1400. Construct K Palindrome Strings

Medium Greedy Hash Table String Counting

Leetcode Link

Problem Description

The LeetCode problem asks us to determine if we can construct `k` palindromic strings using all the characters of a given string `s`. A palindrome is a word or phrase that reads the same forwards and backwards. The constraint is that we must use every character in `s` exactly once to create these `k` palindromes. If we can achieve this, we return `true`. Otherwise, we return `false`.

To put it simply, we want to know if the characters in `s` can be rearranged in a way that they form `k` separate palindromes. It's important to remember that the lengths of the palindromes can vary, and they do not need to be all of the same length.

Intuition

To solve this problem, we need to understand the characteristics of palindromes:

1. Even-count characters can be used entirely to form palindromes as they can be symmetrically placed around the center. For example, 'aabb' can be used to form the palindrome 'abba'.
2. Odd-count characters have a different constraint. At most, one character with an odd count can be in the center of a palindrome (for example, 'racecar' where 'e' is in the middle), and all others must be paired to maintain symmetry.
3. Given these properties, if `k` is greater than the length of `s`, we cannot form `k` non-empty palindromes.

The intuition behind the solution focuses on points 1 and 2:

- First, we need to know if there are enough characters in `s` to form `k` palindromes. We achieve this by comparing the length of `s` to `k`. If `s` is shorter than `k`, we cannot form `k` palindromes, so we return `false`.
- Next, we count how many characters have an odd count because each odd-count character can be the center of a palindrome only once. This means the number of palindromes we can create is limited by the number of characters with odd counts, as every palindrome can only have one center character.
- We then check if the number of odd-count characters is less than or equal to `k` because if there are more odd-count characters than `k`, we can't place them at the center of each palindrome, and thus can't create `k` valid palindromes.
- If the number of odd-count characters is less than or equal to `k`, we can always rearrange the characters to create the required palindromes, so we return `true`.

The solution makes use of Python's `Counter` class from the `collections` module to count the instances of each character in `s`. The expression `v & 1` is a bitwise operation checking if the count `v` is odd (since odd numbers have the lowest bit set to 1). The summation counts how many characters have odd occurrences, and finally, this count is compared to `k`.

Solution Approach

The solution approach follows a well-defined set of steps that leverage a few Python-specific tools as well as some general algorithmic concepts.

Here is a breakdown of the implementation:

1. **Check Length Against `k`:** We start by comparing the length of the string `s` to the integer `k`. If `len(s)` is less than `k`, we immediately return `false` because you cannot construct more non-empty palindromic strings than the number of available characters.

```
if len(s) < k:
    return False
```

2. **Count Characters with `Counter`:** Next, we use the `Counter` class from the `collections` module of Python. This is a subclass of the dictionary which is designed to count hashable objects (like characters in a string). It helps in finding the frequency of each character in `s`.

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

3. **Count Odd-Occurrence Characters:** We then proceed to calculate the number of characters that appear an odd number of times in the string. To find out if the count `v` is odd, we use the bitwise AND (`&`) operation with `1`. If `v & 1` yields `1`, the count is odd. This trick works because in binary representation the least significant bit determines the oddness of a number. Summing these up gives us the total number of odd-count characters.

```
sum(v & 1 for v in cnt.values())
```

The comprehension iterates over the counts of each character obtained from `Counter`. Each value is checked for oddness, and then they are summed together to get a total count of characters that have to be placed in the center of a palindrome.

4. **Compare Odd Count to `k`:** Lastly, the implementation compares the number of odd-count characters to `k`. As mentioned before in the intuition, if the sum of odd-count characters is less than or equal to `k`, it is possible to form `k` palindromic strings using all the characters of `s`. Otherwise, it's not feasible.

```
return sum(v & 1 for v in cnt.values()) <= k
```

By understanding that each palindrome can only have one odd-count character at its center, the solution efficiently validates the possibility of creating the `k` palindromic strings. There is no need to actually construct the palindromes, only to ensure the conditions for their existence are met. This approach leads to a time complexity of O(n), where n is the length of the string `s`, which is the time required to count the frequency of each character.

Example Walkthrough

Let's consider an example with string `s = "aaabbbcc"` and `k = 2`.

Step 1: Check Length Against `k`

First, we compare the length of `s` with `k`. Since `len("aaabbbcc")` is 8, which is greater than `k` (which is 2), we can proceed. If the string `s` was shorter than `k`, we wouldn't have enough characters to form `k` non-empty palindromic strings, and the function would return `false` immediately.

Step 2: Count Characters with `Counter`

We count the frequency of each character in the string `s`:

- 'a' appears 3 times
- 'b' appears 3 times
- 'c' appears 2 times

Step 3: Count Odd-Occurrence Characters

From our counts, we can see that 'a' and 'b' have odd occurrences (3 times each), while 'c' is even. There are two characters ('a' and 'b') that have an odd count.

