2697. Lexicographically Smallest Palindrome

Easy Two Pointers String

Leetcode Link

Problem Description

The problem presents a scenario in which you have a string `s` comprised entirely of lowercase English letters. The goal is to transform this string into a palindrome using a series of operations with two conditions:

1. Minimize the number of operations.
2. If multiple palindromes can be achieved with the same minimum number of operations, choose the one that is lexicographically smallest.

An operation is defined as replacing any single character in the string with another lowercase English letter. And finally, the concept of a lexicographically smaller string is explained: it's a string that has an earlier occurring letter at the first position where two strings differ.

Your task is to write a function that returns the resulting palindrome that satisfies the above conditions.

Intuition

Transforming a string into a palindrome means making the string the same when read forwards and backwards. A straightforward approach to doing this is to iterate over the string from both ends towards the center, comparing characters at symmetric positions.

If the characters at symmetric positions are different, you must choose which one to change to match the other. To satisfy the condition of lexicographical order, we should choose the smaller (earlier in the alphabet) of the two characters and set both positions to that character.

We continue this process until we reach the middle of the string. Since we operate on symmetric positions, the number of operations is inherently minimized, as each operation ensures that both the left and right sides match.

Knowing this, the implemented solution approach is as follows:

1. We convert the string into a list of characters to allow in-place modification because strings in Python are immutable.
2. We set two pointers, `i` and `j`, at the start and end of the list, respectively.
3. We iterate, moving `i` forward and `j` backward, and at each step, we compare the characters at these indices.
4. If they are not the same, we set both `cs[i]` and `cs[j]` to the lexicographically smaller of the two characters.
5. We increment `i` and decrement `j` and repeat this process until `i` is no longer less than `j`.
6. We join the list back into a string and return it, which is the lexicographically smallest palindrome obtainable with the minimal number of operations.

Solution Approach

The solution makes use of a two-pointer approach and array manipulation. Here is a step-by-step breakdown of the algorithm:

1. **Convert the Immutable String to a Mutable List**: Python strings are immutable, meaning they cannot be changed after they are created. To overcome this, the string `s` is converted into a list of characters `cs` using `list(s)`. Lists allow in-place modification, which is necessary for our operations.

2. **Initialize Two Pointers**: Two pointers `i` and `j` are initialized. Pointer `i` starts at the beginning of the list (index 0) and `j` starts at the end of the list (index `len(s) - 1`). These pointers will traverse the list from both ends towards the center.

3. **Iterate and Compare Characters**: While `i` is less than `j`, indicating that we haven't reached the middle of the list, we compare the characters at `cs[i]` and `cs[j]`.

4. **Perform the Replacement**: If the characters at these two positions are different, we find the lexicographically smaller character by using the `min(s[i], s[j])` function and set both `cs[i]` and `cs[j]` to that character.

5. **Move the Pointers**: After handling the characters at the current positions, we move `i` forward by 1 (`i + 1`) and `j` backward by 1 (`j - 1`) to compare the next set of characters.

6. **Exit the Loop**: The loop continues until `i` is not less than `j`. At this point, we have ensured that for every position `i` from the start to the middle of the list, there is a corresponding position `j` from the end to the middle where `cs[i]` and `cs[j]` are identical—guaranteeing palindrome property.

7. **Return the Result**: After exiting the loop, we have a list of characters representing our smallest lexicographically possible palindrome. We join this list back into a string with `"".join(cs)` and return this final palindrome string.

No additional complex data structures beyond the basic list and standard control structures (a while loop with two pointers) are used. The underlying pattern used is the two-pointer technique, which is a common strategy to compare and move towards the center from both ends of a sequence.

Example Walkthrough

Let's consider the following example to illustrate the solution approach:

Suppose we have the string `s = "abzca"`. We want to transform this string into a palindrome using the least number of operations such that if there are multiple possible palindromes, we choose the lexicographically smallest one.

Following the solution steps:

1. **Convert the Immutable String to a Mutable List**: We convert `s` into a list `cs = ['a', 'b', 'z', 'c', 'a']`.

2. **Initialize Two Pointers**: We set `i = 0` at the beginning of the list and `j = len(cs) - 1 = 4` at the end of the list.

