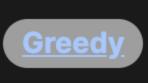
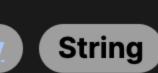
1529. Minimum Suffix Flips







Problem Description

You have a binary string called target that is indexed starting from 0 and has a length of n. At the beginning, there's another binary string named s which also has a length of n but is set to all zeroes. The goal is to change s so that it eventually matches target.

To turn s into target, there's a specific operation you can do: pick any index i (where i can range from 0 to n - 1), and then flip every bit from that index to the end of the string (index n - 1). To flip a bit means if it's a 0, it becomes a 1, and vice versa.

Your task is to figure out the minimum number of these operations that are needed to make the string s exactly the same as target.

Intuition

an index i to the end, each operation essentially toggles the state of the remainder of the string from that index on. If you flip twice at the same point, you get back to the original state, making the operation redundant. Therefore, the main insight is that you only need to perform a flip when you encounter a bit in target that differs from the current

To solve this problem efficiently, you observe the pattern of operations needed to change s into target. When you flip bits from

state of s at that index. This state is represented by the number of flips you've done before - if it's even, the state matches the initial; if it's odd, the state is flipped. At the beginning, s is all zeros, so you need to flip whenever target[i] is 1 and the flip count is even, or target[i] is 0 and the flip count is odd. In other words, you flip whenever the current bit differs from the expected bit that would result from all previous flips. The code iterates through target, checking at each position if a flip is required by comparing the state with the current bit v of

target. It uses the bitwise XOR ^ operator to compare the bit at position i with the parity (0 or 1) of the count of flips made so far (represented by the ans variable). The expression (ans & 1) ^ int(v) evaluates to 1 (true) when a flip is needed and 0 (false) otherwise. If a flip is needed, the solution increments the flip count ans.

change s into target. Solution Approach

In essence, the number of times the condition becomes true (which triggers a flip) is the minimum number of flips needed to

To implement the solution, we need to keep track of the current state of flips performed on string s so that we can determine when a flip operation is actually needed as we iterate through each bit of the target string.

The Python code defines a class Solution with a method minFlips which takes a single argument target, a string representing the target binary configuration.

Here's the step-by-step implementation: Initialize a variable ans to 0. This variable will hold the number of flips performed.

Iterate over each character v in the target string. On each iteration, v represents the current bit in the target configuration we

- want to achieve.
- Check whether we need to flip starting from the current bit index to the end of the binary string. We do this by comparing the least significant bit of ans (which keeps track of the number of flips and therefore the current state of s) to the current bit v in
- the target (int(v) converts the bit character to an integer). The comparison is performed with the expression (ans & 1) ^ int(v). If we've performed an even number of flips (ans is even), the least significant bit of ans is 0; if odd, it is 1. o If the result of the comparison is 1, it means the current bit of s is not the same as the bit in target, hence a flip operation must be performed. ∘ If the result of the comparison is ∅, no flip operation is required at this point, as the current bit of s already matches the target.
 - If a flip is required, increment ans by 1. This not only records a new flip operation but also changes the current state of which bit would result if a flip were done at the next different bit in target.
- After iterating through all bits in target, return the value of ans. This final value represents the total number of flips necessary to transform s into target.
- time complexity of O(n), where n is the length of the target string. No additional data structures are used, as the state of the string can be inferred from the number of flips, resulting in a constant O(1) space complexity.

The implementation uses a linear scan of the input string, and a bitwise operation to determine when to flip, leading to an overall

Example Walkthrough Let's walk through a small example to illustrate the solution approach using the target string target = "001011".

Initially, we have s = "0000000" and we want to transform it to s = "001011" using the minimum number of operations. We'll start

with ans = 0, as no flips have been made yet.

and s is updated to "000011".

Now we'll go through the target string bit by bit and decide whether to flip based on the current state of ans.

For index i = 0, the target bit is 0. Since ans is 0 (even), the least significant bit of ans is also 0. We compute (0 & 1) ^ 0, which is 0, meaning no flip is needed because s [0] is already 0.

For index i = 1, the target bit is 0. The calculation (0 & 1) ^ 0 is still 0, so no flip needed. s remains unchanged.

At index i = 2, the target bit is 1. We compute (0 & 1) ^ 1, which is 1, indicating a flip is required. We increment ans to 1,

- and now s is "111111". For index i = 3, the target bit is 0. The calculation (1 & 1) $^{\circ}$ 0 is 1, meaning another flip is necessary. We increment ans to 2,
- 3, and s changes to "001111". Finally, at index i = 5, the target bit is 1. The calculation (3 & 1) ^ 1 is 0, so no flip is needed, and s remains at "001111".