Step 4: Compare Odd Count to `k`

Since we have 2 odd-count characters ('a' and 'b'), and `k` is also 2, this fulfills the requirement that we can have at most one odd-count character at the center of a palindrome. Therefore, we can potentially form 2 palindromes, one with 'a' at the center and one with 'b' at the center, using 'c' to complete both palindromes since it has an even count.

Here, `sum(v & 1 for v in cnt.values())` equals 2, which is `<= k`. Thus, the final comparison is `true`.

Based on these steps, we could rearrange 'aaabbbcc' into two palindromic strings such as 'abcba' and 'acbca', so the function should return `true`.

This example illustrates the solution approach step-by-step and confirms the possibility of forming `k` palindromic strings using all characters of `s`.

Python Solution

```
from collections import Counter

class Solution:
    def canConstruct(self, string: str, num_partitions: int) -> bool:
        # If the length of the string is less than the required number of partitions,
        # we cannot construct the required partitions, so return False.
        if len(string) < num_partitions:
            return False

        # Count the frequency of each character in the string using Counter.
        char_counter = Counter(string)

        # Calculate the number of characters that have an odd count.
        # This loop goes through the values (counts) in the char_counter
        # and uses a bitwise AND operation (& 1 with i to determine if the count is odd.
        odd_count = sum(count & 1 for count in char_counter.values())

        # The number of characters with odd counts should not exceed the number
        # of partitions we want to create, because each palindrome must have at most
        # one character with an odd count (in the middle). Therefore, if we have
        # no more odd counts than the number of partitions, we can construct the palindromes.
        return odd_count <= num_partitions

```

Java Solution

```
class Solution {
    public boolean canConstruct(String inputString, int palindromeCount) {
        // Length of the input string
        int length = inputString.length();

        // If the input string is shorter than the required number of palindromes,
        // it is not possible to construct the palindromes.
        if (length < palindromeCount) {
            return false;
        }

        // Array to hold the count of each character in the inputString.
        int[] characterFrequency = new int[26];

        // Count the frequency of each character in the inputString.
        for (int i = 0; i < length; ++i) {
            characterFrequency[inputString.charAt(i) - 'a']++;
        }

        // Count the number of characters that appear an odd number of times.
        int oddCount = 0;
        for (int frequency : characterFrequency) {
            oddCount += frequency % 2;
        }

        // It is possible to form palindromes if the number of characters with
        // odd frequency is less than or equal to the number of palindromes we need to construct.
        return oddCount <= palindromeCount;
    }
}

```

C++ Solution

```
class Solution {
public:
    // Method to determine if a string s can be rearranged to form exactly k palindromic substrings.
    bool canConstruct(string s, int k) {
        // If the size of s is less than k, it's not possible to construct k palindromes
        if (s.size() < k) {
            return false;
        }

        // Array to store the frequency of each letter in the string
        int letterCount[26] = {0};
        // Count the frequency of each character in s
        for (char& c : s) {
            ++letterCount[c - 'a'];
        }

        // Variable to keep track of the number of characters with odd frequency
        // (which determines how many palindromes can be made)
        int oddCount = 0;
        // Check the letter frequencies
        for (int count : letterCount) {
            // If the count is odd, increment oddCount
            oddCount += count & 1;
        }

        // A palindrome can accommodate one odd count character (the middle one), so if there
        // are more odd count characters than k, it's not possible to create k palindromes.
        // Hence, we return whether the oddCount is less than or equal to k.
        return oddCount <= k;
    }
};

```

Typescript Solution

```
function canConstruct(s: string, k: number): boolean {
    // Early return if the string's length is less than k
    if (s.length < k) {
        return false;
    }

    // Initialize an array to hold the frequency of each character
    const frequency: number[] = new Array(26).fill(0);

    // Populate the frequency array with counts for each character in the string
    for (const char of s) {
        frequency[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Counter for the number of characters having odd frequency
    let oddCount = 0;

    // Calculate the number of characters with odd frequency
    for (const count of frequency) {
        if (count % 2 !== 0) {
            oddCount++;
        }
    }

    // A string can be constructed if the number of odd-frequency characters
    // is less than or equal to k (since each odd count character can start a new palindrome)
    return oddCount <= k;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by a few operations:

1. `len(s)`: This operation takes `O(n)` where `n` is the length of the string `s`.
2. `Counter(s)`: Counting the frequency of each character in the string takes `O(n)` since it goes through the string once.
3. `sum(v & 1 for v in cnt.values())`: This operation iterates over the values in the counter, which in the worst case are as many as `n`, and performs a bitwise AND and summing operation. The iteration is `O(n)` and the bitwise AND and summation are `O(1)` per operation.

Combining these, the overall time complexity is `O(n)`.

Space Complexity

The space complexity is also determined by the data structures and operations used:

1. `Counter(s)`: The counter object stores character counts, and in the worst case, if all characters are unique, it could store `n` key-value pairs. Hence, the space complexity is `O(n)`.

Overall, the space complexity of the given code is `O(n)`.