1702. Maximum Binary String After Change

Medium Greedy String

Problem Description

You are given a binary string binary that contains only the characters '0' and '1'. Two types of operations can be performed on this string an unlimited number of times: Operation 1: If the string contains the substring "00", you can replace it with "10".

- Operation 2: If the string contains the substring "10", you can replace it with "01".

end up. After that, it constructs the string based on the counts.

of the maximum binary string after all possible operations.

the index of the first occurrence of a substring—in this case, '0'.

Here's the key part of the code with comments explaining each operation:

Increase `k` by the count of '0's following the first '0'.

This simulates the position of the last '0' in the string.

Construct the final binary string with the calculated numbers

Your task is to determine the highest-value binary string possible after any number of these operations. Here, the "value" of a binary string is defined by its decimal equivalent. For example, the binary string "1010" has a decimal equivalent of 10, while

"1100" has a decimal equivalent of 12. The goal is to strategically apply these operations to the initial string to transform it into the highest possible numeric value in

Intuition To solve this problem, we should clearly understand the effect of each operation. Let's analyze the operations:

• Operation 1 ("00" -> "10"): This operation increases the value of the binary number. So whenever we see "00", we want to perform this

binary form.

operation.

to move '1' bits towards the end of the string (right-most side).

- Intuitively, we can see that to maximize the binary value, we should have the most number of '1's towards the left side as
- possible, except possibly one '0' which will be necessary if we started with at least one '0'. Here are the steps to find the solution:

• Operation 2 ("10" -> "01"): This operation does not increase the value; instead, it shifts the higher value bit ('1') to the right. We may use this

Find the first occurrence of '0'. If there is no '0' in the string, the binary string is already at its maximum value, so return it as is.

Count the number of '0's after the first '0'. Every '0' except the last one can be turned into a '1' by using Operation 1 followed by Operation 2 as needed to shift the '1's to the left.

position where the last '0' should be, then one '0', followed by '1's for the rest of the string. The code achieves this by first looking for the first '0' then counting all subsequent '0's, which tells us where the last '0' will

Knowing how many '0's there are in total, we can build the resulting string. There will be a sequence of '1's up to the

Solution Approach The solution approach is fairly straightforward with a focus on string manipulation. The key insight here is realizing the final form

The string will be composed of three sections: A prefix of all '1's up to the position before the last '0'.

Knowing this final form allows us to skip simulating each operation and directly construct the final maximum binary string. Here is how the algorithm does it step by step:

a single Python method.

return binary

k += binary[k + 1 :].count('0')

of '1's, the last '0', and then '1's again.

return '1' * k + '0' + '1' * (len(binary) - k - 1)

A suffix of all '1's after the last '0'.

• A single '0'.

If no '0' is found, it means the string is already the maximum binary string possible (since it's composed of only '1's), so the original string is returned.

If a '0' is found, we calculate the index k where the last '0' would be after all possible operations. This is done by first

setting k to the index of the first '0' found, then increasing k by the count of remaining '0's in the string after the first '0'.

The given binary string is first checked for the presence of '0'. This is done using Python's string find method, which returns

This effectively "moves" the last '0' to its final position. Now that we know where the last '0' will be, we construct the final string. This is done by concatenating the following:

∘ '1' * k, which creates a string of '1's that has the same length as the number of '1's that should be on the left side of the last '0'.

○ '0', which places the last '0'. ○ '1' * (len(binary) - k - 1), which creates the suffix consisting of the remaining number of '1's after the last '0'. And that is it. There are no complex data structures involved—only simple string operations. The algorithm is efficient because it

calculates the final string in one pass without actually performing the transformations, and the entire logic is implemented within

class Solution: def maximumBinaryString(self, binary: str) -> str: # Find the first '0' in the string. k = binary.find('0') # If no '0' is found, return the original string as-is. if k == -1:

Using the count of '0's and string concatenation operations results in a clean, efficient solution that avoids cumbersome loops or

We first search for the first occurrence of the character '0'. In this string, it occurs at index 0. The string binary find('0')

Next, we count the number of '0's after the first '0'. In the string "00110", there are three '0's in total. After finding the first

We set k to the index of the first '0' found which is 0 and then increase k by the count of remaining '0's, which is 2.

