2384. Largest Palindromic Number

Medium Topics Hash Table String

Problem Description

You are given a string `num` that consists only of digits. The task is to construct the largest palindromic integer (representing it as a string) by reordering digits taken from `num`. A palindrome is a sequence that reads the same backward as forward (like "121" or "1331"). The resulting palindrome should not have leading zeroes, meaning it should not start with the digit '0' unless the palindrome is simply "0". It is also important to note that you can choose to use some or all the digits from `num`, but you must use at least one digit to construct the palindrome.

Intuition

The intuition behind the solution is to strategically utilize the digits in `num` to create the largest possible palindrome. To achieve this, we consider several factors:

1. The largest digit should be placed in the middle of the palindrome for odd-length palindromes.
2. Even-length palindromes are formed by placing mirrored digits around the center.
3. Leading zeroes can be avoided by ensuring that we don't start the construction of the palindrome with zeroes, except if the largest palindrome is "0" itself.

With this in mind, the following steps are taken in the solution:

1. Count the frequency of each digit in the string.
2. Starting from the largest digit (9) and moving to the smallest (0), look for a digit that has an odd count.
   - This digit, if any, can be used as the central character in an odd-length palindrome.
   - Decrease its count by one and store it.
3. Then, for each digit from 0 to 9, append half of the remaining count of that digit to both sides of the palindrome.
   - This creates the mirrored effect around the center.
4. Finally, remove any leading zeroes (if they are not the only character) from the resulting string to maintain the 'no leading zero' constraint.
5. If the resulting string is empty (which can happen if we start with lots of zeroes), return "0".

By following these steps, we utilize the counts of digits in such a way to maximize the integer value of the resulting palindrome.

Solution Approach

The solution implemented above uses a *greedy* approach and some basic understanding of how palindromes are structured. Here's the step-by-step explanation of the solution:

1. Import the `Counter` class from Python's `collections` module to count the frequency of each digit in the input string `num`. `Counter(num)` provides a dictionary-like object where keys are unique digits from `num`, and the values are counts of those digits.
2. Initialize an empty string `ans` to accumulate the palindrome constructed so far.
3. Iterate through the digits in descending order, from '9' to '0', to find a digit that occurs an odd number of times.
   - When such a digit is found (`cnt[v] % 2`), use it as the central digit of the palindrome (`ans = v`) only if the palindrome is meant to be of odd length. This digit will not have a mirrored counterpart.
   - Decrease the count of that digit by 1 and break the loop as we are interested in only one such digit for the center of the palindrome.
4. Iterate through the digits again, this time starting from '0'. For each digit `v`:
   - Check if the digit has a non-zero count in `cnt`.
   - Divide the count by 2 (`cnt[v] //= 2`) because we place half on each side of the palindrome.
   - Create a string `s` by repeating the digit `v`, `cnt[v]` times.
   - Update the palindrome string `ans` by placing string `s` before and after the current `ans`.
5. Before returning the result, use `ans.strip('0')` to ensure that there are no leading zeroes, unless the palindrome is '0'. If `ans` is an empty string at this point (meaning it was made up of only zeroes), simply return '0'.

In summary, the algorithm employs a counter to keep track of frequency, a *greedy* approach for constructing the largest palindrome from the center outwards, and simple string manipulation to assemble the final palindromic string, making sure to adhere to the constraints laid out in the problem description.

Example Walkthrough

Let's illustrate the solution approach with a small example where the input string `num` is "310133".

Following the solution steps:

1. We count the frequency of each digit:

```
'1': 1, '3': 2, '0': 1
```

2. We find that digit '1' occurs an odd number of times. It can be the central digit of the palindrome.

   After this step, our palindrome under construction looks like this:

```
"_ _ 1 _ _"
```

3. Next, we iterate from digit '9' to '0' and add the digits in pairs around the central digit:

   - We skip '9', '8', '7', '6', '5', '4', and '2' because they are not present in `num` in zero.
   - For digit '3' (which has a count of 2), we will take one to place on each side of '1'.

   After updating, the palindrome is:

```
"_ 3 1 3 _"
```

   - We've added all digits where possible. No further digits can be placed.

4. There's no need to trim leading zeroes since the palindrome does not start or end with '0'.
5. If our constructed palindrome were empty (e.g., if `num` was just "0"s), we would return "0".

As a result, for the given `num` "310133", the largest palindromic integer we can construct is "313".

