2938. Separate Black and White Balls

Medium Greedy Two Pointers String

Problem Description

In this problem, we're given `n` balls on a table, colored black or white, and represented by a binary string `s` of length `n`. The character `1` represents a black ball, while `0` represents a white ball. The objective is to group all the black balls to the right side of the string and all the white balls to the left side, using a series of adjacent swaps. Each step allows a swap of two neighboring balls. The final goal is to determine the minimum number of such swaps needed to achieve the segregation of the balls into their respective groups by color.

Intuition

The key insight behind the solution is to count the number of steps it takes to bring each black ball (represented by `1`) to its final position on the right side. Due to the nature of the swaps, every time we move a black ball one position to the right, one swap is required. However, as we progress in grouping the black balls to the right, each successive black ball we encounter will have to be moved a few positions less than the previous one because of the black balls that we have already moved.

To calculate the number of moves efficiently, we simulate this process in reverse order by iterating from the right to the left. We use a variable `cnt` to keep track of the black balls that have been encountered, and we use `ans` to sum up the moves needed. With every black ball encountered, we increase `cnt` by one since one more black ball is now in its right-most position. We add `n - i - cnt` to `ans` each time we find a black ball because the ball needs to pass over `n - i - cnt` black balls that have already been placed to the right.

By moving from right to left, we can calculate the minimum steps with a single pass through the string, making the process efficient and avoiding the need for simulating actual swaps.

Solution Approach

The implementation of the solution starts by initializing two variables, `cnt` and `ans`, both set to 0. The variable `cnt` will keep track of the number of black balls (`1`s) we have encountered starting from the rightmost side, while `ans` will accumulate the total number of steps (swaps) required to move all black balls to the right.

We begin with a reverse iteration of the string, from the last character to the first (index `n-1` to `0`). We traverse in reverse because we're interested in calculating the number of steps a black ball needs to move from its current position to its final destination, which we can easily compute once we know how many black balls are already positioned to the right of it.

During the iteration, for each black ball we encounter, which is marked by a `1` in string `s`:

- We increment `cnt` by 1, representing another black ball that is in its appropriate position on the right.
- We calculate `n - i - cnt`, which gives us the number of swaps needed to move this particular black ball past the `cnt` black balls already in place. Note that `i` is the current index in the iteration. We do this because each black ball already placed means one less swap needed for the current one.
- We add this calculated number of swaps to `ans`, which is our total.

The algorithm does not use auxiliary data structures, thereby maintaining a space complexity of `O(1)`. The time complexity is `O(n)`, as we only need to traverse the string once to count the necessary swaps.

Here's the reference approach, expressed as pseudocode to describe the process:

```
function minimumSteps(s: string) -> integer:
    n = length(s)
    cnt = 0
    ans = 0
    for i from n-1 to 0:
        if s[i] == '1':
            cnt = cnt + 1
            ans = ans + (n - i - cnt)
    return ans
```

The beauty of this solution lies in its simplicity. By intelligently choosing to iterate in reverse and keep track of the number of encountered black balls, we can calculate the solution in a single pass without simulating the swaps, which could have been significantly costlier in terms of performance.

Example Walkthrough

Let's walk through an example to better understand the solution approach using the problem content above.

Imagine we have the binary string `10101`, representing 5 balls where the `1`s are black balls, and the `0`s are white balls. Using the solution approach, we want to calculate the minimum number of swaps needed to group all black balls to the right.

Following the steps of the algorithm:

1. We start with `cnt` and `ans` initialized to 0.
2. The string `10101` has length `n = 5`.
3. We iterate in reverse, starting from the last character (index 4) to the first (index 0).
   - i = 4, s[4] = `1` (black ball): We increment `cnt` to 1. `ans` becomes `ans + (5 - 4 - 1)` which is `ans + 0` (no swap needed because it is already in the last position).
   - i = 3, s[3] = `0` (white ball): No action taken because we only count swaps for black balls. `cnt` and `ans` are unchanged.
   - i = 2, s[2] = `1` (black ball): We increment `cnt` to 2. `ans` becomes `ans + (5 - 2 - 2)` which is `ans + 1` (one swap needed to move to the end).
   - i = 1, s[1] = `0` (white ball): Again, no action is needed for white balls. No changes to `cnt` or `ans`.
   - i = 0, s[0] = `1` (black ball): Increment `cnt` to 3. `ans` becomes `ans + (5 - 0 - 3)` which is `ans + 2` (two swaps needed to move to the end).
4. Adding these up, `ans = 0 + 1 + 2` which equals `3`.

