Pi|)

vector<long long> adjustedPositions(n); // Create a vector to store adjusted positions.

long long prefixSum = 0; // Keep track of the sum of previous adjusted positions.

const int MOD = 1e9 + 7; // Define the modulus value for avoiding integer overflow.

// Return the result cast back to integer

// Calculate the weighted sum of distances and update result

// Update cumulative sum with the current element's value

result = (result + i \* adjustedDistances[i] - cumulativeSum) % modulo;

numbers[index] += distance

numbers[index] -= distance

### cumulative\_sum = 0 # Calculate the sum-distance (sum of all |Pj - Pi|) for index, number in enumerate(numbers): # The current element's contribution is its value times its index

else:

numbers.sort()

total sum distance = 0

```
return total_sum_distance % modulo_value
Java
class Solution {
    public int sumDistance(int[] numbers, String direction, int distance) {
        // Get the length of the input array
        int n = numbers.length;
        // Create an array to store adjusted distances
        long[] adjustedDistances = new long[n];
        // Calculate adjusted distances based on direction and store them
        for (int i = 0; i < n; ++i) {
            // Subtract or add the distance based on if the direction is 'L' or 'R'
            adjustedDistances[i] = (long) numbers[i] + (direction.charAt(i) == 'L' ? -distance : distance);
        // Sort the adjusted distances
        Arrays.sort(adjustedDistances);
        // Initialize variables for storing the result and the cumulative sum
```

// Update the result with the current index times the element minus the cumulative sum so far

C++

public:

#include <string>

class Solution {

```
// Calculate the sum of distances using prefix sums.
        for (int i = 0: i < n: ++i) {
            answer = (answer + i * adjustedPositions[i] - prefixSum) % MOD;
            prefixSum = (prefixSum + adjustedPositions[i]) % MOD; // Update prefix sum.
        return answer; // Return final answer.
};
TypeScript
function sumDistance(nums: number[], directions: string, distance: number): number {
    // Define the length of the nums array
    const length = nums.length;
    // Update the nums array by adding or subtracting the distance based on the direction
    for (let i = 0; i < length; ++i) {
        nums[i] += directions[i] === 'L' ? -distance : distance;
    // Sort the nums array in ascending order
    nums.sort((a, b) => a - b);
    // Initialize the answer and a variable to keep track of the cumulative sum
    let answer = 0;
    let cumulativeSum = 0;
    // Define the modulus value to handle large numbers
    const modulus = 1e9 + 7;
    // Iterate over nums to calculate the final answer
    for (let i = 0; i < length; ++i) {
        // Update the answer according to the formula
        answer = (answer + i * nums[i] - cumulativeSum) % modulus;
        // Update the cumulative sum with the current number
        cumulativeSum += nums[i];
```

## cumulative\_sum += number # Return the total sum distance modulo the defined modulo\_value return total\_sum\_distance % modulo\_value

Time and Space Complexity

// Return the answer

else:

numbers.sort()

total sum distance = 0

cumulative sum = 0

 $modulo_value = 10**9 + 7$ 

if direction char == "R":

numbers[index] += distance

numbers[index] -= distance

for index, number in enumerate(numbers):

n), the overall time complexity remains  $O(n \log n)$ .

return answer;

class Solution:

The given code's time complexity primarily comes from the sorting operation. • Sorting a list of n elements typically takes 0(n log n) time. This is the dominating factor in the time complexity of this function.

- For space complexity:
- The given code modifies the input list nums in-place and uses a fixed number of integer variables (mod, ans, s). Thus, apart from the input list, only constant extra space is used. • However, since the input list nums itself takes 0(n) space, and we consider the space taken by inputs for space complexity analysis, the overall

• The remaining part of the code iterates over the list once, which is an O(n) operation. However, since O(n log n) + O(n) simplifies to O(n log

- space complexity is O(n). Therefore, we can conclude that:
- The time complexity of the code is O(n log n). The space complexity of the code is O(n).