At index i = 4, the target bit is 1. We compute (2 & 1) ^ 1, which is 1, indicating yet another flip is required. ans increases to

"001011". To summarize, we performed flips at indices [2, 3, 4] to achieve the target configuration, leading us to return the ans value of 3

After going through all the bits in the target, we conclude that a minimum of 3 flips is required to change s from "000000" to

as the result. This walkthrough illustrates how a simple, linear scan through the target string, combined with bitwise XOR and

AND operations, can efficiently determine the minimum number of flips needed.

Return the total number of flips required to obtain the target state

// If a flip is required, increment the flip count.

// Function that returns the minimum number of flips needed to achieve the target state

let bulbState = parseInt(target[i], 10); // Convert character to integer (0 or 1)

// If the current number of flips results in a different state than the target bulb state,

let flipsCount = 0; // Initialize counter for minimum number of flips

// Return the total flips made to achieve the target state.

Iterate over each character in the target string

if (flip_count & 1) ^ int(bulb_state):

if (((flips & 1) ^ value) != 0) {

++flips;

function minFlips(target: string): number {

for (let i = 0; i < target.length; i++) {</pre>

// Loop through each character in the target string

flip_count += 1

Python class Solution: def minFlips(self, target: str) -> int: # Initialize the flip counter to zero

for bulb_state in target: # Check if the number of flips made is odd (flip_count & 1) # and compare with the current bulb state (int(bulb_state) == 1 if it's '1', 0 otherwise). # If there's a mismatch between the current state after flips and the target bulb state, # it indicates that another flip is needed.

flip_count = 0

Solution Implementation

```
return flip_count
Java
class Solution {
   // Function to find the minimum number of flips required to make the bulb status string equal to the target.
    public int minFlips(String target) {
       // 'flips' counts the number of flips made.
       int flips = 0;
       // Iterate over each character of the target string.
       for (int i = 0; i < target.length(); ++i) {</pre>
           // Convert the current character to an integer value (0 or 1).
            int value = target.charAt(i) - '0';
           // If the current flip state is different from the current target bulb state,
           // a flip is required.
           // (flips & 1) finds the current state after an even or odd number of flips
           // The ^ (XOR) operator compares that state with the desired value (value).
           // When they are different, the result is 1 (true); otherwise, it's 0 (false).
```

```
C++
```

```
return flips;
class Solution {
public:
   int minFlips(string target) {
        int flipsCount = 0; // Initialize counter for minimum number of flips
       // Loop through each character in the target string
        for (char state : target) {
            int bulbState = state - '0'; // Convert character to integer (0 or 1)
            // When the current number of flips results in a different state than the target bulb state,
            // increment the flip count. The ^ operator is a bitwise XOR used for comparison here.
            if ((flipsCount & 1) ^ bulbState) {
                ++flipsCount;
       // Return the final count of flips required to achieve the target state
        return flipsCount;
};
TypeScript
```

// increment the flip count. The ^ operator is a bitwise XOR used for comparison here. if ((flipsCount & 1) ^ bulbState) { flipsCount++;

```
// Return the final count of flips required to achieve the target state
      return flipsCount;
class Solution:
   def minFlips(self, target: str) -> int:
       # Initialize the flip counter to zero
       flip_count = 0
       # Iterate over each character in the target string
       for bulb_state in target:
           # Check if the number of flips made is odd (flip_count & 1)
           # and compare with the current bulb state (int(bulb_state) == 1 if it's '1', 0 otherwise).
           # If there's a mismatch between the current state after flips and the target bulb state,
           # it indicates that another flip is needed.
           if (flip_count & 1) ^ int(bulb_state):
               flip_count += 1
       # Return the total number of flips required to obtain the target state
       return flip_count
Time and Space Complexity
Time Complexity
```

Space Complexity

space complexity is 0(1).

The time complexity of the code is determined by how many times we iterate through the string target. There is a single for loop that goes through the length of the target string, which is of length n. Each iteration does constant-time operations such as checking the condition and possibly incrementing ans. Hence, the overall time complexity is O(n).

The space complexity is determined by the amount of extra space used apart from the input itself. In this case, only a finite

number of variables (ans and v) are used which occupy constant space regardless of the length of the input string. Thus the