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def largestPalindromic(self, num: str) -> str:
        # Create a counter for the digits in the input string
        digit_count = Counter(num)
        # Initialize the middle character of the palindrome as empty
        middle = ''
        # Initialize the first half of the palindrome as empty
        half_palindrome = ''

        # Loop from the largest digit to the smallest to construct the first half of the palindrome
        for digit in range(9, -1, -1):
            char = str(digit)
            if digit_count[char] > 1:
                # Add half of the even occurrences of the digit to the first half of the palindrome
                # Only add non-zero digits or zero digits if other digits are already in the half_palindrome
                if char != '0' or (char == '0' and half_palindrome.__length__ > 0):
                    half_palindrome += char * (digit_count[char] // 2)
                # Leave one odd occurrence for potential use in the middle
                digit_count[char] %= 2
            if digit_count[char] == 1 and middle == '':
                # Use the first odd-occurring digit as the middle character
                middle = char
                digit_count[char] -= 1

        # Reverse the first half and concatenate with the middle and first half to form the palindrome
        palindrome = half_palindrome + middle + half_palindrome[::-1]

        # If after forming the palindrome, it is empty or only zeros, return '0'
        if not palindrome or palindrome.lstrip('0') == '':
            return '0'
        else:
            return palindrome
```

Java

```java
import java.util.HashMap;
import java.util.Map;

public class Solution {
    public String largestPalindromic(String num) {
        // Create a counter for the digits in the input string
        Map<Character, Integer> digitCount = new HashMap<>();
        for (char c : num.toCharArray()) {
            digitCount.put(c, digitCount.getOrDefault(c, 0) + 1);
        }

        // Initialize the middle character of the palindrome as empty
        String middle = "";
        // Initialize the first half of the palindrome as empty
        StringBuilder halfPalindrome = new StringBuilder();

        // Loop from the largest digit to the smallest to construct the first half of the palindrome
        for (char digit = '9'; digit >= '0'; digit--) {
            if (digitCount.getOrDefault(digit, 0) > 1) {
                // Add half of the even occurrences of the digit to the first half of the palindrome
                // Only add non-zero digits or zero digits if other digits are already in the halfPalindrome
                if (digit != '0' || (digit == '0' && halfPalindrome.length() > 0)) {
                    char[] chars = new char[digitCount.get(digit) / 2];
                    java.util.Arrays.fill(chars, digit);
                    halfPalindrome.append(new String(chars));
                }
                // Leave any odd occurrence for potential use in the middle
                digitCount.put(digit, digitCount.get(digit) % 2);
            }
            if (digitCount.containsKey(digit) && digitCount.get(digit) == 1 && middle.isEmpty()) {
                // Use the first odd-occurring digit as the middle character
                middle = Character.toString(digit);
                digitCount.put(digit, digitCount.get(digit) - 1);
            }
        }

        // Reverse the first half and concatenate with the middle and first half to form the palindrome
        String palindrome = halfPalindrome.toString() + middle + halfPalindrome.reverse().toString();

        // If after forming the palindrome, it is empty or only zeros, return '0'
        if (palindrome.isEmpty() || palindrome.replace("0", "").isEmpty()) {
            return "0";
        } else {
            return palindrome;
        }
    }
}
```

C++

```cpp
#include <string>
#include <map>
#include <algorithm>

class Solution {
public:
    std::string largestPalindromic(std::string num) {
        // Create a counter for the digits in the input string
        std::map<char, int> digitCount;
        for (char c : num) {
            digitCount[c]++;
        }

        // Initialize the middle character of the palindrome as empty
        std::string middle = "";
        // Initialize the first half of the palindrome as empty
        std::string halfPalindrome = "";

        // Loop from the largest digit to the smallest to construct the first half of the palindrome
        for (char digit = '9'; digit >= '0'; digit--) {
            if (digitCount[digit] > 1) {
                // Add half of the even occurrences of the digit to the first half of the palindrome
                // Only add non-zero digits or zero digits if other digits are already in the halfPalindrome
                if (digit != '0' || halfPalindrome.empty()) {
                    halfPalindrome += std::string(digitCount[digit] / 2, digit);
                }
                // Leave any odd occurrence for potential use in the middle
                digitCount[digit] %= 2;
            }
            if (digitCount[digit] == 1 && middle.empty()) {
                // Use the first odd-occurring digit as the middle character
                middle = digit;
                digitCount[digit] -= 1;
            }
        }

        // Reverse the first half and concatenate with the middle and first half to form the palindrome
        std::string reversedHalf = halfPalindrome;
        std::reverse(reversedHalf.begin(), reversedHalf.end());
        std::string palindrome = halfPalindrome + middle + reversedHalf;

