# Problem 2731: Movement of Robots

## Problem Information

**Difficulty:** Medium
**Acceptance Rate:** 0.00%
**Paid Only:** No

## Problem Description

Some robots are standing on an infinite number line with their initial coordinates given by a

0-indexed

integer array

nums

and will start moving once given the command to move. The robots will move a unit distance each second.

You are given a string

s

denoting the direction in which robots will move on command.

'L'

means the robot will move towards the left side or negative side of the number line, whereas

'R'

means the robot will move towards the right side or positive side of the number line.

If two robots collide, they will start moving in opposite directions.

Return

the sum of distances between all the pairs of robots

d

seconds after the command.

Since the sum can be very large, return it modulo

10

9

+ 7

.

Note:

For two robots at the index

i

and

j

, pair

(i,j)

and pair

(j,i)

are considered the same pair.

When robots collide, they

instantly change

their directions without wasting any time.

Collision happens when two robots share the same place in a moment.

For example, if a robot is positioned in 0 going to the right and another is positioned in 2 going to the left, the next second they'll be both in 1 and they will change direction and the next second the first one will be in 0, heading left, and another will be in 2, heading right.

For example, if a robot is positioned in 0 going to the right and another is positioned in 1 going to the left, the next second the first one will be in 0, heading left, and another will be in 1, heading right.

Example 1:

Input:

nums = [-2,0,2], s = "RLL", d = 3

Output:

8

Explanation:

After 1 second, the positions are [-1,-1,1]. Now, the robot at index 0 will move left, and the robot at index 1 will move right. After 2 seconds, the positions are [-2,0,0]. Now, the robot at index 1 will move left, and the robot at index 2 will move right. After 3 seconds, the positions are [-3,-1,1]. The distance between the robot at index 0 and 1 is abs(-3 - (-1)) = 2. The distance between the robot at index 0 and 2 is abs(-3 - 1) = 4. The distance between the robot at index 1 and 2 is abs(-1 - 1) = 2. The sum of the pairs of all distances = 2 + 4 + 2 = 8.

Example 2:

Input:

nums = [1,0], s = "RL", d = 2

Output:

5

Explanation:

After 1 second, the positions are [2,-1]. After 2 seconds, the positions are [3,-2]. The distance between the two robots is abs(-2 - 3) = 5.

Constraints:

2 <= nums.length <= 10

5

-2 * 10

9

<= nums[i] <= 2 * 10

9

0 <= d <= 10

9

nums.length == s.length

s

consists of 'L' and 'R' only

nums[i]

will be unique.

## Code Snippets

### C++:

```
class Solution {
public:
int sumDistance(vector<int>& nums, string s, int d) {

}
};
```

### Java:

```
class Solution {
public int sumDistance(int[] nums, String s, int d) {

}
}
```

### Python3:

```
class Solution:
def sumDistance(self, nums: List[int], s: str, d: int) -> int:
```

### Python:

```
class Solution(object):
def sumDistance(self, nums, s, d):
"""
:type nums: List[int]
:type s: str
:type d: int
:rtype: int
"""
```

### JavaScript:

```
/**
* @param {number[]} nums
* @param {string} s
* @param {number} d
* @return {number}
*/
```

```
var sumDistance = function(nums, s, d) {

};
```

**TypeScript:**

```
function sumDistance(nums: number[], s: string, d: number): number {

};
```

**C#:**

```
public class Solution {
public int SumDistance(int[] nums, string s, int d) {

}
}
```

**C:**

```
int sumDistance(int* nums, int numsSize, char* s, int d) {

}
```

**Go:**

```
func sumDistance(nums []int, s string, d int) int {

}
```

**Kotlin:**

```
class Solution {
fun sumDistance(nums: IntArray, s: String, d: Int): Int {

}
}
```

**Swift:**

```
class Solution {
func sumDistance(_ nums: [Int], _ s: String, _ d: Int) -> Int {
```

```
}
}
```

**Rust:**

```
impl Solution {
pub fn sum_distance(nums: Vec<i32>, s: String, d: i32) -> i32 {

}
}
```

**Ruby:**

```
# @param {Integer[]} nums
# @param {String} s
# @param {Integer} d
# @return {Integer}
def sum_distance(nums, s, d)

end
```

**PHP:**

```
class Solution {

/**
* @param Integer[] $nums
* @param String $s
* @param Integer $d
* @return Integer
*/
function sumDistance($nums, $s, $d) {

}
}
```

