2864. Maximum Odd Binary Number

Easy Greedy Math String

Problem Description

In this problem, you are given a string `s` that represents a binary number, which is composed of the characters `'0'` and `'1'`. It is guaranteed that there is at least one `'1'` in the string `s`. Your task is to rearrange the bits (characters) in `s` to form the largest possible odd binary number. The key points to note are:

- An odd binary number always ends with the digit `'1'`.
- The largest binary number has all its `'1'`s bunched together followed by all the `'0'`s.
- Leading zeros in the binary number are allowed. The goal is to return the rearranged binary string.

Intuition

To arrive at the largest odd binary number, you want to maximize the number and position of the '1' bits in the binary string:

- Since an odd number must end with a '1', one '1' must be at the end of the string.
- To form the maximum number, the remaining '1's should be placed at the beginning of the string, maximizing their value.
- '0's contribute nothing to the value of the number, so they can be placed in the middle, between the block of '1's and the final '1'.

The solution can be arrived at by counting the number of '1's in the original string (`cnt`). Then, we rearrange the string so that we have `cnt - 1` occurrences of '1' at the beginning (since one '1' must be at the end), followed by the necessary number of '0's (which is the original length minus the number of '1's), and finally, we append a single '1' at the end to ensure the number is odd.

The Python code provided does exactly that, constructing a new binary string using string multiplication and concatenation to achieve the desired result.

Solution Approach

The solution approach is straightforward and does not use any complex algorithms or data structures, instead it relies on simple string manipulation techniques.

Here's the process step by step, as followed in the provided reference solution:

1. Count the number of '1's in the input string `s`. This is done using Python's built-in `count` method on the string object: `cnt = s.count("1")`.
2. Construct a new string with all the '1's except one at the beginning. This is `cnt - 1` number of '1's, because you're reserving one '1' for the end of the string to make sure the number is odd. In the code, this is done with `"1" * (cnt - 1)`, which repeats the `1` character `cnt - 1` times.
3. Add the '0's, using the length of the original string minus the number of '1's (`len(s) - cnt`). These '0's are the ones originally present in the string and all are bunched together here: `(len(s) - cnt) * "0"` repeats the `0` character `len(s) - cnt` times.
4. Append a '1' at the end to ensure the binary number is odd. Since the pattern for any odd binary number must end in '1', this is directly concatenated to the end with the `+ "1"` part of the code.
5. The result of the concatenation of the three parts above gives you the maximum odd binary number, which is then returned.

There are no particular patterns used except for the understanding that any binary number's value can be maximized by keeping the '1's as left (or as significant) as possible. This approach is the most effective because it is direct and takes linear time with respect to the length of the string, making it an efficient solution.

The implementation is elegant due to Python's capabilities to handle strings and perform operations like character multiplication and concatenation with such simplicity.

Example Walkthrough

Let's say we are given the binary string `s = "010101"`. We need to rearrange the bits in `s` to form the largest possible odd binary number.

1. **Count the '1's**: First, we count the number of '1's present in `s`. There are three '1's in our example: `cnt = s.count("1")`, which results in `cnt = 3`.
2. **Arrange the '1's**: We want to place `cnt - 1` number of '1's at the beginning of the new string to maximize the number's value, reserving one '1' for the end. So we have `"1" * (cnt - 1)` which is `"1" * 2` equal to `"11"`.
3. **Add the '0's**: We then add all the '0's that were originally in the string in a block in the middle. The number of '0's is the length of `s` minus the count of '1's, which is `len(s) - cnt`. In our example, this is `6 - 3 = 3`, so we get `"0" * 3`, which results in `"000"`.
4. **Ensure it's odd**: To make sure our number is odd, we need to end it with a '1'. So we append `'1'` to our string, resulting in the addition of `+ "1"` to our construction.
5. **Final result**: Combining the parts from steps 2 to 4 gives us the final rearranged string `"11" + "000" + "1" = "110001"`, which is the largest odd binary number that can be formed from `s`.

By following the outlined solution approach, the `s` has been successfully rearranged to the largest odd binary number `110001`. This approach is efficient as it relies on simple string operations, is easy to understand, and directly forms the solution in a single pass without any additional complexities.

