2384. Largest Palindromic Number

String

Hash Table

```
Problem Description
```

Greedy

Medium

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.

You are given a string num that consists only of digits. The task is to construct the largest palindromic integer (representing it as a

this, we consider several factors:

Intuition The intuition behind the solution is to strategically utilize the digits in num to create the largest possible palindrome. To achieve

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

1. The largest digit should be placed in the middle of the palindrome for odd-length palindromes.

is "0" itself.

2. Even-length palindromes are formed by placing mirrored digits around the center.

- With this in mind, the following steps are taken in the solution: 1. Count the frequency of each digit in the string.
- Decrease its count by one and store it.
- This digit, if any, can be used as the central character in an odd-length palindrome.
- 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.

2. Starting from the largest digit (9) and moving to the smallest (0), look for a digit that has an odd count.

- 5. If the resulting string is empty (which can happen if we start with lots of zeroes), return "0".
- Solution Approach The solution implemented above uses a greedy approach and some basic understanding of how palindromes are structured.

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

Here's the step-by-step explanation of the solution:

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.

Initialize an empty string ans to accumulate the palindrome constructed so far. Iterate through the digits in descending order, from '9' to '0', to find a digit that occurs an odd number of times. 3.

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. Iterate through the digits again, this time starting from '0'. For each digit v:

• 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

Before returning the result, use ans.strip('0') to ensure that there are no leading zeroes, unless the palindrome is '0'. If ans

 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.

Check if the digit has a non-zero count in cnt.

Update the palindrome string ans by placing string s before and after the current ans.

is an empty string at this point (meaning it was made up of only zeroes), simply return '0'.

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

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

- 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.
- We count the frequency of each digit: '1': 1, '3': 2, '0': 1

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).

"_ _ 1 _ _"

"_ 3 1 3 _"

Example Walkthrough

Following the solution steps:

We skip '9', '8', '7', '6', '5', '4', and '2' because their count in num is zero.

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

Our palindrome is currently: "3 1 3"

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

After updating, the palindrome is:

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

For digit '3' (which has a count of 2), we will take one to place on each side of '1'.

Solution Implementation

There's no need to trim leading zeroes since the palindrome does not start or end with '0'.

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".

digit_count = Counter(num) # Initialize the middle character of the palindrome as empty middle = ''

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:

if char != '0' or (char == '0' and half_palindrome != ''):

half_palindrome += char * (digit_count[char] // 2)

Leave any odd occurrence for potential use in the middle

Use the first odd-occurring digit as the middle character

If after forming the palindrome, it is empty or only zeros, return '0'

Add half of the even occurrences of the digit to the first half of the palindrome

Reverse the first half and concatenate with the middle and first half to form the palindrome

Only add non-zero digits or zero digits if other digits are already in the half_palindrome

C++

public:

#include <string>

class Solution {

#include <algorithm>

std::string largestPalindromic(std::string num) {

std::map<char, int> digitCount;

std::string halfPalindrome = "";

if (digitCount[digit] > 1) {

digitCount[digit] %= 2;

for (char c : num) {

digitCount[c]++;

std::string middle = "";

// Create a counter for the digits in the input string

// Initialize the first half of the palindrome as empty

for (char digit = '9'; digit >= '0'; digit--) {

// Initialize the middle character of the palindrome as empty

// Loop from the largest digit to the smallest to construct the first half of the palindrome

if (digit != '0' || (digit == '0' && !halfPalindrome.empty())) {

// Leave any odd occurrence for potential use in the middle

halfPalindrome += std::string(digitCount[digit] / 2, digit);

// 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

#include <map>

Python

class Solution:

from collections import Counter

half_palindrome = ''

def largestPalindromic(self, num: str) -> str:

digit_count[char] %= 2

digit_count[char] -= 1

middle = char

Create a counter for the digits in the input string

Initialize the first half of the palindrome as empty

if digit_count[char] == 1 and middle == '':

if not palindrome or palindrome.lstrip('0') == '':

palindrome = half_palindrome + middle + half palindrome[::-1]

```
return '0'
       else:
            return palindrome
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.containsKey(digit) && digitCount.get(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' || (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;
```

```
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
  function largestPalindromic(num: string): string {
      // Create a counter for the digits in the input string
      const digitCount: { [key: string]: number } = {};
      for (const c of num) {
          digitCount[c] = (digitCount[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 (digitCount[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(digitCount[char] / 2));
              // Leave any odd occurrence for potential use in the middle
              digitCount[char] %= 2;
          if (digitCount[char] === 1 && middle === "") {
              // Use the first odd-occurring digit as the middle character
              middle = char;
              digitCount[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"
from collections import Counter
class Solution:
   def largestPalindromic(self, num: str) -> str:
```

```
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:
```

else:

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 0(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

number of possible different digits (0-9) is constant and does not grow with n.

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

Create a counter for the digits in the input string

Initialize the first half of the palindrome as empty

Initialize the middle character of the palindrome as empty

Loop from the largest digit to the smallest to construct the first half of the palindrome

if char != '0' or (char == '0' and half_palindrome != ''):

half_palindrome += char * (digit_count[char] // 2)

Leave any odd occurrence for potential use in the middle

Use the first odd-occurring digit as the middle character

If after forming the palindrome, it is empty or only zeros, return '0'

Add half of the even occurrences of the digit to the first half of the palindrome

Reverse the first half and concatenate with the middle and first half to form the palindrome

Only add non-zero digits or zero digits if other digits are already in the half_palindrome

digit_count = Counter(num)

for digit in range(9, -1, -1):

middle = char

return '0'

return palindrome

Time and Space Complexity

if digit_count[char] > 1:

digit_count[char] %= 2

digit_count[char] -= 1

The space complexity is determined by:

if digit_count[char] == 1 and middle == '':

if not palindrome or palindrome.lstrip('0') == '':

palindrome = half_palindrome + middle + half_palindrome[::-1]

char = str(digit)

half_palindrome = ''

middle = ''

- concatenation in this case takes 0(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:**

1. The Counter cnt, which can potentially store a count for each different digit, thus having a space complexity of 0(10) or 0(1) since the

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