1328. Break a Palindrome

Medium Greedy String

Problem Description

The task is to take a palindromic string, which reads the same forwards and backwards, and alter exactly one character to create a new string that is no longer a palindrome. Moreover, the resulting string must be the lexicographically smallest (alphabetically earliest) string possible. Provided that the palindromic string consists of lowercase English letters, if it's impossible to execute such an operation because the string cannot be changed in a way that would not result in a palindrome (for example, if the string is a single character), the function must return an empty string.

Intuition

To solve this problem, we need to understand two key concepts: what makes a string a palindrome and what it means for a string to be lexicographically smaller than another. A palindrome is a string that reads the same forwards as it does backwards. Changing any character in the first half of the string would naturally require a change in the corresponding character in the second half to maintain the property of being a palindrome. To make the resulting string lexicographically smallest, we should aim to replace the first non-'a' character in the string (since 'a' is the smallest letter in the alphabet) with an 'a' if such a character exists in the first half of the string. If the string consists entirely of 'a', any replacement in the first half will still result in a palindrome; therefore, we change the very last character to 'b' (which follows 'a' in the alphabet). The intuition behind changing the last character to 'b' when the string is all 'a's is that we are guaranteed to break the palindrome property while ensuring that the change is minimal and still lexicographically smallest. This strategy leads us to the solution approach where we inspect each character in the first half of the string until we find a non-'a' character to substitute with 'a'. If no such character is found, we overwrite the last character with 'b'.

Solution Approach

The solution employs a straightforward iterative approach, with some early returns for edge cases. Specifically, the solution makes use of the following algorithmic steps and simple data structures:

1. Convert the input palindromic string, `palindrome`, into a list of characters, stored in `s`, to allow mutable operations since strings in Python are immutable.
2. Determine the length of the input string, `n`, for iterating through its characters.
3. If `n` equals 1, immediately return an empty string as it's not possible to create a non-palindromic string by changing just one character. This serves as an early exit condition.
4. Commence a loop to iterate through the first half of the character list `s`, as changing any character in the second half would have a mirrored counterpart in the first half, thus maintaining the palindrome property.
5. Seek the first occurrence of a character different from `'a'`. If such a character is found at position `i` before reaching the midpoint (`n // 2`), replace it with `'a'` to make the string lexicographically smaller while breaking the palindrome.
6. If the loop finishes without finding any non-'a' characters (meaning the string was composed wholly of 'a's), change the last character in `s` to `'b'`. This ensures that the palindrome is broken, and because we are only altering the very end of the string, the outcome remains the lexicographically smallest.
7. Join the characters in the list `s` back into a string and return it.

This implementation doesn't require complex data structures or patterns; it relies on basic string manipulation and the understanding that a minimal change at the earliest point in a string will lead to the lexicographically smallest outcome.

Example Walkthrough

Let's take the palindromic string `abba` as a small example to illustrate the solution approach:

1. Convert `abba` into a list of characters, `s`, so it becomes `['a', 'b', 'b', 'a']`.
2. The length of `abba` is `n = 4`.
3. Since `n` is greater than 1, we don't return an empty string but proceed with the solution.
4. We loop through the first half of `s`, which means we only look at the first two characters, `['a', 'b']`.
5. As we start iterating, we find that the first character is an `'a'` and move on to the second character, which is `'b'`.
6. Since `'b'` is different from `'a'`, we have found our target character before reaching the midpoint (`n // 2`) of `s`, which in this case is at index 1.
7. We replace the character at index 1 with an `'a'` and now `s` becomes `['a', 'a', 'b', 'a']`.
8. We join `s` back into a string, resulting in the non-palindromic string `aaba`, which is lexicographically smaller than `abba`.

In this case, `aaba` is returned as the solution since it's the lexicographically smallest string that can be obtained by changing exactly one character in `abba` to ensure it's no longer a palindrome.

Solution Implementation

Python

```python
class Solution:
    def breakPalindrome(self, palindrome: str) -> str:
        # Calculate the length of the palindrome string
        length = len(palindrome)

        # If the length is 1, it is the smallest palindrome and cannot be made non-palindromic
        if length == 1:
            return ""

        # Convert the palindrome string into a list of characters for easy manipulation
        characters = list(palindrome)

        # Initialize an index to iterate over the characters
        index = 0

        # Iterate over the first half of the palindrome
        # to find a character that is not 'a'
        while index < length // 2 and characters[index] == "a":
            index += 1

        # If all characters in the first half are 'a',
        # change the last character to 'b' to make it non-palindromic
        if index == length // 2:
            characters[-1] = "b"
        else:
            # Change the first non-'a' character found in the first half to 'a'
            characters[index] = "a"

        # Join the characters list to form a new string and return
        return "".join(characters)

# Example usage:
# solution = Solution()
# result = solution.breakPalindrome("abccba") # Returns "aaccba"
```

