2384. Largest Palindromic Number

Medium Topics Hash Table String

Leetcode Link

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": 2, "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 "_ _ 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 their count in `num` is 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:

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

   - Since '1' is already used as the central digit, we move to '0'. There's one '0', so we cannot form a pair (it's left out).

     Our palindrome is currently:

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

   - We've added all digits were 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".

Python Solution

```python
from collections import Counter

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

        # Start from the largest digit and find the highest odd occurring digit
        for digit in range(9, -1, -1):
            char = str(digit)
            # If the count of the digit is odd, it can be used in the middle
            if digit_count[char] % 2:
                middle = char
                # Decrease the count as it's used in the middle
                digit_count[char] -= 1
                break

        # Initialize the first half of the palindrome as empty
        half_palindrome = ''

        # Loop through the digits to construct the first half of the palindrome
        for digit in range(10):
            char = str(digit)
            # Add half of the digits left after possibly using one in the middle
            if digit_count[char]:
                # Half the count as the other half will be mirrored
                digit_count[char] //= 2
                # Construct a string with half the count of the current digit
                half = digit_count[char] * char
                # Concatenate to the first half of the palindrome
                half_palindrome = half + half_palindrome

        # If the result is all zeros, strip leading zeros, otherwise return '0'
        return half_palindrome.strip('0') or '0'
```

Java Solution

```java
class Solution {

    public String largestPalindromic(String num) {
        // Count the occurrences of each digit
        int[] count = new int[10];
        for (char c : num.toCharArray()) {
            count[c - '0']++;
        }

        // Track the middle digit which may be placed in the center of palindrome
        String middle = "";
        for (int i = 9; i >= 0; i--) {
            // If digit i has an odd count, use one of this digit in the middle
            if (count[i] % 2 == 1) {
                middle = Integer.toString(i);
                count[i]--;
                break;
            }
        }

        // Build the first half of the palindrome
        StringBuilder firstHalf = new StringBuilder();
        for (int i = 9; i >= 0; --i) {
            if (count[i] > 0) {
                // Each digit should be added count[i]/2 times
                firstHalf.append(String.valueOf(i).repeat(count[i] / 2));
            }
        }

        // Remove leading zeros from the first half of the palindrome
        while (firstHalf.length() > 1 && firstHalf.charAt(firstHalf.length() - 1) == '0') {
            firstHalf.deleteCharAt(firstHalf.length() - 1);
        }

        // Reverse the first half to create the second half
        String secondHalf = firstHalf.reverse().toString();

        // If no digits are usable, return "0", else construct the full palindrome by combining both halves and the middle digit
        String palindrome = secondHalf + middle + firstHalf;
        return palindrome.isEmpty() ? "0" : palindrome;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    string largestPalindromic(string num) {
        vector<int> count(10); // Count digit frequencies in the input string
        for (char digit : num) {
            ++count[digit - '0']; // Increment the count for the current digit
        }

        string middle = ""; // To store middle character of palindrome
        // Check in reverse order which is the highest digit that can be put in the middle of the palindrome
        for (int i = 9; i >= 0; --i) {
            if (count[i] % 2 != 0) { // If there is an odd occurrence of the digit
                middle = to_string(i); // Store this digit as the middle character
                --count[i]; // Remove one occurrence of this odd digit from the count
                break; // Only one odd-count digit can be in the middle of a palindrome
            }
        }

        string leftHalf = ""; // To store the first half of the palindrome
        // Construct left half of the palindrome with remaining digits
        for (int i = 9; i >= 0; ++i) {
            if (count[i]) { // For each digit with a count
                count[i] /= 2; // We use half of them in our left half (mirror will be the other half)
                while (count[i]--) {
                    leftHalf += (i + '0'); // Append digit 'i' count[i] times to the left half
                }
            }
        }

        // If the left half is not empty and the leading character is '0', we should remove it
        // (since we can't have leading zeros) and make sure it's not the number '0' itself.
        while (leftHalf.size() > 1 && leftHalf.back() == '0') {
            leftHalf.pop_back(); // Remove trailing zeros from the left half
        }

        string rightHalf = leftHalf; // The right half is a mirror of the left half
        // Reverse the right half to arrange the left half
        reverse(rightHalf.begin(), rightHalf.end());

        // Concatenate the left half, middle digit (if any), and right half
        string largestPalindrome = rightHalf + middle + leftHalf;
        // If largestPalindrome is empty, it means the string was comprised of all zeros
        return largestPalindrome.empty() ? "0" : largestPalindrome;
    }
};
```

Typescript Solution

```typescript
function largestPalindromic(num: string): string {
    // Initialize an array to keep track of the digit counts
    const digitsCount = new Array(10).fill(0);

    // Count the occurrences of each digit in the input number
    for (const digit of num) {
        digitsCount[digit]++;
    }

    // Ensure there is at most one digit with an odd count
    while (digitsCount.reduce((acc, count) => count % 2 === 1 ? acc + 1 : acc, 0) > 1) {
        for (let i = 0; i < 10; i++) {
            if (digitsCount[i] % 2 === 1) {
                digitsCount[i]--;
                break;
            }
        }
    }

    let result: number[] = [];
    let oddDigitIndex = -1;

    // Build the left part of the palindrome by adding half of the even-count digits
    for (let i = 9; i >= 0; i--) {
        if (digitsCount[i] % 2 === 1) {
            // Save the odd-count digit (only one is allowed)
            oddDigitIndex = i;
            digitsCount[i]--;
        }
        // Push half the occurrences of the current digit to the result array
        result.push(...new Array(digitsCount[i] >> 1).fill(i));
    }

    // Insert the odd-count digit (if any) in the middle
    if (oddDigitIndex !== -1) {
        result.push(oddDigitIndex);
    }

    // Determine the number of elements in the result
    const resultLength = result.length;

    // Remove leading zeros if any and finalize the palindrome
    for (let i = 0; i < resultLength; i++) {
        // Find the first non-zero digit
        if (result[i] !== 0) {
            result = result.slice(i);
            // Add the second half of the palindrome
            if (oddDigitIndex !== -1) {
                result.push(...result.slice(0, resultLength - i - 1).reverse());
            } else {
                result.push(...result.slice(0, resultLength - i).reverse());
            }
            return result.join('');
        }
    }

    // If the input was all zeros, return '0'
    return '0';
}
```

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 $O(k)$ time (where k is the length of the string being concatenated), the maximum length of s 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`.