Therefore, for the given string `10101`, it will take a **minimum of 3 swaps** to move all the black balls to the right side of the string. This matches the logic of our solution approach. The black balls initially at indices 0 and 2 need to each move past two white balls to reach the far right, with the last black ball at index 4 already in the correct position.

Solution Implementation

Python

```python
class Solution:
    def minimumSteps(self, s: str) -> int:
        # Length of the given string
        length = len(s)

        # Initialize 'answer' for holding the minimum steps and
        # 'one_count' for keeping track of the number of '1's encountered
        answer = 0
        one_count = 0

        # Traverse the string in reverse from last to first character
        for i in range(length - 1, -1, -1):
            # Check if the current character is '1'
            if s[i] == '1':
                # If it's '1', increment the 'one_count'
                one_count += 1
                # Update answer by how many steps needed to move this '1' to the end
                # considering the number of ones already counted.
                answer += (length - i - 1) - one_count + 1

        # Return the total minimum steps calculated
        return answer
```

Java

```java
class Solution {
    /**
     * Calculates the minimum number of steps to move all '1's to the right end of the string
     *
     * @param s the input string representing a binary number
     * @return the minimum number of steps required
     */
    public long minimumSteps(String s) {
        // Initialize the answer to 0
        long answer = 0;
        // Counter for the number of '1's found
        int oneCount = 0;
        // Length of the string
        int length = s.length();

        // Iterate over the string from right to left
        for (int i = length - 1; i >= 0; --i) {
            // Check if the current character is '1'
            if (s.charAt(i) == '1') {
                // Increase the count of '1's
                ++oneCount;
                // Add the number of steps to move this '1' to the right end
                answer += length - i - oneCount;
            }
        }
        // Return the total number of steps calculated
        return answer;
    }
}
```

C++

```cpp
#include <string>

class Solution {
public:
    // function to calculate the minimum number of steps required.
    long long minimumSteps(std::string s) {
        long long steps = 0; // Variable to store the minimum steps.
        int onesCount = 0; // Counter for the occurrences of '1'.
        int length = s.size(); // Get the size of the string.

        // Traverse the string from the end to the beginning.
        for (int i = length - 1; i >= 0; --i) {
            if (s[i] == '1') { // If the current character is '1'.
                ++onesCount; // Increment the count of '1's.

                // Accumulate the distance from right-most end minus
                // the number of '1's encountered so far.
                steps += length - i - onesCount;
            }
        }
        // Return the computed minimum steps.
        return steps;
    }
};
```

TypeScript

```typescript
function minimumSteps(s: string): number {
    const stringLength = s.length;          // Get the length of the input string.
    let steps = 0;                          // Initialize the steps counter to zero.
    let countOfOnes = 0;                    // Initialize a counter to keep track of the number of '1's encountered.

    // Iterate over the string in reverse order.
    for (let i = stringLength - 1; i >= 0; --i) {
        // The `~` before `i` is a bitwise NOT operator, used here as a shorthand for (i !== -1).
        if (s[i] === '1') {
            // If the current character is '1', increment the count of '1's.
            ++countOfOnes;
            // Accumulate the necessary steps to bring the '1' to the end of the string, taking into account
            // the count of '1's encountered so far, which do not need to be moved.
            steps += stringLength - i - countOfOnes;
        }
    }

    // Return the total number of steps required to move all '1's to the end of the string.
    return steps;
}
```

```python
class Solution:
    def minimumSteps(self, s: str) -> int:
        # Length of the given string
        length = len(s)

        # Initialize 'answer' for holding the minimum steps and
        # 'one_count' for keeping track of the number of '1's encountered
        answer = 0
        one_count = 0

        # Traverse the string in reverse from last to first character
        for i in range(length - 1, -1, -1):
            # Check if the current character is '1'
            if s[i] == '1':
                # If it's '1', increment the 'one_count'
                one_count += 1
                # Update answer by how many steps needed to move this '1' to the end
                # considering the number of ones already counted.
                answer += (length - i - 1) - one_count + 1

        # Return the total minimum steps calculated
        return answer
```

Time and Space Complexity

Time Complexity

The given code involves a single for-loop that iterates over the string `s` in reverse order, starting from the last character to the first. Since each element of the string `s` is processed exactly once during this iteration, the time complexity of this operation is `O(n)`, where `n` is the length of the string `s`. Inside the loop, only constant time operations are performed (i.e., basic arithmetic operations and conditional checks), which do not depend on the size of the input, therefore they don't affect the overall linear time complexity.

Space Complexity

The algorithm uses a fixed number of variables (`n`, `ans`, `cnt`, and `i`) that do not scale with the size of the input. This implies that the space complexity is constant. Therefore, the space complexity of the code is `O(1)`, as the amount of memory used does not increase with the size of the input string `s`.