266. Palindrome Permutation

Easy | Bit Manipulation | Hash Table | String

Leetcode Link

Problem Description

The problem asks to determine whether a permutation of the given string `s` can form a palindrome. A palindrome is a word, phrase, number, or other sequences of characters which reads the same forward and backward (ignoring spaces, punctuation, and capitalization). For a string to have a palindromic permutation, it must satisfy certain conditions based on the count of each character. Specifically:

- If the length of the string is even, every character must appear an even number of times.
- If the length of the string is odd, only one character can appear an odd number of times, while all others must appear an even number of times.

The goal is to return `true` if at least one permutation of the string could potentially be a palindrome. Otherwise, the function should return `false`.

Intuition

A palindrome has a characteristic symmetry. For strings of even length, this means that each character must have a matching pair. For strings of odd length, there can be one unique character (without a pair), but all other characters must still have matching pairs.

The intuition behind the solution is based on this observation. We can count the frequency of each character in the string using Python's `Counter` class from the `collections` module, which will give us a dictionary where the keys are the characters and the values are the counts. We then check the parity (even or odd) of these counts.

In the solution, the expression `v & 1` will give `1` for odd counts and `0` for even counts of characters, because the bitwise AND operation with `1` will check the least significant bit of the count (odd numbers have a least significant bit of `1`).

By summing these up, we're effectively counting how many characters appear an odd number of times. For the string to have a palindromic permutation, this sum must be less than `2` (allowing for the middle character in an odd-length palindrome). If this condition is satisfied, the method returns `true`; otherwise, it returns `false`.

Solution Approach

The solution uses a hash table to count the occurence of each character in the string. This is done using Python's `Counter` class from the `collections` module, which upon passing the string, it will return a dictionary where keys are characters from the string and values are the counts of those characters.

```
Counter(s).values()
```

Once we have the character counts, we need to determine how many characters have odd counts. In a palindromic string, at most one character can appear an odd number of times (which would be the middle character in an odd length palindrome).

We can find the number of characters with odd counts using the following line of code:

```
sum(v & 1 for v in Counter(s).values())
```

Here, `v & 1` is a bitwise AND operation that returns `1` if `v` is odd and `0` if `v` is even. This is because odd numbers have a least significant bit of `1`. Running this operation in a comprehension and summing the results, we get the total number of characters with odd counts.

The condition for the string to potentially be a palindrome (after a permutation) is for the sum of odd counts to be less than `2`. This makes sense because if the string length is even, there should be no characters with odd counts, and if the string length is odd, there should be only one.

The final return statement in the code checks this condition:

```
return sum(v & 1 for v in Counter(s).values()) < 2
```

If the sum is less than `2`, the function returns `true`, indicating that a palindromic permutation is possible. If not, it returns `false`.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider the string `s = "civic"`.

First, we pass this string to the `Counter` class to count the occurrences of each character:

```
from collections import Counter
character_counts = Counter("civic").values() # character_counts will be { 'c': 2, 'i': 2, 'v': 1 }
```

Here are the counts we get:

- 'c' appears 2 times
- 'i' appears 2 times
- 'v' appears 1 time

Now, we need to check how many characters have odd counts. Since 'c' and 'i' both appear an even number of times and 'v' appears an odd number of times, we will have:

```
sum(v & 1 for v in Counter("civic").values()) # This will compute to 1
```

The bitwise AND operation (`v & 1`) will return `1` for the count of 'v' as it's odd and `0` for the counts of 'c' and 'i' since those are even. Adding these up gives us `1`.

Since the sum of odd counts is `1`, which is less than `2`, the string `s = "civic"` has a palindromic permutation. The permutation `"civic"` itself is a palindrome.

Hence, according to the final check:

```
return sum(v & 1 for v in Counter(s).values()) < 2
```

The function will return `true`, indicating that it's indeed possible to permute the string `s = "civic"` into a palindrome.

Python Solution

```
from collections import Counter

class Solution:
    def canPermutePalindrome(self, s: str) -> bool:
        # Count the occurrences of each character in the string
        char_count = Counter(s)

        # Calculate the number of characters that appear an odd number of times
        odd_count = sum(value % 2 for value in char_count.values())

        # A string can be permuted into a palindrome if it has at most one character
        # with an odd occurrence count (which would be the middle character)
        return odd_count < 2

```

Java Solution

```
class Solution {
    // This method checks if a permutation of the input string can form a palindrome
    public boolean canPermutePalindrome(String s) {
        // Frequency array to store the count of each character assuming only lowercase English letters
        int[] charCount = new int[26];

        // Iterate over each character in the string
        for (char character : s.toCharArray()) {
            // Increment the count for this character
            charCount[character - 'a']++;
        }

        // Counter for the number of characters that appear an odd number of times
        int oddCount = 0;

        // Iterate over the frequency array
        for (int count : charCount) {
            // If the count of a character is odd, increment the oddCount
            if (count % 2 == 1) {
                oddCount++;
            }
        }

        // A string can form a palindrome if there is no more than one character that appears an odd number of times
        return oddCount < 2;
    }
}

```

C++ Solution

```
class Solution {
public:
    // Function to check if a permutation of the input string can form a palindrome
    bool canPermutePalindrome(string s) {
        // Create a vector to count occurrences of each lowercase letter
        vector<int> charCounts(26, 0); // Initialize counts to 0 for 'a'-'z'

        // Count occurrences of each character in the input string
        for (char c : s) {
            // Increase the count of the current character
            ++charCounts[c - 'a']; // Assuming input string has only lowercase letters
        }

        // Variable to keep track of the number of characters with odd counts
        int oddCount = 0;

        // Iterate through the character counts
        for (int count : charCounts) {
            // If the count is odd, increment the odd count
            oddCount += count % 2; // Count is odd if % 2 is 1
        }

        // A string can be permuted to form a palindrome if there is at most one character with an odd count
        return oddCount < 2;
    }
};

```

Typescript Solution

```
function canPermutePalindrome(s: string): boolean {
    // Create an array to count occurrences of each lowercase letter.
    const charCounts: number[] = new Array(26).fill(0);

    // Iterate over each character in the string.
    for (const char of s) {
        // Increment the count for the current character.
        ++charCounts[char.charCodeAt(0) - 'a'.charCodeAt(0)];
    }

    // Use filter to count how many characters have an odd number of occurrences.
    const oddCounts = charCounts.filter(count => count % 2 === 1).length;

    // For a string to be permutable to a palindrome, there must be
    // at most one character with an odd count (which would be the middle character).
    return oddCounts < 2;
}

```

Time and Space Complexity

The time complexity of the given code snippet is `O(n)`, where `n` is the length of the input string `s`. This is because we're iterating through the string once to create a counter (a dictionary) of the characters, which takes `O(n)` time. Then, we iterate through the values in the counter, which in the worst case contains `n` unique characters, to calculate the sum of the odd counts. However, since the number of unique characters is typically far less than `n`, this is still considered `O(n)` complexity.

The space complexity of the given code snippet is also `O(n)` for the same reason. As we create a counter that could potentially have as many entries as there are characters in the string if all characters are unique. Hence, the space required for the counter can grow linearly with the size of the input string `s`.