1769. Minimum Number of Operations to Move All Balls to Each Box

String Medium <u>Array</u>

In this problem, we're working with a simulated array of n boxes, each represented by 0 if the box is empty, or by 1 if the box

Problem Description

box one by one. In each operation, we are allowed to move one ball to an adjacent box (to the left or right). The goal is to figure out for every box i the number of moves needed if all balls were to be moved to box i. The answer should be returned as an array where each index i contains the minimum operations for box i. The problem tests one's ability to efficiently compute cumulative operations while taking into account the both directions—left to right and right to left.

contains one ball. We are tasked with calculating the minimum number of operations required to move all balls into each individual

Intuition

To approach this problem, we need to come up with a way to count the operations for all positions without simulating each

individual move, as that would be too slow. We do so in two sweeps, first from left to right, and then from right to left.

In the first sweep, we start from the leftmost box and move right, counting how many balls we've seen so far and adding that

count to our operations because for each ball we pass, it would take one more operation to move it to the current box. We keep track of the total operations we'd need if we were moving all the balls to the right. In the second pass, we do the opposite. We start from the rightmost box and move left, also counting balls and accumulating the

operations needed to move all balls to the current position from the right. We then add this value to the corresponding position in our result from the first pass. The accumulation of these movements gives us the minimum operation count for each box because it folds in the moves needed from both sides.

Solution Approach The given solution takes a two-pass approach, which can be thought of as dynamic programming to some extent where we are

to store the operations required for each box. • Starting with the second element (since the first box would always require zero moves to get to itself), we iterate through the boxes string from

left to right.

Final Answer

First Pass: Left to Right

Second Pass: Right to Left

• The operations to move all balls encountered so far to the current box i is the sum of operations needed for the previous box ans [i-1] plus the number of balls encountered so far cnt. We set this value in the ans array at index i.

∘ If the previous box (at index i-1) contains a ball ('1'), we increment cnt as we have encountered another ball.

• We initialize a counter cnt to keep track of the number of balls seen so far and an array ans of zeros with the same length as the boxes string

 Now, we iterate through the string from right to left starting from the second-to-last box. • If the next box (at index i+1) contains a ball, we increment cnt by one because we've encountered another ball. • We then add cnt to s. The reasoning behind this is the same as the left to right pass but in reverse; s keeps a running total of the number

• We then initialize another counter s that will serve a similar purpose as cnt but for the right-to-left pass. We also reset cnt back to zero.

of operations needed to bring the balls from the right to the current box. Finally, we add s to the existing value in ans[i] to account for the operations needed from the right side.

• After completing both passes, the ans array now contains the sum of the operations needed from both sides (left and right) for each box and

This algorithm has a time complexity of O(n) because it passes through the boxes string only twice, irrespective of the number of

thus gives us the correct minimum number of operations for each box. The function returns the array ans as the final result.

building up the answer based on previous computations.

Let's consider a small example to illustrate the solution approach. Suppose we have an array of boxes: boxes = "001011". Let's walk through the algorithm.

We start with cnt = 0 and ans = [0, 0, 0, 0, 0] representing the initial minimum operations for each box.

At i = 1 (boxes[1] = "0"), the previous box is empty, so cnt remains 0 and ans[1] = ans[0] + cnt = 0.

Now, ans represents the operations to move the balls from left to right, and it looks like this: [0, 0, 1, 2, 4, 7].

At i = 3 (boxes[3] = "0"), we've seen one ball so far, so ans[3] = ans[2] + cnt = 1 + 1 = 2.

1, and add s to ans[4]: ans[4] = ans[4] + s = 4 + 1 = 5.

means if we want to move all balls to box 1, it will take us a total of 6 moves.

If the previous box contains a ball, increment the count

the number of operations carried over (cumulative)

operations[i] = operations[i - 1] + left_to_right_count

Traverse the boxes from right to left to add remaining operations

If the next box contains a ball, increment the count

Accumulate the sum of operations from right to left

The current box operation count is the previous box's count plus

Reset a variable to store the sum of operations when moving right to left

Add the number of operations from the right to the current answer

// Forward pass to calculate the operations required for each box from the left

// Accumulate the number of operations required for the current box

ballCount++; // Increment ball count if the next box has a ball.

return operations; // Return the final operations required for each box.