Therefore, k becomes 0 + 2 = 2. This is the index where the last '0' will be after performing all the operations possible.

Now, we construct the final string. Since k is 2, we know there will be two '1's before the last '0' in the string. And since the

The final maximum binary string after all operations have been performed will be '1' * 2 + '0' + '1' * 2 which evaluates to

Let's use the binary string binary = "00110" as a small example to illustrate the solution approach.

'0', there are two more '0's to count. Hence, binary[0 + 1:].count('0') would give us 2.

original string binary has a length of 5, there will be 5 - 2 - 1 = 2 '1's following the last '0'.

conditional operations which would have been necessary if we were to simulate each operation.

Within the given code framework, it would execute as follows:

k = binary.find('0') # k = 0

Solution Implementation

k += binary[k + 1:].count('0') # k = 2

def maximumBinaryString(self, binary: str) -> str:

Construct the maximum binary string:

3. And the rest of the string filled with '1's

public String maximumBinaryString(String binary) {

int firstZeroIndex = binary.indexOf('0');

int binaryLength = binary.length();

string maximumBinaryString(string binary) {

// Find the first occurrence of '0'.

int firstZeroPos = binary.find('0');

if (firstZeroPos == string::npos) {

int binaryLength = binary.length();

if (binary[i] == '0') {

return binary;

};

/**

TypeScript

// If the string has no '0's, it's already maximized.

for (int i = firstZeroPos + 1; i < binaryLength; ++i) {</pre>

// The rest of the string, if any, will also be '1's.

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

if (binary[i] === '0') {

// Example usage of the function:

class Solution:

numOfOnesBeforeLastZero++;

// const result = maximumBinaryString("000110");

Construct the maximum binary string:

3. And the rest of the string filled with '1's

Total length stays the same as the original string

2. Followed by a single '0'

return maximum_binary

Time and Space Complexity

// console.log(result); // Output should be "111010"

numOfOnesBeforeTheLastZero++;

int numOfOnesBeforeTheLastZero = firstZeroPos; // Number of '1's before the leftmost '0'.

// Iterate through the string starting from the next position after the first '0'.

// The ultimate binary string will have a single '0' after the maximal number of '1's.

// If we find a '0', it means we can perform the operation to increase

// the count of continuous '1's at the beginning of the string.

// Calculate the new binary string based on the operations performed.

// Find the index of the first '0' in the string

// If there are no '0's in the string, return it as is

Total length stays the same as the original string

2. Followed by a single '0'

return maximum_binary

Find the index of the first '0' in the binary string

"11011".

binary = "00110"

Python

Java

class Solution {

class Solution:

Example Walkthrough

would return 0.

maximum_binary = '1' * k + '0' + '1' * (len(binary) - k - 1) # $maximum_binary = "11011"$ After these steps, maximum_binary holds the value "11011", which is the highest-value binary string possible from the initial string

original string (6). This illustrates the effectiveness of the described solution approach.

first_zero_index = binary.find('0') # If there's no '0', the binary string is already at its maximum value if first_zero_index == -1: return binary # Calculate the position for the '0' after conversion to maximum binary string

maximum_binary = '1' * zero_position_after_conversion + '0' + '1' * (len(binary) - zero_position_after_conversion - 1)

It counts the number of '0's after the first '0' found, and adds this to the index of the first '0'

zero_position_after_conversion = first_zero_index + binary[first_zero_index + 1:].count('0')

1. A string of '1's with length equal to the position of '0' after conversion

// The following loop counts how many '0's are there starting from the position

// right after the first '0' found, we will use `totalZeroCount`

// after transforming the binary string into its maximum value.