Solution Implementation

Python

```python
class Solution:
    def maximumOddBinaryNumber(self, s: str) -> str:
        # Count the number of '1's in the binary string
        count_ones = s.count("1")

        # If there are no '1's, there's no odd number possible, return empty string
        if count_ones == 0:
            return ""

        # Otherwise, construct the maximum odd binary number by doing the following:
        # - Use all '1's except one to maintain oddness ('1' * (count_ones - 1))
        # - Fill the rest of the string with '0's (len(s) - count_ones) * "0")
        # - Add a '1' at the end to ensure the number is odd
        return "1" * (count_ones - 1) + (len(s) - count_ones) * "0" + "1"
```

Java

```java
class Solution {
    // Function to find the maximum odd binary number from the given string
    public String maximumOddBinaryNumber(String binaryString) {

        // Initialize a counter to count the number of 1s in the binary string
        int oneCount = 0;

        // Iterate through each character in the binary string
        for (char bit : binaryString.toCharArray()) {
            // Increment the counter when a '1' is found
            if (bit == '1') {
                oneCount++;
            }
        }

        // Generate the maximum odd binary number:
        // 1. Repeat '1' (oneCount - 1) times, since the last digit of an odd binary number must be 1.
        // 2. Repeat '0' for the remaining length (binaryString.length() - oneCount).
        // 3. Append '1' at the end to ensure the number is odd.
        return "1".repeat(oneCount - 1) + "0".repeat(binaryString.length() - oneCount) + "1";
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to find the maximum odd binary number based on the given string
    string maximumOddBinaryNumber(string s) {
        // Count the number of '1's in the string
        int oneCount = count_if(s.begin(), s.end(), [](char c) { return c == '1'; });

        // Initialize an empty string to build the answer
        string answer;

        // Place '1's in the answer string, one less than the count of '1's present
        for (int i = 1; i < oneCount; ++i) {
            answer.push_back('1');
        }

        // Append enough '0's to the answer to make it the same length as the original string minus 1 (for the last '1')
        for (int i = 0; i < s.size() - oneCount; ++i) {
            answer.push_back('0');
        }

        // Append the last '1' to make the binary number odd
        answer.push_back('1');

        // Return the constructed maximum odd binary number
        return answer;
    }
};
```

TypeScript

```typescript
function maximumOddBinaryNumber(binaryString: string): string {
    // Initialize count to keep track of the number of '1's in the binary string.
    let onesCount = 0;

    // Iterate through each character of the binary string input.
    for (const character of binaryString) {
        // Increment the count for every '1' found.
        onesCount += character === '1' ? 1 : 0;
    }

    // Generate the maximum odd binary number by repeating '1' for (onesCount - 1) times,
    // followed by repeating '0' for (string length - onesCount) times, and then appending '1' at the end of the string.
    // This maximizes the number of '1' digits at the front, while ensuring the last digit is '1' for odd number.
    return '1'.repeat(onesCount - 1) + '0'.repeat(binaryString.length - onesCount) + '1';
}
```

```python
class Solution:
    def maximumOddBinaryNumber(self, s: str) -> str:
        # Count the number of '1's in the binary string
        count_ones = s.count("1")

        # If there are no '1's, there's no odd number possible, return empty string
        if count_ones == 0:
            return ""

        # Otherwise, construct the maximum odd binary number by doing the following:
        # - Use all '1's except one to maintain oddness ('1' * (count_ones - 1))
        # - Fill the rest of the string with '0's (len(s) - count_ones) * "0")
        # - Add a '1' at the end to ensure the number is odd
        return "1" * (count_ones - 1) + (len(s) - count_ones) * "0" + "1"
```

Time and Space Complexity

Time Complexity

The time complexity of the provided function is determined by three operations:

1. The `s.count("1")` operation goes through the entire string `s` to count the number of `"1"` characters. This operation has a time complexity of `O(n)`, where `n` is the length of the input string `s`.
2. The construction of the string `"1" * (cnt - 1)` which repeats the character `"1"` (cnt - 1) times has a time complexity of `O(cnt)` because it needs to create a new string with the '1's.
3. Similarly, `(len(s) - cnt) * "0"` creates a string of zeros with length equal to the `len(s) - cnt`, giving a time complexity of `O(n - cnt)`.
4. The final concatenation of the two strings and the additional `"1"` character has a time complexity of `O(n)` since string concatenation is typically linear in the length of the strings being concatenated.

Since the string `s` must be traversed entirely in the worst case, and the subsequent operations also depend linearly on the length of `s`, the overall time complexity can be approximated to `O(n)` where 'n' represents the length of the string `s`.

Space Complexity

The space complexity comes from the storage needed for the resulting string. Since the function constructs a new string whose maximum length can be `n` characters long (which is the length of the original string `s`), the space complexity is `O(n)` as well.