Java

```java
class Solution {
    public String breakPalindrome(String palindrome) {
        // Get the length of the palindrome string.
        int length = palindrome.length();

        // If the string is only one character, we cannot make it non-palindrome by changing any letter.
        if (length == 1) {
            return "";
        }

        // Convert the string into a character array for manipulation.
        char[] charArray = palindrome.toCharArray();

        // Initialize an index variable to iterate over the first half of the string.
        int index = 0;

        // Iterate over the first half of the palindrome.
        // We only need to check the first half since the second half is a mirror of the first.
        while (index < length / 2 && charArray[index] == 'a') {
            // Move to the next character if the current character is 'a'.
            ++index;
        }

        // If we've reached the middle without finding a character that is not 'a',
        // then all characters must be 'a'. In this case, change the last character to 'b'.
        if (index == length / 2) {
            charArray[length - 1] = 'b';
        } else {
            // Otherwise, we found a non-'a' character in the first half,
            // change it to 'a' to break the palindrome.
            charArray[index] = 'a';
        }

        // Convert the character array back to a string and return the result.
        return new String(charArray);
    }
}
```

C++

```cpp
class Solution {
public:
    // Break a palindrome by changing the minimum lexicographical character.
    // The input is guaranteed to be a non-empty palindrome.
    string breakPalindrome(string palindrome) {
        int length = palindrome.size();

        // If the palindrome has only one character, it's impossible to make it non-palindromic
        // by changing any single letter, hence return an empty string.
        if (length == 1) {
            return "";
        }

        // Iterate through the first half of the string
        for (int i = 0; i < length / 2; ++i) {
            // If the character is not 'a', change it to 'a' to make the string lexicographically smaller
            // and ensure it's no longer a palindrome.
            if (palindrome[i] != 'a') {
                palindrome[i] = 'a';
                return palindrome; // The string is no longer a palindrome, return it
            }
        }

        // If the loop completes, it means all the characters in the first half are 'a'.
        // Change the last character to 'b' to make the palindrome lexicographically smallest.
        palindrome[length - 1] = 'b';
        return palindrome; // Return the modified palindrome.
    }
};
```

TypeScript

```typescript
function breakPalindrome(palindrome: string): string {
    // Get the length of the palindrome
    const length = palindrome.length;

    // If the string is a single character, it can't be broken into a non-palindrome,
    // returning an empty string as per the problem statement.
    if (length === 1) {
        return '';
    }

    // Convert the string into an array of characters for easy manipulation.
    const characters = palindrome.split('');

    // Initialize an index to iterate through the characters of the palindrome.
    let index = 0;

    // Iterate through the first half of the palindrome searching for a non-'a' character.
    // If we find an 'a', we increase the index and keep looking
    // As it's a palindrome, we only need to iterate through the first half of the string.
    while (index < length >> 1) && characters[index] === 'a') {
        index++;
    }

    // If the whole first half consists of 'a's, it means we have to change the last character
    // to 'b' because we are looking for the lexicographically smallest palindrome bigger than
    // the original. All preceding 'a's are the smallest possible characters.
    if (index == (length >> 1)) {
        characters[length - 1] = 'b';  // Change the last character to 'b'.
    } else {
        characters[index] = 'a';  // Replace the first non-'a' character with 'a'.
    }

    // Join the array of characters back into a string and return the result.
    return characters.join('');
}


class Solution:
    def breakPalindrome(self, palindrome: str) -> str:
        # Calculate the length of the palindrome string
        length = len(palindrome)

        # If the length is 1, it is the smallest palindrome and cannot be made non-palindromic
        if length == 1:
            return ""

        # Convert the palindrome string into a list of characters for easy manipulation
        characters = list(palindrome)

        # Initialize an index to iterate over the characters
        index = 0

        # Iterate over the first half of the palindrome
        # to find a character that is not 'a'
        while index < length // 2 and characters[index] == "a":
            index += 1

        # If all characters in the first half are 'a',
        # change the last character to 'b' to make it non-palindromic
        if index == length // 2:
            characters[-1] = "b"
        else:
            # Change the first non-'a' character found in the first half to 'a'
            characters[index] = "a"

        # Join the characters list to form a new string and return
        return "".join(characters)

# Example usage:
# solution = Solution()
# result = solution.breakPalindrome("abccba") # Returns "aaccba"
```

Time and Space Complexity

Time Complexity

The provided code has a time complexity of `O(n)`, where `n` is the length of the input string `palindrome`. The reason for this time complexity is because the code iterates over at most half of the string (up to `n/2`) in the worst-case scenario, in a linear fashion, to find the first non-'a' character. Changing a character and joining the list into a string both require linear time proportional to the length of the string.

Space Complexity

The space complexity is `O(n)` as well, because a new list `s` of characters is created from the input string, which requires additional space proportional to the size of the input string. The space used by the indices and the temporary storage when changing characters is constant and does not depend on the size of the input, so it does not contribute significantly to the space complexity.