// to determine the index at which the single '0' should be placed

"00110" by applying the given operations. It has a decimal equivalent of 27, which is greater than the decimal equivalent of the

```
if (firstZeroIndex == -1) {
    return binary;
// Length of the binary string
```

```
int totalZeroCount = firstZeroIndex;
        for (int i = firstZeroIndex + 1; i < binaryLength; ++i) {</pre>
            if (binary.charAt(i) == '0') {
                ++totalZeroCount;
       // Create a character array from the binary string to facilitate manipulation
       char[] maximumBinaryCharArray = binary.toCharArray();
       // Fill the array with '1's as we want to maximize the binary value
       Arrays.fill(maximumBinaryCharArray, '1');
       // The index where the single '0' will be placed is determined by `totalZeroCount`
       // which has effectively shifted as we replaced '0's with '1's after the first '0'
       maximumBinaryCharArray[totalZeroCount] = '0';
       // Convert the resulting character array back to a string and return it
       return String.valueOf(maximumBinaryCharArray);
C++
class Solution {
public:
   // Method to find maximum binary string after operations.
   // The operation is defined as:
    // 1. Choose two consecutive bits of binary string.
    // 2. If it's "00", you can change it to "10".
```

```
* Function to find the maximum binary string after performing operations.
* An operation is defined as choosing two consecutive bits of a binary string.
* If the selected bits are "00", you can change it to "10".
* @param binary A string representing the initial binary value.
* @returns A string representing the maximum binary value possible after operations.
function maximumBinaryString(binary: string): string {
   // Find the first occurrence of '0' in the string.
   const firstZeroPos = binary.indexOf('0');
   // If the string has no '0's, it's already at the maximum value.
   if (firstZeroPos === -1) {
       return binary;
   const binaryLength = binary.length;
   // Initialize count of '1's found before the first '0'.
    let numOfOnesBeforeLastZero = firstZeroPos;
   // Iterate through the string starting from the position after the first '0'.
```

return string(numOfOnesBeforeTheLastZero, '1') + '0' + string(binaryLength - numOfOnesBeforeTheLastZero - 1, '1');

```
def maximumBinaryString(self, binary: str) -> str:
    # Find the index of the first '0' in the binary string
    first_zero_index = binary.find('0')
   # If there's no '0', the binary string is already at its maximum value
    if first_zero_index == -1:
        return binary
```

Calculate the position for the '0' after conversion to maximum binary string

1. A string of '1's with length equal to the position of '0' after conversion

// For each '0' found, we can perform the operation to transform "00" to "10",

// Construct the new binary string based on the number of operations performed.

// Any remaining characters will also be '1's, ensuring the highest possible value.

return '1'.repeat(num0f0nesBeforeLastZero) + '0' + '1'.repeat(binaryLength - num0f0nesBeforeLastZero - 1);

It counts the number of '0's after the first '0' found, and adds this to the index of the first '0'

zero_position_after_conversion = first_zero_index + binary[first_zero_index + 1:].count('0')

// The string will have a single '0' following as many '1's as possible.

// this effectively means we can increment the count of '1's before the last '0'.

Time Complexity

1. The find method called on binary to locate the first occurrence of '0' runs in 0(n) time, where n is the length of the binary string.

2. Slicing the string after the found index and counting the number of '0's with count('0') also runs in 0(n) in the worst case.

The given Python function maximumBinaryString operates on a string binary. The time complexity is analyzed as follows:

maximum_binary = '1' * zero_position_after_conversion + '0' + '1' * (len(binary) - zero_position_after_conversion - 1)

3. The multiplication and concatenation of strings return '1' * k + '0' + '1' * (len(binary) - k - 1) comprise constant-time operations for each character in the resulting string, resulting in O(n) complexity. Thus, by considering each operation, the overall time complexity of the function is O(n).

Space Complexity The space complexity of this function is considered as follows:

1. The variables k and binary itself, as inputs, do not count towards additional space, since they represent existing memory allocations and not

new space that the algorithm requires. 2. The function creates new strings that are returned, which are of length n. Therefore, the space complexity due to the output string is 0(n). Considering the above, the space complexity of the function is O(n) because of the new string that is created and returned.

Overall, the function maximumBinaryString has a time complexity of O(n) and a space complexity of O(n).