* Calculate the minimum number of operations to move all balls to each box.

operations[i] += operationSum;

function minOperations(boxes: string): number[] {

for (let i = 1; i < totalBoxes; i++) {</pre>

if (boxes[i - 1] === '1') {

if (boxes[i + 1] === '1') {

return operations;

from typing import List

class Solution:

const operations = new Array(totalBoxes).fill(0);

// Forward pass: Count operations from left to right

// Backward pass: Count operations from right to left

let totalOperations = 0; // Sum of operations needed

def minOperations(self, boxes: str) -> List[int]:

Calculate the length of the boxes string

for (let $i = totalBoxes - 2; i >= 0; i--) {$

let countLeft = 0: // Count of balls to left of current box

countLeft++; // Increment if a ball is found

let countRight = 0; // Count of balls to right of current box

countRight++; // Increment if a ball is found

totalOperations += countRight: // Accumulate total operations

const totalBoxes = boxes.length;

// Add the current right pass operations to the forward pass operations for each box.

* @param {string} boxes - A string where each character represents a box, '1' contains a ball, '0' does not.

* @returns {number[]} - An array containing the minimum number of operations needed for each box

operations[i] = operations[i - 1] + countLeft; // Add count to operations

operations[i] += totalOperations; // Add to previous count of operations

Initialize an array to hold the answer (number of operations for each box)

// If the previous box contains a ball, increment the count

Reset the counter for the number of operations when moving from right to left

ans[2] = ans[2] + s = 1 + 4 = 5.

def minOperations(self, boxes: str) -> List[int]:

Calculate the length of the boxes string

balls. It uses additional space for the array ans, giving it a space complexity of O(n) as well.

At i = 2 (boxes[2] = "1"), we find the first ball. We increment cnt to 1 since we've encountered a ball, and ans[2] = ans [1] + cnt = 0 + 1 = 1.

Example Walkthrough

First Pass: Left to Right

At i = 4 (boxes[4] = "1"), we encounter another ball, increment cnt to 2, and ans[4] = ans[3] + cnt = 2 + 2 = 4. At i = 5 (boxes[5] = "1"), we once again increment cnt to 3 as we've found another ball, and ans[5] = ans[4] + cnt = 4

We reset cnt to 0 and introduce s = 0. We start from the rightmost box and move leftward.

Python

class Solution:

from typing import List

num boxes = len(boxes)

if boxes[i - 1] == '1':

right to left count = 0

sum_operations = 0

left to right count += 1

for i in range(num boxes -2, -1, -1):

right to left count += 1

operations[i] += sum_operations

int[] operations = new int[length];

++count;

sum operations += right to left count

// Initialize the answer array with the same length

operations[i] = operations[i - 1] + count;

for (int i = 1, count = 0; i < length; ++i) {

if (boxes.charAt(i - 1) == '1') {

if boxes[i + 1] == '1':

Second Pass: Right to Left

+ 3 = 7.

2 = 4.

2. At i = 4 (boxes[4] = "1"), since the next box contains a ball, we increment cnt to 1. We add cnt to s, which makes s =

At i = 3 (boxes[3] = "0"), we have seen one ball on the right, so s becomes 1 + 1 = 2, and ans[3] = ans[3] + s = 2 +

4. At i = 2 (boxes[2] = "1"), we encounter another ball, and hence increment cnt to 2. Now, s = s + cnt = 2 + 2 = 4, and

At i = 0 (boxes [0] = "0"), we continue and s = s + cnt = 6 + 2 = 8, but since it's the leftmost box, we don't need to add

it to ans [0], which remains 0. The final ans array after accumulating operations from both sides is: [0, 6, 5, 4, 5, 7].

