1864. Minimum Number of Swaps to Make the Binary String Alternating

<u>Greedy</u> Medium String

In this problem, we are given a binary string s which only contains '0's and '1's. Our goal is to find the minimum number of

Problem Description

character swaps required to convert the string into an alternating binary string, where no two adjacent characters are the same.

For example, "010101" and "101010" are alternating strings, but "110" is not as there are two '1's adjacent to each other.

If making the string alternating is not possible, we should return -1. A character swap means choosing any two characters in the string (not necessarily adjacent) and exchanging their positions. The

challenge is to do the minimum number of such swaps to achieve an alternating pattern.

extra characters of one type that we cannot place without creating adjacent duplicates.

To arrive at the solution, we first need to understand that there are only two possible alternating patterns for any string:

Intuition

"010101..." or "101010...". Also, for a binary string that can be made alternating, the difference between the counts of '0's and '1's cannot be more than one. If the difference is more than one, it is impossible to form an alternating string because there would be

We need to track the following: • sono: Number of '0's that are in the wrong place if we are trying to form the "010101..." pattern. • son1: Number of '1's that are in the wrong place if we are trying to form the "010101..." pattern. • s1n0: Number of '0's that are in the wrong place if we are trying to form the "101010..." pattern. • s1n1: Number of '1's that are in the wrong place if we are trying to form the "101010..." pattern.

With this understanding, the next insight is to count the number of misplaced '0's and '1's in both possible alternating patterns.

For a valid alternating string:

- The counts of '0's and '1's should be equal, or there should be exactly one more '0' or exactly one more '1'. • The numbers of misplaced '0's should equal the numbers of misplaced '1's for a given pattern. That is, sono should equal son1 and s1no should
- equal s1n1. If they are not equal, it means we can't swap a '0' with a '1' to correct the string since there's an unequal amount to swap. After counting, we have the following cases:

- If both the counts of sono and son1 aren't equal, and the counts of sino and sini aren't equal either, we return -1 because it's impossible to make the string alternating. • If sono and son1 aren't equal, which means we can't form "010101..." pattern, we should check if we can form "101010..." pattern and return
- s1n0 (or s1n1 since they are equal). • If s1n0 and s1n1 aren't equal, meaning we can't form "101010..." pattern, we should return s0n0 (or s0n1 since they are equal). • If we can form both "010101..." and "101010..." patterns, we return the minimum of sono and sino.

Therefore, by counting the number of misplaced characters and comparing them, we can determine the minimum number of

swaps required. If any configuration allows for alternating characters, that minimum count is the answer. If neither does, then it is impossible to form an alternating string, and we return -1.

■ If it is a '0', it's wrong for "010101..." and we increment s0n1.

O(n) time complexity and O(1) space complexity, where n is the length of string s.

■ It's not '0', so it's misplaced for the "010101..." pattern. Increment sono to 1.

■ Since it's '1', it's correctly placed for the "101010..." pattern, so s1n1 remains 0.

■ Since it's not '1', it's misplaced for the "101010..." pattern. Increment s1n1 to 1.

■ It's not '0', so it's misplaced for the "010101..." pattern. Increment son1 to 1.

■ Since it's '1', it's correctly placed for the "101010..." pattern, so s1n0 remains 1.

Initialize our counters: s0n0 = 0, s0n1 = 0, s1n0 = 0, s1n1 = 0.

For the first character (index 0, even), we have '1'.

For the third character (index 2, even), we have '0'.

For the fourth character (index 3, odd), we have '1'.

■ It's correctly placed for the "010101..." pattern, so sono remains 1.

We begin iterating through the string:

Initialize four counters: sono, son1, s1no, s1n1 to zero. These will count the number of misplaced '0's and '1's for both potential alternating patterns "010101..." (sono and son1) and "101010..." (s1no and s1n1).

The solution provided follows a simple but effective approach without the need for any complex algorithms or data structures.

Iterate through the string s using a for-loop and the range function, checking each character:

○ If we're at an even index (using i & 1 to check if i is odd or even), we compare the character with '0'.

■ If it's not '0', then it's in the wrong place for the "010101..." pattern, so we increment sono.

■ If it is a '0', then it's in the wrong place for the "101010..." pattern, so we increment s1n1.

If we're at an odd index, we do the reverse: ■ If the character is not '0', it's wrong for "101010..." and we increment s1n0.

Solution Approach

Here's the breakdown:

After the loop, we have the counts of misplaced '0's and '1's for both patterns. We then examine these counts: ∘ If sono does not equal son1, and s1no does not equal s1n1, we can't form an alternating string. We return -1.

∘ If sono does not equal son1, we know that the "010101..." pattern is not possible, but the "101010..." pattern is, so we return s1no (or s1n1)

as they are the same). Conversely, if s1n0 does not equal s1n1, we can't form "101010...", but we know "010101..." is possible, so we return s0n0 (or s0n1 since they are equal). • If both sets of counts are equal, meaning either pattern could be formed, we return the minimum of sono and sino.