3. **Iterate and Compare Characters**:
   - First iteration: `cs[i] = 'a'` and `cs[j] = 'a'`, since they are the same, we don't need to perform any operations. Move `i` to `1` and `j` to `3`.
   - Second iteration: Now, `cs[i] = 'b'` and `cs[j] = 'c'`. They are different, so we have to replace one of them. We choose the lexicographically smaller character, which is 'b', and set both positions to 'b': `cs = ['a', 'b', 'z', 'b', 'a']`. Move `i` to `2` and `j` to `2`.
   - Since `i` and `j` are the same value, we have reached the middle of the list and no further iterations are needed.
4. **Return the Result**: The resulting list represents the palindrome `'abbba'`. We convert it back to a string and, therefore, the function would return `"abbba"`.

Here, only one operation was needed to achieve the palindrome, and `'abbba'` is the lexicographically smallest palindrome that can be made from `abzca`.

Python Solution

```python
class Solution:
    def makeSmallestPalindrome(self, s: str) -> str:
        # Initialize pointers for the start and end of the string.
        left_pointer, right_pointer = 0, len(s) - 1
        # Convert the string to a list of characters for easier manipulation.
        char_list = list(s)

        # Loop until the pointers meet in the middle or cross each other.
        while left_pointer < right_pointer:
            # If the characters at the current pointers don't match,
            # replace them with the lexicographically smaller one to help
            # create the smallest possible palindrome.
            if s[left_pointer] != s[right_pointer]:
                smaller_char = min(s[left_pointer], s[right_pointer])
                char_list[left_pointer] = smaller_char
                char_list[right_pointer] = smaller_char

            # Move the pointers closer to the center.
            left_pointer += 1
            right_pointer -= 1

        # Join the character list to form the resulting string and return it.
        return "".join(char_list)

```

Java Solution

```java
class Solution {
    public String makeSmallestPalindrome(String s) {
        // Convert the input string to a character array for easy manipulation.
        char[] characters = s.toCharArray();

        // Use two pointers, starting from the beginning and end of the character array.
        for (int left = 0, right = characters.length - 1; left < right; ++left, --right) {
            // If the characters at the two pointers don't match, replace both with the smaller one.
            if (characters[left] != characters[right]) {
                characters[left] = characters[right] =
                    (characters[left] < characters[right]) ? characters[left] : characters[right];
            }
        }

        // Convert the character array back to a string and return.
        return String.valueOf(characters);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to create the lexicographically smallest palindrome from a given string
    string makeSmallestPalindrome(string s) {
        // Use two pointers, starting from the beginning and end of the string
        for (int left = 0, right = s.size() - 1; left < right; ++left, --right) {
            // If the characters at the current left and right positions are different
            if (s[left] != s[right]) {
                // Update both characters to the lexicographically smaller one
                s[left] = s[right] = std::min(s[left], s[right]);
            }
        }
        // Return the modified string which is the smallest palindrome
        return s;
    }
};

```

Typescript Solution

```typescript
function makeSmallestPalindrome(inputString: string): string {
    // Convert the input string into an array of characters.
    const charArray = inputString.split('');

    // Iterate over the string from both ends towards the center.
    for (let leftIndex = 0, rightIndex = inputString.length - 1; leftIndex < rightIndex; ++leftIndex, --rightIndex) {

        // Check if characters at current left and right indexes are different.
        if (inputString[leftIndex] !== inputString[rightIndex]) {
            // Choose the lexicographically smaller character among the two
            // and replace both characters with it to make the string a palindrome.
            charArray[leftIndex] = charArray[rightIndex] =
                inputString[leftIndex] < inputString[rightIndex] ?
                inputString[leftIndex] : inputString[rightIndex];
        }
    }

    // Return the modified array as a string.
    return charArray.join('');
}

```

Time and Space Complexity

Time Complexity

The given Python function, `makeSmallestPalindrome`, primarily consists of a while loop that iterates over the string elements. The `i` index starts from the beginning of the string, and the `j` index starts from the end, moving towards the center of the string. Since each iteration moves both pointers (i.e., `i` and `j`), the loop runs for approximately `n/2` iterations, where `n` is the length of the input string `s`. The operations inside the loop (like comparing characters, assigning characters, and incrementing or decrementing pointers) all have a constant time complexity, `O(1)`.

Therefore, the time complexity of the function is `O(n/2)`, which simplifies to `O(n)` since we drop constants when expressing big O notation.

Space Complexity

The space complexity is determined by the additional space used by the function beyond the input itself. Here, a new list `cs` of characters is created from the input string `s`, which occupies `O(n)` space, where `n` is the length of the input string. The rest of the variables (`i`, `j`) use constant space, `O(1)`.

Hence, the overall space complexity of the function is `O(n)+O(1)` which simplifies to `O(n)`, as the linear term dominates over the constant one.