At i = 1 (boxes[1] = "0"), s = s + cnt = 4 + 2 = 6, and ans[1] = ans[1] + s = 0 + 6 = 6.

large strings. Solution Implementation

This array represents the minimum number of moves to get all balls in front of each respective box. For example, ans [1] = 6

Thus, our function would return [0, 6, 5, 4, 5, 7] as the final output for the input string "001011". This example demonstrates

the efficiency of the algorithm by avoiding the brute force approach of simulating each move, which would be impractical for

Initialize an array to hold the answer (number of operations for each box) operations = [0] * num boxes # Initialize a counter for the number of operations moving from left to right left_to_right_count = 0 # Traverse the boxes from left to right to calculate the number of operations needed for i in range(1, num boxes):

class Solution { public int[] minOperations(String boxes) { // Get the length of the input string int length = boxes.length();

Java

return operations

```
// Backward pass to calculate the operations required for each box from the right
        for (int i = length - 2, count = 0, sum = 0; i >= 0; --i) {
            // If the next box contains a ball, increment the count
            if (boxes.charAt(i + 1) == '1') {
                ++count;
            // Accumulate the number of operations required from the right side
            sum += count;
            // Add the right side operations to the current box's operations
            operations[i] += sum;
        // Return the final array with minimum operations required for each box
        return operations;
C++
#include <vector>
#include <string>
class Solution {
public:
    // Function to calculate the minimum number of operations required to move all balls to each box.
    std::vector<int> minOperations(std::string boxes) {
        int numberOfBoxes = boxes.size();
        std::vector<int> operations(numberOfBoxes); // This vector will store the result.
        // Forward pass to calculate the operations from the left.
        // Start from the second box (index 1), accumulate the count of balls (1's)
        // and the operations needed to bring them to each box.
        for (int i = 1, ballCount = 0; i < numberOfBoxes; ++i) {</pre>
            if (boxes[i - 1] == '1') {
                ballCount++; // Increment ball count if the previous box has a ball.
            // Accumulate the number of operations needed to bring all balls from left up to the current box.
            operations[i] = operations[i - 1] + ballCount;
       // Backward pass to calculate the operations from the right.
        // Start from the second-to-last box, accumulating the ball count going backward.
        // To operate both passes separately we use 'operationSum' to accumulate operations in this pass.
        for (int i = numberOfBoxes - 2, ballCount = 0, operationSum = 0; i >= 0; --i) {
            if (boxes[i + 1] == '1') {
```

operationSum += ballCount; // Accumulate the number of operations needed from right side up to the current box.

};

/**

TypeScript

```
# Initialize a counter for the number of operations moving from left to right
        left_to_right_count = 0
       # Traverse the boxes from left to right to calculate the number of operations needed
       for i in range(1, num boxes):
           # If the previous box contains a ball, increment the count
           if boxes[i - 1] == '1':
                left to right count += 1
           # The current box operation count is the previous box's count plus
           # the number of operations carried over (cumulative)
            operations[i] = operations[i - 1] + left_to_right_count
       # Reset the counter for the number of operations when moving from right to left
       right to left count = 0
       # Reset a variable to store the sum of operations when moving right to left
       sum_operations = 0
       # Traverse the boxes from right to left to add remaining operations
       for i in range(num boxes -2, -1, -1):
           # If the next box contains a ball, increment the count
           if boxes[i + 1] == '1':
               right to left count += 1
           # Accumulate the sum of operations from right to left
           sum operations += right to left count
           # Add the number of operations from the right to the current answer
           operations[i] += sum_operations
       return operations
Time and Space Complexity
  The given code consists of two separate for loops that each iterate through the array boxes. Each loop runs in linear time relative
  to the number of elements n in the input string boxes.
```

num boxes = len(boxes)

operations = [0] * num boxes

Time Complexity

0(n).

The first loop runs from index 1 to n, with constant-time operations within the loop, leading to an O(n) complexity. The second loop similarly runs with constant-time operations but in reverse order, from n-2 down to 0. This loop also results in an O(n) complexity.

Space Complexity The space complexity is determined by the additional memory used by the algorithm besides the input. In this case, the algorithm only allocates space for one additional array, ans, which stores the result and has the same length as the input string boxes.

Thus, the space complexity is directly proportional to the length of the input, resulting in a space complexity of O(n).

Since these loops do not nest and run in sequence, the overall time complexity of the function is 0(n) + 0(n), which simplifies to