This approach ensures we find the minimum swaps needed for either pattern if it's possible to build an alternating string out of s.

It efficiently utilizes bitwise operations and simple if-else constructs without the need for additional space, hence operating in

Let's use a small example to illustrate the solution approach. Consider the binary string s = "1001". We want to determine the

Example Walkthrough

minimum number of swaps required to make this string into an alternating binary string, either "1010" or "0101".

For the second character (index 1, odd), we have '0' ■ It's not '1', so it's misplaced for the "101010..." pattern. Increment s1n0 to 1. ■ Since it's '0', it's correctly placed for the "010101..." pattern, so son1 remains 0.

After iterating, our counts are as follows: s0n0 = 1, s0n1 = 1, s1n0 = 1, s1n1 = 1.

Python

class Solution:

We examine our counts:

Since sono equals son1, and sino equals sini, either pattern "010101..." or "101010..." can be formed. We return the minimum of s@n@ and sln@, which is min(1, 1) = 1.

Thus, it only takes a single swap to turn the string "1001" into an alternating binary string. We can swap the second and third

Solution Implementation

swaps 0 to 1 - number of swaps needed if the even-index should be '0'

swaps 1 to 0 - number of swaps needed if the even-index should be '1'

If current character should be '0' on even index, but it's not

If current character should be '1' on odd index, but it's not

If current character should be '0' on odd index, but it's '1'

// Initialize counters to track the number of swaps required for each pattern.

// Pattern "01" requires 'swapCountPattern01' and 'swapCountPattern10' swaps.

// Pattern "10" requires 'swapCountPattern10' and 'swapCountPattern01' swaps.

For the swaps to be possible, the number of required swaps for both scenarios must be the same

If the total number of swaps is equal, one of them must be even since it's impossible to have an odd number of swaps for bo

Return the minimum number of swaps if both are even, otherwise, return the even count since the odd count will require an ϵ

If they are not, it's impossible to create a valid string of alternate characters by swapping

For the strings to be valid they should alternate '01' or '10'.

If it's not possible to create a valid string, return -1.

Initializing counters for each possible scenario:

Iterate over each character in the string

Check if the current index is even

swaps_1_to_0 += 1

swaps 0 to 1 += 1

swaps_1_to_0 += 1

if (swaps 0 to_1 % 2) != (swaps_1_to_0 % 2):

return min(swaps_0_to_1, swaps_1_to_0) // 2

return min(swaps_0_to_1//2, swaps_1_to_0//2)

if s[index] != '1':

if s[index] != '0':

characters to obtain "1010" or swap the first and second characters to obtain "0101".

if s[index] != '0': swaps 0 to 1 += 1 # If current character should be '1' on even index, but it's '0' **if** s[index] != '1':

else:

return -1

if (swaps 0 to 1 % 2) == 0:

public int minSwaps(String s) {

int swapCountPattern01 = 0;

int swapCountPattern10 = 0;

// Calculate the number of '0's directly.

for (int i = 0; i < length; i++) {</pre>

currentSwapCount++;

for (int i = 0; i < length; i++) {</pre>

currentSwapCount++;

return minCount == INT_MAX ? -1 : minCount;

const int halfLength = length / 2;

const int numberOfZeros = length - numberOfOnes;

int minCount = INT MAX; // Use maximum integer to initialize minimum count.

// Case for strings that should start with '1' (e.g., '1010' or '101').

int currentSwapCount = 0; // Swaps needed for current iteration.

// Case for strings that should start with '0' (e.g., '0101' or '010').

int currentSwapCount = 0; // Swaps needed for current iteration.

* Determines the minimum number of swaps to make a binary string alternating.

// If no valid scenario was found, return -1; otherwise, return the minimum swap count found.

if (numberOfZeros == (length + 1) / 2 && numberOfOnes == length / 2) {

if (numberOfOnes == (length + 1) / 2 && numberOfZeros == length / 2) {

if (i % 2 == 0 && binaryString[i] != '1') {

if (i % 2 == 0 && binaryString[i] != '0') {

minCount = std::min(minCount, currentSwapCount);

minCount = std::min(minCount, currentSwapCount);

def minSwaps(self, s: str) -> int:

swaps_0_to_1 = swaps_1_to_0 = 0

for index in range(len(s)):

