409. Longest Palindrome

Easy Accepted Hash Table String

Problem Description

In this problem, we're given a string `s` containing a mix of lowercase and uppercase letters. Our task is to determine the maximum length of a palindrome that can be created using the letters from the string. It's important to note that case matters here; 'A' and 'a' are treated as different characters, which means a string like "Aa" wouldn't count as a palindrome.

Intuition

To solve this problem, we can use the fact that a palindrome is symmetrical around its center. This symmetry means that most letters have to appear an even number of times in the string, so they can be mirrored on both sides of the palindrome. The only exception is the center of the palindrome, which can hold a single character if the length of the palindrome is odd.

With this in mind, we can iterate over the counts of each letter in the string. For each letter:

1. If it has an even count, we can use all occurrences of that letter in the palindrome.
2. If it has an odd count, we can use all but one occurrence of that letter to maintain symmetry.

Additionally, we can place exactly one character with an odd count in the center of the palindrome (if we haven't already placed a center character). To handle this gracefully, we can use a greedy approach: always add characters in pairs to the palindrome, and if there is no center yet and we encounter an odd count, we place one of those characters at the center.

The implementation uses a `Counter` to count occurrences of each character in `s`. We then iterate over these counts, and for each:

- We add to the answer the largest even number that is less than or equal to the count of the character. This is achieved by `v - (v & 1)` which subtracts one if `v` is odd.
- We potentially add one more to the answer (for the center character) if there isn't already a center character. This is determined by `ans & 1 ^ 1` which is true if `ans` is even, signifying that we haven't added a center character yet, and `v & 1` which is true if `v` is odd.

Solution Approach

The implementation starts by counting the frequency of each character in the given string `s`. This is done using the `Counter` class from Python's `collections` module. The `Counter` class creates a dictionary-like object where the keys are the characters from `s`, and the values are the counts of those characters.

```
cnt = Counter(s)
```

We then initialize the `ans` variable to 0, which will serve as our answer to hold the length of the longest palindrome we can build.

```
ans = 0
```

Next, we iterate over the values in `cnt` (which are the counts of each character in the string) and for each value `v`:

1. We want to add as many characters as possible to the palindrome while maintaining its symmetrical structure. To accomplish this, we add the largest even number smaller than or equal to `v` to our answer `ans`. This is done by using the expression `v - (v & 1)`, which subtracts 1 from `v` if `v` is odd, effectively giving us the largest even number.

```
ans += v - (v & 1)
```

2. We need to consider the possibility of a central character in the palindrome. We can afford to add one such character if we haven't already added one. The expression `(ans & 1 ^ 1)` checks if `ans` is still even. If it is, it means we haven't placed a center character in the palindrome yet, and `(v & 1)` checks if `v` is odd, indicating that this character could potentially be the center. If both conditions are true, we add 1 to `ans`, thereby adding a central character.

```
ans += (ans & 1 ^ 1) and (v & 1)
```

Finally, we return the `ans`, which gives us the length of the longest palindrome that can be created with the characters from `s`.

The overall algorithm is a greedy one, as it tries to use as many characters as possible while respecting the condition that only one character can be in the middle of a palindrome if the length is odd. By using a Counter and iterating through its values, we efficiently consider each unique character without having to check the entire string multiple times.

```
class Solution:
    def longestPalindrome(self, s: str) -> int:
        cnt = Counter(s)
        ans = 0
        for v in cnt.values():
            ans += v - (v & 1)
            ans += (ans & 1 ^ 1) and (v & 1)
        return ans
```

Example Walkthrough

Let's walk through the solution approach with a small example. Suppose our input string `s` is `"AabbcC"`.

1. We first count the frequency of each character using the `Counter` class.
   - The `Counter(s)` would give us `{'A': 1, 'a': 1, 'b': 2, 'c': 1, 'C': 1}`.
2. Now we initialize our answer `ans` to 0.
3. We start iterating over the character counts:
   - For `'A'` with a count of 1: `ans += 1 - (1 & 1)` which adds 0 as 'A' has an odd count, and we can't have a pair yet. Then, `ans += (ans & 1 ^ 1) and (1 & 1)` will add 1, because `ans` is even, and 'A' could potentially be at the center.
   - For `'a'` with a count of 1: We again add 0 for pairs of 'a'. Since we already have a center character, we don't add another.
   - For `'b'` with a count of 2: `ans += 2 - (2 & 1)` adds 2 as 'b' has an even count, we can use both.
   - For `'c'` with a count of 1: Analogously to 'A', we add 0 for pairs and do not add to the center as we already have one.
   - For `'C'` with a count of 1: Same as with 'c', we add 0 for pairs and nothing to the center.
4. The final `ans` is 3, representing the longest palindrome `"AbA"`, which we can build from the input string `s`.

So, from input string `"AabbcC"`, the maximum length palindrome we can create is 3, and the palindrome could be `"AbA"`.