        // If after forming the palindrome, it is empty or only zeros, return '0'
        if (palindrome.empty() || palindrome.find_first_not_of('0') == std::string::npos) {
            return "0";
        } else {
            return palindrome;
        }
    }
};
```

TypeScript

```typescript
function largestPalindromic(num: string): string {
    // Create a counter for the digits in the input string
    const digitCounts: { [key: string]: number } = {};
    for (const c of num) {
        digitCounts[c] = (digitCounts[c] || 0) + 1;
    }

    // Initialize the middle character of the palindrome as empty
    let middle = "";
    // Initialize the first half of the palindrome as empty
    let halfPalindrome = "";

    // Loop from the largest digit to the smallest to construct the first half of the palindrome
    for (let digit = 9; digit >= 0; digit--) {
        const char = digit.toString();
        if (digitCounts[char] > 1) {
            // Add half of the even occurrences of the digit to the first half of the palindrome
            // Only add non-zero digits or zero digits if other digits are already in the halfPalindrome
            if (char !== '0' || (char === '0' && halfPalindrome.length > 0)) {
                halfPalindrome += char.repeat(Math.floor(digitCounts[char] / 2));
            }
            // Leave any odd occurrence for potential use in the middle
            digitCounts[char] %= 2;
        }
        if (digitCounts[char] === 1 && middle === "") {
            // Use the first odd-occurring digit as the middle character
            middle = char;
            digitCounts[char] -= 1;
        }
    }

    // Reverse the first half and concatenate with the middle and first half to form the palindrome
    const palindrome = halfPalindrome + middle + [...halfPalindrome].reverse().join("");

    // If after forming the palindrome, it is empty or only zeros, return '0'
    return palindrome.length === 0 || parseInt(palindrome) === 0 ? "0" : palindrome;
}

console.log(largestPalindromic("00009")); // Output: "9"
```

```python
from collections import Counter

class Solution:
    def largestPalindromic(self, num: str) -> str:
        # Create a counter for the digits in the input string
        digit_count = Counter(num)
        # Initialize the middle character of the palindrome as empty
        middle = ''
        # Initialize the first half of the palindrome as empty
        half_palindrome = ''

        # Loop from the largest digit to the smallest to construct the first half of the palindrome
        for digit in range(9, -1, -1):
            char = str(digit)
            if digit_count[char] > 1:
                # Add half of the even occurrences of the digit to the first half of the palindrome
                # Only add non-zero digits or zero digits if other digits are already in the half_palindrome
                if char != '0' or half_palindrome:
                    half_palindrome += char * (digit_count[char] // 2)
                # Leave one odd occurrence for potential use in the middle
                digit_count[char] %= 2
            if digit_count[char] == 1 and middle == '':
                # Use the first odd-occurring digit as the middle character
                middle = char
                digit_count[char] -= 1

        # Reverse the first half and concatenate with the middle and first half to form the palindrome
        palindrome = half_palindrome + middle + half_palindrome[::-1]

        # If after forming the palindrome, it is empty or only zeros, return '0'
        if not palindrome or palindrome.lstrip('0') == '':
            return '0'
        else:
            return palindrome
```

Time and Space Complexity

The given code snippet aims to find the largest palindromic number that can be formed by rearranging the digits of the given number `num`. The code uses a Counter to store the frequency of each digit and then constructs the palindrome.

- **Time Complexity:**

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

  1. Creating a Counter from the string `num` takes $O(n)$ where n is the length of `num`, as each character in the string needs to be read once.
  2. The first loop runs a constant 10 times (digits 9 to 0) which is $O(1)$ since it doesn't depend on the length of `num`.
  3. The second loop also runs a constant 10 times, and inside this loop, it performs string concatenation. Assuming that the Python string concatenation in this case takes $O(1)$ time (where n is the length of the string being concatenated), the maximum length of i will be $n/2$. Therefore, the overall work done here is proportional to n.

  Since these operations occur sequentially, the time complexity is the sum of their individual complexities, resulting in $O(n)$.

- **Space Complexity:**

  The space complexity is determined by:

  1. The Counter `cnt`, which can potentially store a count for each different digit, thus having a space complexity of $O(10)$ or $O(1)$ since the number of possible different digits (0-9) is constant and does not grow with n.
  2. The string `ans`, which can grow up to a length of n in the worst case when each character is identical. Thus it has a space complexity of $O(n)$.

  Therefore, the total space complexity is $O(n)$, where n is the length of the input number `num`.