**Dart:**

```
class Solution {
int sumDistance(List<int> nums, String s, int d) {
```

```
}
}
```

**Scala:**

```
object Solution {
def sumDistance(nums: Array[Int], s: String, d: Int): Int = {

}
}
```

**Elixir:**

```
defmodule Solution do
@spec sum_distance(nums :: [integer], s :: String.t, d :: integer) :: integer
def sum_distance(nums, s, d) do

end
end
```

**Erlang:**

```
-spec sum_distance(Nums :: [integer()], S :: unicode:unicode_binary(), D ::
integer()) -> integer().
sum_distance(Nums, S, D) ->
.
```

**Racket:**

```
(define/contract (sum-distance nums s d)
(-> (listof exact-integer?) string? exact-integer? exact-integer?)
)
```

## Solutions

### C++ Solution:

```
/*
* Problem: Movement of Robots
* Difficulty: Medium
```

```
* Tags: array, string, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
int sumDistance(vector<int>& nums, string s, int d) {

}
};
```

### Java Solution:

```
/**
* Problem: Movement of Robots
* Difficulty: Medium
* Tags: array, string, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int sumDistance(int[] nums, String s, int d) {

}
}
```

### Python3 Solution:

```
"""
Problem: Movement of Robots
Difficulty: Medium
Tags: array, string, sort

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
```

```
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def sumDistance(self, nums: List[int], s: str, d: int) -> int:
# TODO: Implement optimized solution
pass
```

### Python Solution:

```
class Solution(object):
def sumDistance(self, nums, s, d):
"""
:type nums: List[int]
:type s: str
:type d: int
:rtype: int
"""
```

### JavaScript Solution:

```
/**
* Problem: Movement of Robots
* Difficulty: Medium
* Tags: array, string, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[]} nums
* @param {string} s
* @param {number} d
* @return {number}
*/
var sumDistance = function(nums, s, d) {

};
```

### TypeScript Solution:

```
/**
* Problem: Movement of Robots
* Difficulty: Medium
* Tags: array, string, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function sumDistance(nums: number[], s: string, d: number): number {

};
```

### C# Solution:

```
/*
* Problem: Movement of Robots
* Difficulty: Medium
* Tags: array, string, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int SumDistance(int[] nums, string s, int d) {

}
}
```

### C Solution:

```
/*
* Problem: Movement of Robots
* Difficulty: Medium
* Tags: array, string, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
```

```
* Space Complexity: O(1) to O(n) depending on approach
*/

int sumDistance(int* nums, int numsSize, char* s, int d) {

}
```

### Go Solution:

```
// Problem: Movement of Robots
// Difficulty: Medium
// Tags: array, string, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func sumDistance(nums []int, s string, d int) int {

}
```

### Kotlin Solution:

```
class Solution {
fun sumDistance(nums: IntArray, s: String, d: Int): Int {

}
}
```

### Swift Solution:

```
class Solution {
func sumDistance(_ nums: [Int], _ s: String, _ d: Int) -> Int {

}
}
```

### Rust Solution:

```
// Problem: Movement of Robots
// Difficulty: Medium
```

```
// Tags: array, string, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn sum_distance(nums: Vec<i32>, s: String, d: i32) -> i32 {

}
}
```

### Ruby Solution:

```
# @param {Integer[]} nums
# @param {String} s
# @param {Integer} d
# @return {Integer}
def sum_distance(nums, s, d)

end
```

### PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @param String $s
* @param Integer $d
* @return Integer
*/
function sumDistance($nums, $s, $d) {

}
}
```

### Dart Solution:

```
class Solution {
int sumDistance(List<int> nums, String s, int d) {
```

```
}
}
```

### Scala Solution:

```
object Solution {
def sumDistance(nums: Array[Int], s: String, d: Int): Int = {

}
}
```

### Elixir Solution:

```
defmodule Solution do
@spec sum_distance(nums :: [integer], s :: String.t, d :: integer) :: integer
def sum_distance(nums, s, d) do

end
end
```

### Erlang Solution:

```
-spec sum_distance(Nums :: [integer()], S :: unicode:unicode_binary(), D ::
integer()) -> integer().
sum_distance(Nums, S, D) ->
.
```

### Racket Solution:

```
(define/contract (sum-distance nums s d)
(-> (listof exact-integer?) string? exact-integer? exact-integer?)
)
```