Solution Implementation

Python

```
from collections import Counter

class Solution:
    def longestPalindrome(self, s: str) -> int:
        # Count the occurrences of each character in the input string
        char_count = Counter(s)
        # Initialize the length of the longest palindrome to be 0
        longest_palindrome_length = 0

        # Iterate through the counts of each character
        for count in char_count.values():
            # Add the largest even number less than or equal to 'count' to the
            # length of the longest palindrome. This represents the maximum
            # number of characters that can be used in both halves of the palindrome.
            longest_palindrome_length += count - (count % 2)

            # Ensure that there is a center character if one has not been chosen already:
            # Check if longest palindrome length is currently even and if 'count' is odd.
            # If conditions satisfy, increase the longest palindrome length by 1
            # to include one of the odd-count characters as the center of the palindrome.
            longest_palindrome_length += ((longest_palindrome_length % 2 == 0) and (count % 2 == 1))

        # Return the length of the longest palindrome that can be built with the characters
        return longest_palindrome_length
```

Java

```
class Solution {
    public int longestPalindrome(String s) {
        // Create an array to count occurrences of each character.
        // The ASCII value of a character will be used as the index.
        int[] charCounts = new int[128];
        for (int i = 0; i < s.length(); i++) {
            // Count each character's occurrences.
            charCounts[s.charAt(i)]++;
        }

        int lengthOfLongestPalindrome = 0;
        for (int count : charCounts) {
            // Add the largest even number below or equal to the current character count.
            // This is equivalent to count - (count % 2).
            lengthOfLongestPalindrome += count - (count & 1);

            // If the current palindrome length is even and the count is odd,
            // we can add one more character to the center of the palindrome.
            if (lengthOfLongestPalindrome % 2 == 0 && count % 2 == 1) {
                lengthOfLongestPalindrome++;
            }
        }
        // Return the length of the longest possible palindrome.
        return lengthOfLongestPalindrome;
    }
}
```

C++

```
#include <string> // Include the string header for using the std::string type

class Solution {
public:
    int longestPalindrome(std::string s) {
        // Create an array to hold the count of each character.
        // 128 covers all ASCII characters which include standard English letters, digits, and punctuation.
        int charCount[128] = {};

        // Count the occurrence of each character in the string.
        for (char c : s) {
            charCount[c]++;
        }

        int maxLength = 0; // Initialize the length of the longest palindrome to 0.

        // Iterate through the character counts.
        for (int count : charCount) {
            // Add the largest even number that is less than or equal to the current count.
            // This is because even counts can be mirrored on both sides of a palindrome.
            maxLength += count - (count % 2);

            // If the current maxLength is even and there's an odd count of characters,
            // we can add one of those characters to the center of the palindrome.
            // Only one center character is allowed for a palindrome, hence the check (maxLength % 2 == 0).
            if (maxLength % 2 == 0 && count % 2 == 1) {
                maxLength++;
            }
        }

        // Return the length of the longest palindrome that can be created.
        return maxLength;
    }
};
```

TypeScript

```
function longestPalindrome(s: string): number {
    let lengthOfString = s.length;
    let lengthOfLongestPalindrome = 0;
    let charCount = new Array(128).fill(0); // array to store character frequencies

    // Count the frequency of each character in the string.
    for (let i = 0; i < lengthOfString; i++) {
        charCount[s.charCodeAt(i)]++;
    }

    // Iterate over the character frequency array.
    for (let i = 0; i < 128; i++) {
        let frequency = charCount[i];
        // If the frequency is even, add it to the lengthOfLongestPalindrome, since
        // even counts of a character can be placed symmetrically in the palindrome.
        // If the frequency is odd, the maximum even count that can be used is frequency - 1.
        lengthOfLongestPalindrome += frequency % 2 == 0 ? frequency : frequency - 1;
    }

    // If the length of the constructed palindrome is less than the length of the original string,
    // we can add one more character (center of the palindrome).
    // This character does not need to have a matching pair.
    return lengthOfLongestPalindrome < lengthOfString ? lengthOfLongestPalindrome + 1 : lengthOfLongestPalindrome;
}
```

```
from collections import Counter

class Solution:
    def longestPalindrome(self, s: str) -> int:
        # Count the occurrences of each character in the input string
        char_count = Counter(s)
        # Initialize the length of the longest palindrome to be 0
        longest_palindrome_length = 0

        # Iterate through the counts of each character
        for count in char_count.values():
            # Add the largest even number less than or equal to 'count' to the
            # length of the longest palindrome. This represents the maximum
            # number of characters that can be used in both halves of the palindrome.
            longest_palindrome_length += count - (count % 2)

            # Ensure that there is a center character if one has not been chosen already:
            # Check if longest palindrome length is currently even and if 'count' is odd.
            # If conditions satisfy, increase the longest palindrome length by 1
            # to include one of the odd-count characters as the center of the palindrome.
            longest_palindrome_length += ((longest_palindrome_length % 2 == 0) and (count % 2 == 1))

        # Return the length of the longest palindrome that can be built with the characters
        return longest_palindrome_length
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily determined by the traversal through the characters of the string `s` and the values in the `cnt` (counter) object.

1. The first operation is creating a frequency counter (`cnt`) for the characters in the string with `Counter(s)` which takes `O(n)` time where `n` is the length of the string `s`.
2. The second part involves iterating through the values of the `cnt` object. The number of unique characters in `s` will be at most `k`, where `k` is the size of the character set (such as 26 for lowercase English letters, etc.), thus this iteration is `O(k)`.

Combining these steps, since `k` can be at most `n` when all characters are unique, the overall time complexity is `O(n)`.

Space Complexity

The space complexity of the code includes the space required for storing the frequency of each character in the string `s`.

1. The `Counter(s)` creates a dictionary with at most `k` key-value pairs where `k` is the number of unique characters in `s`. Hence, the space required for this is `O(k)`.

Since `k` is the number of unique characters and `k` can be at most `n`, the space complexity is also `O(n)` if we assume that the input string can have a large and varied set of characters.

Overall, the code has a time complexity of `O(n)` and a space complexity of `O(n)`.