if (index % 2) == 0:

```
Java
class Solution {
```

else:

```
// Loop through the string to count the number of swaps needed.
        for (int i = 0; i < s.length(); ++i) {</pre>
            // If the index 'i' is even, we expect a '0' for pattern "01" and a '1' for pattern "10".
            if ((i & 1) == 0) {
                if (s.charAt(i) == '1') {
                    swapCountPattern01 += 1;
                } else {
                    swapCountPattern10 += 1;
            } else {
                // If the index 'i' is odd, we expect a '1' for pattern "01" and a '0' for pattern "10".
                if (s.charAt(i) == '1') {
                    swapCountPattern10 += 1;
                } else {
                    swapCountPattern01 += 1;
        // If the number of swaps needed for both patterns is not the same, it's impossible to achieve the pattern.
        if (swapCountPattern01 != swapCountPattern10) {
            return -1;
        // If only one pattern is possible, return the number of swaps needed for that pattern.
        if (swapCountPattern01 != swapCountPattern10) {
            return swapCountPattern10;
        // If both patterns are possible, return the minimum number of swaps.
        return Math.min(swapCountPattern01, swapCountPattern10);
C++
#include <string>
#include <algorithm>
/**
 * Determines the minimum number of swaps to make a binary string alternating.
 * It only considers valid scenarios where the number of 1's and 0's differ by at most one.
 * @param binaryString - The binary string to be processed.
 * @return The minimum number of swaps, or -1 if not possible.
int minSwaps(const std::string& binaryString) {
    const int length = binaryString.length();
    // Count the number of '1's.
    const int numberOfOnes = std::count(binaryString.begin(), binaryString.end(), '1');
```

/**

TypeScript

```
* It only considers valid scenarios where the number of 1's and 0's differ by at most one.
* @param {string} binaryString - The binary string to be processed.
* @return {number} - The minimum number of swaps, or -1 if not possible.
function minSwaps(binaryString: string): number {
   const length: number = binaryString.length;
   const numberOfOnes: number = Array.from(binaryString).reduce((accumulated, current) => parseInt(current) + accumulated, 0);
   const numberOfZeros: number = length - numberOfOnes;
   let minCount: number = Infinity;
   const halfLength: number = length / 2;
   // Case for strings that should start with '1' (e.g. '1010' or '101')
   if (numberOfOnes === Math.ceil(halfLength) && numberOfZeros === Math.floor(halfLength)) {
        let currentSwapCount: number = 0: // Swaps needed for current iteration
       for (let i = 0; i < length; i++) {</pre>
           if (i % 2 === 0 && binaryString.charAt(i) !== '1') currentSwapCount++;
       minCount = Math.min(minCount, currentSwapCount);
   // Case for strings that should start with '0' (e.g. '0101' or '010')
   if (numberOfZeros === Math.ceil(halfLength) && numberOfOnes === Math.floor(halfLength)) {
        let currentSwapCount: number = 0: // Swaps needed for current iteration
       for (let i = 0; i < length; i++) {</pre>
           if (i % 2 === 0 && binaryString.charAt(i) !== '0') currentSwapCount++;
       minCount = Math.min(minCount, currentSwapCount);
   // If no valid scenario was found, return -1, otherwise return the minimum swap count found
   return minCount === Infinity ? -1 : minCount;
class Solution:
   def minSwaps(self, s: str) -> int:
       # Initializing counters for each possible scenario:
       # swaps 0 to 1 - number of swaps needed if the even-index should be '0'
       # swaps 1 to 0 - number of swaps needed if the even-index should be '1'
       # For the strings to be valid they should alternate '01' or '10'.
       # If it's not possible to create a valid string, return -1.
       swaps 0 to 1 = swaps 1 to 0 = 0
       # Iterate over each character in the string
       for index in range(len(s)):
           # Check if the current index is even
           if (index % 2) == 0:
               # If current character should be '0' on even index, but it's not
               if s[index] != '0':
                   swaps 0 to 1 += 1
               # If current character should be '1' on even index, but it's '0'
               if s[index] != '1':
                   swaps_1_to_0 += 1
           else:
               # If current character should be '1' on odd index, but it's not
               if s[index] != '1':
                   swaps 0 to 1 += 1
               # If current character should be '0' on odd index, but it's '1'
               if s[index] != '0':
                   swaps_1_to_0 += 1
       # For the swaps to be possible, the number of required swaps for both scenarios must be the same
       # If they are not, it's impossible to create a valid string of alternate characters by swapping
       if (swaps 0 to_1 % 2) != (swaps_1_to_0 % 2):
```

The given code iterates through the string s once, which means that the loop runs for n iterations, where n is the length of the string s. Within each iteration of the loop, the code performs a constant number of operations, such as comparison, bitwise AND,

Time Complexity

return -1

else:

if (swaps 0 to 1 % 2) == 0:

Time and Space Complexity

return min(swaps_0_to_1, swaps_1_to_0) // 2

return min(swaps_0_to_1//2, swaps_1_to_0//2)

proportional to the length of the string, which gives us a time complexity of O(n). **Space Complexity** Regarding space complexity, the code allocates a constant amount of extra space. It uses four integer variables sono, son1, s1n0, and s1n1 to keep track of counts, no matter how large the input string s is. Since these variables do not depend on the

and increment operations, all of which take constant 0(1) time. Therefore, the time complexity of the entire function is directly

If the total number of swaps is equal, one of them must be even since it's impossible to have an odd number of swaps for bo

Return the minimum number of swaps if both are even, otherwise, return the even count since the odd count will require an e

size of the input, and no other significant space is used (no dynamic data structures or arrays are allocated), the space complexity is constant, 0(1).